Improving the inheritance concept of Dynamic Meta Modelling

Diploma Thesis

Daniela Fisseler

Paderborn, August 2007

University of Paderborn
# Contents

1 Introduction ............................................. 1

2 Dynamic Meta Modelling (DMM) .......................... 5
   2.1 Introduction to Dynamic Meta Modelling ................. 5
   2.2 Graph Transformation .................................. 7
      2.2.1 Graphs ......................................... 7
      2.2.2 Typed Graph ................................. 9
      2.2.3 Inheritance .................................. 10
      2.2.4 Graph transformation rules .................... 12
   2.3 Concepts of Dynamic Meta Modelling ....................... 13
      2.3.1 Graphs in DMM ................................ 13
      2.3.2 DMM Rules .................................. 15
      2.3.3 DMM Rule Invocation .......................... 18
      2.3.4 DMM Rule Signature ........................... 20
      2.3.5 DMM Final Rules .............................. 21
   2.4 DMM Example ........................................ 22

3 Inheritance in Object-Orientation ....................... 27
   3.1 Object-Orientation .................................. 27
   3.2 Inheritance ......................................... 28
      3.2.1 Class Inheritance ........................... 28
      3.2.2 Subtyping .................................. 30
   3.3 Polymorphism ....................................... 33

4 Improving the inheritance concept of DMM ............ 37
   4.1 Current State ....................................... 37
      4.1.1 Resemblances between Java and DMM ............ 38
      4.1.2 Differences between Java and DMM ............. 40
   4.2 Extension of a Visual Modelling Language (VML) .... 41
# List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1</td>
<td>Current Extension mechanism in DMM</td>
<td>3</td>
</tr>
<tr>
<td>1.2</td>
<td>New Extension mechanism in DMM</td>
<td>4</td>
</tr>
<tr>
<td>2.1</td>
<td>Overview of the DMM approach</td>
<td>6</td>
</tr>
<tr>
<td>2.2</td>
<td>Type Graph</td>
<td>10</td>
</tr>
<tr>
<td>2.3</td>
<td>Typed Graph</td>
<td>10</td>
</tr>
<tr>
<td>2.4</td>
<td>Type Graph with inheritance</td>
<td>12</td>
</tr>
<tr>
<td>2.5</td>
<td>Abstract Transitive Closure</td>
<td>12</td>
</tr>
<tr>
<td>2.6</td>
<td>Concrete Transitive Closure</td>
<td>12</td>
</tr>
<tr>
<td>2.7</td>
<td>Basic Graph Transformation rule</td>
<td>13</td>
</tr>
<tr>
<td>2.8</td>
<td>Matching of a rule without NAC</td>
<td>14</td>
</tr>
<tr>
<td>2.9</td>
<td>Matching of a rule with NAC</td>
<td>14</td>
</tr>
<tr>
<td>2.10</td>
<td>Rule without merging</td>
<td>18</td>
</tr>
<tr>
<td>2.11</td>
<td>Merged rule</td>
<td>18</td>
</tr>
<tr>
<td>2.12</td>
<td>Illustration of rule signature and invocation</td>
<td>21</td>
</tr>
<tr>
<td>2.13</td>
<td>Type Graph</td>
<td>22</td>
</tr>
<tr>
<td>2.14</td>
<td>Rule node.enqueue()</td>
<td>23</td>
</tr>
<tr>
<td>2.15</td>
<td>Rule node.enqueue when there is at least one other token in the queue</td>
<td>23</td>
</tr>
<tr>
<td>2.16</td>
<td>Rule node.enqueue when there is no other token in the queue</td>
<td>23</td>
</tr>
<tr>
<td>2.17</td>
<td>Rule node.dequeue when there is at least one other token in the queue</td>
<td>24</td>
</tr>
<tr>
<td>2.18</td>
<td>Rule node.dequeue when there is no other token in the queue</td>
<td>24</td>
</tr>
<tr>
<td>2.19</td>
<td>State S₀: Start Graph</td>
<td>24</td>
</tr>
<tr>
<td>2.20</td>
<td>State S₁: Graph after first rule invocation n₂.enqueue(t₂:Token) of the rule n₁.flow()*</td>
<td>24</td>
</tr>
<tr>
<td>2.21</td>
<td>State S₂: Graph after second rule invocation n₁.dequeue(t₂:Token) of the rule n₁.flow()*</td>
<td>24</td>
</tr>
<tr>
<td>2.22</td>
<td>State S₃: Graph after first rule invocation n₃.enqueue(t₁:Token) of the rule n₂.flow()* (First Token)</td>
<td>25</td>
</tr>
<tr>
<td>2.23</td>
<td>State S₄: Graph after second rule invocation n₂.dequeue(t₁:Token) of the rule n₂.flow()* (First Token)</td>
<td>25</td>
</tr>
</tbody>
</table>
2.24 State $S_5$: Graph after first rule invocation n3.enqueue(t2:Token) of the rule $n2.tok()^*$ (Second Token) ........................................ 25
2.25 State $S_6$: Graph after second rule invocation n2.dequeue(t2:Token) of the rule $n2.tok()^*$ (First Token) ........................................ 26
2.26 Transition System ........................................ 26
3.1 Single Inheritance ........................................ 29
3.2 Multiple Inheritance ........................................ 29
3.3 Subtyping of functions with the contravariance rule ........... 32
3.4 Coherence of typing and subtyping, and classes and inheritance .................. 33
3.5 Example of dynamic polymorphism in Java .................................. 34
3.6 Class B inherits from Class A ........................................ 34
3.7 Different kinds of polymorphism ........................................ 36
4.1 Type Graph which is to be extended ........................................ 42
4.2 Extended Type Graph (1) ........................................ 42
4.3 Extended Type Graph (2) ........................................ 42
4.4 Type Graph (Example 2) ........................................ 44
4.5 Rule node.tok()^* ........................................ 44
4.6 Extended Type Graph with Inheritance ........................................ 44
4.7 Extended Type Graph with ForkNodes and ForkEdges ................. 48
4.8 Rule Forknode.tok()^* with only one outgoing edge ................. 49
4.9 Rule Forknode.tok()^* with more than one outgoing edge ................. 49
4.10 Rule Forknode.tok(e1:ForkEdge) with at least one edge left which is not processed yet ........................................ 50
4.11 Rule Forknode.tok(e1:ForkEdge) with no edges left which are not processed ........................................ 50
4.12 Rule node.enqueueCopy(originaltok:Token) with at least one other token in the queue of node ........................................ 51
4.13 Rule node.enqueueCopy(originaltok:Token) with no tokens in the queue of node ........................................ 51
4.14 Bad example of overwriting when only constraining the signature ................. 52
4.15 Bad example of overwriting when only constraining signature and precondition ................. 54
4.16 node.tok()^* with invocations ........................................ 56
4.17 forknode.tok()^* with invocations ........................................ 56
4.18 possible node.tok()^* without invocations ........................................ 56
4.19 possible forknode.tok()^* without invocations ........................................ 56
4.20 Left hand side of a possible forknode.tok()^* without invocations ........................................ 57
4.21 Left hand side of a possible forknode.tok()^* without invocations ........................................ 57
4.22 Right hand side of a possible forknode.tok()^* without invocations ........................................ 57
4.23 Right hand side of a possible forknode.tok()^* without invocations ........................................ 57
4.24 Rule C.DoSth() .................................................. 59
4.25 Rule ChildOfC.DoSth() ............................................ 59
4.26 Instance graph with an element of type D ..................... 60
4.27 Instance graph without an element of type D ................. 60
4.28 Rule A.accomplishTask() I ...................................... 61
4.29 Rule A.accomplishTask() II ..................................... 61
4.30 Rule ChildOfA.accomplishTask() ................................ 61
4.31 Rule A.doSth() is overwritten by rule B.doSth() ............. 62

6.1 GR OOVE state graph with a non-empty invocation stack .... 73
6.2 GR OOVE state graph .............................................. 76
6.3 GR OOVE auxiliary rule graph for the small-step doSth() hierarchy level 3 77
6.4 GR OOVE small-step rule doSth() ................................ 78
6.5 GR OOVE auxiliary rule for the small-step rule doSth() hierarchy level 2 . . . 78
6.6 GR OOVE auxiliary rule for the small-step rule doSth() when doSth() could not match .................................................. 79
6.7 GR OOVE auxiliary rule for the small-step rule doSth() when doSth() could not match .................................................. 79
6.8 GR OOVE auxiliary rule for the small-step rule doSth() when doSth() could match only one time ........................................ 80
6.9 GR OOVE auxiliary rule for the small-step rule doSth() when doSth() could match only one time ........................................ 80
6.10 GR OOVE auxiliary rule for the small-step rule doSth() hierarchy level 1 . . . 81
6.11 GR OOVE auxiliary rule for big-step rules in the lowest hierarchy level .... 83
6.12 GR OOVE start graph .............................................. 84
6.13 GR OOVE rule node.flow() ...................................... 85
6.14 GR OOVE rule forknode.flow() .................................. 85
6.15 GR OOVE rule forknode.flow() .................................. 85
6.16 GR OOVE auxiliary rule for the rule flow() in the lowest hierarchy level . . . 86
6.17 GR OOVE auxiliary rule for the rule flow() when the rule could not match . . 86
6.18 GR OOVE auxiliary rule for the rule flow() when the rule could not match . . 86
6.19 GR OOVE auxiliary rule for the rule flow() when the first rule matched and the second did not ........................................ 87
6.20 GR OOVE auxiliary rule for the rule flow() when the first rule matched and the second did not ........................................ 87
6.21 GR OOVE auxiliary rule for the rule flow() when none of the forknode.flow() rules matched ........................................ 87
6.22 GR OOVE auxiliary rule for the rule flow() when none of the flow() rules matched ........................................ 87
Chapter 1

Introduction

In most instances, corporate software development is a complex task. Constructing models for a problem might be quite intricate, just as programming a solution. Therefore it is unlikely that one single person can do all the work by him- or herself unless it is a very small project. The people participating might be the client, users, domain experts, programmers, software-engineers among others. Those people have different backgrounds, different levels of knowledge, and they see their environment differently. It is likely that misunderstandings occur. Therefore communication is an important issue to minimize those misunderstandings. Tools and methods used should be easy to comprehend and should be understood in same way by everyone.

In the field of modelling, Visual Modelling Languages (VMLs) can be very helpful. As the name implies, graphical notations are used, which are relatively easy to understand. The most familiar example of a VML is probably the Unified Modeling Language (UML) which was defined and standardized by the Object Management Group (OMG), meanwhile in version 2.1.1. The UML consists of several diagram types, such as class diagrams or activity diagrams, which can be used to model different aspects during the software development process.

But as mentioned above, it is very important that everybody understands those diagrams in the same way. Therefore syntax and semantics need to be well-defined to prevent different interpretations. To define the abstract syntax of a modelling language a meta model can be used, mostly in terms of something resembling UML class diagrams, and as the case may be in combination with the Object Constraint Language (OCL) (cp. [OMG07]).

Defining the semantics of a modelling language is a lot more difficult, especially when the VML describes not structure but behaviour. Until now, there is no commonly accepted definition technique. In the UML specification, for example, the semantics is defined by textual descriptions. Unfortunately, textual descriptions are not very accurate most of the times. There is always room for different interpretations.
Another approach to define the semantics of a VML is Dynamic Meta Modelling (DMM) which is particularly useful when dynamic elements have to be modelled. In DMM an enhanced meta model is derived from an existent meta model. Such an enhanced meta model has some new elements which are necessary to describe the behavioural parts of the model. In the UML specification, for example, the textual description states that token flow is the base of the semantics of activities, but tokens are not part of the UML meta model (cp. [OMG07], p. 319). Therefore tokens have to be introduced in the enhanced meta model in DMM to describe the semantics of activity diagrams adequately.

In addition there are operational rules in DMM to define the semantics of the VML. They say which changes of the model’s instances are feasible and when they are allowed to occur. In other words: they can define the semantics of the dynamic part of the language.

Let us now consider that language engineer whose name we assume to be Fred for simplicity wants to define the syntax and the semantics for a VML, for example, for an activity diagram using DMM. To be more precise: he wants to define deterministically how tokens are allowed to flow in an activity diagram; he can create several operational rules saying exactly what is allowed, and when or where a token can flow. If he had to express this in prose text, it would probably be much more ambiguous, less structured, and less intuitive.

Although Dynamic Meta Modelling seems a good solution for defining syntax and semantics of a VML, it has some weaknesses. To make this clear it is necessary to have a closer look at how DMM works. DMM rules have a pre- and a postcondition. But unlike methods in programming languages DMM rules normally do not get called. (There is an exception for so called small-step invocation rules which we will introduce in a later section.) If the model satisfies the precondition of a rule, the rule will be applied. And if the preconditions of more than one rule are satisfied, all these corresponding rules will be applied. Afterwards we hence have several possible states to consider.

Let us say Fred decides to add a new kind of flow final to his VML, an advanced flow final (short AFF) which is similar to the normal kind but has some extra features. When the token arrives at an AFF, it is not consumed as usual, but the whole activity diagram is started anew. The token is thus copied to every initial node. When Fred creates concrete activity diagrams, he sometimes wants to use advanced flow finals and sometimes he wants to use the old flow finals. In the enhanced meta model he uses an inheritance relation where the advanced flow final inherits from the old flow final as its semantics should be an enhancement of the old flow final. The definition of the DMM rule matching mechanism (which will be investigated in more detail later on in this thesis) implies that all the rules related to the original final node are also available for the AFF. To fulfil the preconditions it does not matter whether there is an ordinary flow final or an AFF.

However Fred does not want to have the same functionality for the AFF,
he wants some different postconditions. Therefore he creates some new rules with extended preconditions ensuring that they are only fulfilled when there is an AFF, not when there is a normal flow final. Besides that the preconditions are the same as before.

But inheritance in DMM does not work as inheritance in Java or C where methods can be overwritten. To assume this would be wrong since the new rules do not overwrite the old ones; they are added to the rule set. Figure 1.1 illustrates this. Rules can only be added to the rule set and not overwritten when the VML is extended. That means, if the new rules' preconditions are fulfilled so are the old ones'. Whenever a precondition of the new rules is satisfied, two rules are applied as the old one is satisfied as well. Of course, that is not what Fred wants. He wants the new rules instead of the original rules for the AFF.

Therefore we have to find a possibility to change this. If a node in the enhanced meta model inherits from another node, rules created for the new node should be applied and not the rules they originate from. After all they were created to change the functionality not to gain some more possibilities of what could happen. In this thesis we will have a closer look at the different parts of Dynamic Meta Modelling to solve the problem. We will investigate what kind of extensions we can apply to the enhanced meta model and how we can adjust the concept of inheritance in DMM. As we will see later on our solution resembles "Overwriting" in object-orientation. This is desirable as the target audience of DMM (advanced language users) is familiar with the wide-spread object-oriented concepts of inheritance and overwriting and therefore those concepts would be intuitive to them in DMM as well. That means our solution has the potential to enhance DMM power and at the same time can still be expected to be easily understandable. This is volitional to prevent misunderstandings which would occur with people understanding DMM inheritance in different ways. Figure 1.2 shows how the new extension mechanism should look like. When extending a VML, it should not only be possible to add rules, but to overwrite rules.

The goal of this thesis is to develop a well-defined concept of inheritance and rule application in DMM. We will then discuss the adequateness of our results using several examples. Finally we will formalize this concept. The structure of this thesis is as follows. Chapter 2 gives an introduction to graph transformation and graph transformation rules which are the basis of DMM. Important
terms and definitions in DMM are presented and how graph transformations work within DMM. For illustration we will then give a small example which will serve as a running example of the rest of this thesis. In Chapter 3 we will look at object-orientation in general and we will see in more detail why we should consider it when defining inheritance for DMM. For this we will examine the two object-oriented concepts "inheritance" and "polymorphism". In Chapter 4 we will investigate what kind of extensions we can apply to the enhanced meta model and how to redefine inheritance for DMM. We will show an example to underline our results. In Chapter 5 we will formalise our results of chapter 4 to make them applicable for DMM and then for possible tool support. The last chapter will summarize the results and give an overview of possible future work.
Chapter 2

Dynamic Meta Modelling (DMM)

In this chapter we give the foundation for understanding the Dynamic Meta Modelling (DMM) approach. First we will give a small introduction to DMM in section 2.1 where we will explain why this approach is necessary. Afterwards we will have a closer look at one of the fundamentals of DMM: graph transformations. By means of graph transformations one is able to define the dynamic semantics of a Visual Modelling Language in a precise way. Therefore we will first introduce graph transformations in section 2.2 before we will have a closer look on DMM in section 2.3. Note that all DMM specific definitions are adopted from [Hau05].

2.1 Introduction to Dynamic Meta Modelling

When defining a language, for example a Visual Modelling Language (VML) one first needs a set of symbols one can use and a grammar describing how to actually use them. Those two basics of a language are called the concrete and the abstract syntax. But for other people to understand the language, one also needs a semantics definition to know the meaning of the used constructs.

To define the syntax of a language is not a major difficulty, compared to the definition of the semantics, especially when there are dynamic elements to describe. In natural languages like English or German the semantics definition is given as a textual description, although this is quite imprecise and often leads to misunderstandings. But the same technique is also used to define the semantics of other languages such as the Unified Modeling Language (UML). The Management Group (OMG) used textual description in their UML specification [OMG07] which led to many discussions and misunderstandings about the meaning of different parts of the specification. In addition this kind of definition does not allow for automated model checking as it lacks a formal description.

Jan Hendrik Hausmann develops in [Hau05] an approach which overcomes the insufficiency of present semantics definition techniques for VMLs such as
UML. He specifies certain criteria a VML definition technique has to fulfil. It has to be formal, precise, adequate (means having a semantic gap between language and its semantics as small as possible), analysable (for example with automated analysis techniques), universal (all concepts of the VML must be expressible), and highly understandable. In his thesis he examines different existing approaches if they fulfilled these criteria, but as none does, he combines the most promising approaches into a new one. In his approach he uses the denotational meta modelling approach where the definition of static semantics is derived from a syntax definition with the help of a semantic mapping. But as this approach can express the dynamic semantics only inadequately, he combines it with operational rules or more precisely with graph transformation rules to define the behaviour of the elements. For more information see [Hau05].

In figure 2.1 we can see a graphical overview of the DMM approach where language level and model level are distinguished. On the language level the meta model of the syntax definition is mapped to the meta model of the semantics definition via a semantic mapping. The meta model of the semantics definition often is an enhanced version of the one of the syntax definition. It can have additional elements which describe dynamic elements, like tokens in an activity diagram. In the dynamic semantics definition the behaviour is described by means of graph transformation rules, e.g. when or where a token can flow.

Figure 2.1: Overview of the DMM approach

On the model level we have model elements, which are defined in the syntax
on the language level. As semantic domain we have a Labelled Transition System (LTS) which can be constructed by a start state and graph transformation rules. By applying some of these rules on the start graph, new states can be created. On these states other rules might be able to get applied which again creates new states and so on. The transitions between the states correspond to the application of graph transformation rules defined for the language. Each transition is labelled with the name of the corresponding rule.

2.2 Graph Transformation

Many different kinds of problems and structures can be represented by graphs, not only in the area of software engineering. For example road networks, the link structure of a website, or other complex patterns can be described via graphs. When this is the case, graph transformations can often be applied, for example to specify operational behaviour (cp.[GW96]) just as in DMM.

In section 2.2.1 we will first give a definition of graphs, which we will extend in section 2.2.2 where we will introduce type graphs and typed graphs. Furthermore we will extend our notion of graphs with the concept of inheritance in section 2.2.3. Afterwards we will explain in section 2.2.4 what graph transformation rules are and how they are used.

2.2.1 Graphs

In this section we start with the formal definition of graphs. The most elementary kind of graphs is an undirected graph which consists of a set of nodes \( N \) and a set of edges \( E \) with \( E \subseteq N \times N \), each edge connecting exactly two nodes. Multiple edges between two nodes are disallowed.

To allow multiple edges between two nodes we need a multi-graph which has two additional functions \( s : E \rightarrow N \) and \( t : E \rightarrow N \) to appoint source and target node of each edge. This has the additional effect that edges are unidirectional. In the following we only want to consider directed graphs which are labelled later on to denote the names of the elements.

The next definition is akin to definitions found in [EEPT06, CMR96].

**Definition 2.1. Directed unlabelled Graph**

A directed unlabelled graph \( G_n = (N, E, s, t) \) consists of a finite and non-empty set of nodes \( N \) (sometimes also called vertices), a finite set of edges \( E \), the source function \( s : E \rightarrow N \) defining the source node of an edge, and the target function \( t : E \rightarrow N \) defining the target node of an edge.

In DMM [Hau05] a combined alphabet \( \Lambda \) is proposed to provide labels for both nodes and edges alike. \( \Lambda \) is an arbitrary alphabet which has an extra symbol \( \bot \) which represents an undefined label or name. Compared with labels other than \( \bot \) it yields false, compared with \( \bot \) it produces true. The extended
alphabet $\Lambda^*$ is defined as $\Lambda \cup \bullet$ where $\bullet$ is the wildcard symbol which yields true in comparison with any other label, even $\perp$. That means when using the wildcard symbol, for example to define which labels for an edge or node are allowed, it is not important which label is used or whether a label is defined at all.

We can define the labelling functions as follows:

$$l_N : N \rightarrow \Lambda^*$$
$$l_E : E \rightarrow \Lambda^* \times \Lambda^* \times \Lambda^*$$

The edge name is a triple of labels conform to the UML class diagram notation, where you can choose to give an association name or role names at both ends to distinguish between single associations. This implies that names of different edges should differ in at least one of the three labels. Nodes may carry the same name, for example when both names consist of the symbol $\perp$. In the following, we refer to an edge $e$ between nodes $a$ and $b$ as $e = <a, l_1, l_2, l_3, b>$, where $<l_1, l_2, l_3>$ is the edge name.

Now we can define a labelled multi-graph (cp.[Hau05]):

**Definition 2.2.** Graph
A graph $G = (N, E, l_N)$ consists of
$N$ a finite and non-empty set of nodes,
$E$ a finite set of edges with $E \subseteq N \times \Lambda^* \times \Lambda^* \times \Lambda^* \times \Lambda^* \times N$,
For convenience, we define the functions
$l_N : N \rightarrow \Lambda^*$ the labelling function for nodes,
$l_E : E \rightarrow \Lambda^* \times \Lambda^* \times \Lambda^*$ the labelling function for edges,
s : $E \rightarrow N$ the source function defining the source node of an edge, and
t : $E \rightarrow N$ the target function defining the target node of an edge.

The following definition is from [Hau05]. There are similar definitions in literature, e.g. in [EEPT06] where it is called "labeled graph morphism".

**Definition 2.3.** Edge-label preserving graph morphism (elp morphism)
An edge-label preserving graph morphism $m$ is a structure and edge-label preserving relation between two graphs $G$ and $H$.
$m(G, H) = (m_N)$ with $m_N : N^G \rightarrow N^H$ the node mapping function,
and for all $<a, l_1, l_2, l_3, b> \in E^G : <m_N(a), l_1', l_2', l_3', m_N(b)> \in E^H$ with $l_1 = l_1', l_2 = l_2', l_3 = l_3'$

The upper index indicates the superordinate element. Therefore $E^G$ means graph $G$ is the superordinate element of edge set $E$. Every node in $G$ is mapped to a node in $H$. For each edge in $G$ there is a mapping to an edge in $H$ where starting and ending node of the edge in $G$ are mapped to the ones in $H$. The labels are the same for those edges in $G$ and $H$. 
2.2.2 Typed Graph

There are different possibilities to designate type information on graphs. We will use type graphs where every node represents a type and every node label represents the name of this type. The edges between the nodes represent associations between these types. For the next two definitions also see [Hau05] and [CMR96].

**Definition 2.4. Type Graph**

A type graph \( TG \) is a graph as defined above with the additional requirement that \( l_N \) is injective.

The labelling function needs to be injective so that it is possible to distinguish between the different types. A type graph describes the structure of how the different types can connect. This structure is imposed on typed graphs. The edges in a type graph describe associations between the elements they connect. Only if an association between two types in the type graph exists, a connection between two nodes of those types in typed graphs is allowed (and sometimes required).

**Definition 2.5. Typed Graph**

A typed graph \( G_{\text{typ}} = (G, \text{type}) \) is a graph \( G \) with type: \( G \to TG \) an ep morphism and \( TG \) a type graph.

With the aid of the ep morphism every node in \( G \) is typed by a mapping to a node of \( TG \).

In DMM there are two different kinds of typed graphs. First, we have rule graphs which we will introduce later on in section 2.3.1. They define the operational rules we already mentioned earlier this chapter. Secondly, we have so called host graphs (in DMM called instance graphs) on which the operational or transformation rules are applied to and which we will formally define in section 2.3.1 as well.

To visualise these definitions we will give a small example. In figure 2.2 we can see a type graph with

\[
N = \{\alpha, \beta\}, \\
l_N = \{<\alpha, "A">, <\beta, "B">\}, \text{ and} \\
E = \{<\alpha, \perp, "c">, \perp, \beta >\}.
\]

The graph in figure 2.3 with

\[
N = \{\delta, \gamma\}, \\
l_N = \{<\delta, "a">, <\gamma, "b">\}, \\
E = \{<\delta, \perp, "e">, \perp, \gamma >\}, \text{ and} \\
\text{type} = \{<\delta, \alpha >, <\gamma, \beta >\}
\]
is typed over the first graph by means of the \(\epsilon\)-morphism \textit{type}. As we can see, the visual representation of the type graph is evocative of a UML class diagram. The typed graph on the other hand reminds of a UML object diagram. This notation corresponds to the visual representation of graphs in DMM. Note that, also we defined the graphs as directed, there are no arrows in the graphs, only lines. This is because on one side the labels should clarify how to read the edges, on the other side arrows in a UML class diagram have a semantic meaning different from indicating the direction. To not confuse the user the graphical representation is done without arrows.

![Type Graph](image1)

\textbf{Figure 2.2: Type Graph}

![Typed Graph](image2)

\textbf{Figure 2.3: Typed Graph}

### 2.2.3 Inheritance

There are several graph transformation approaches in literature which work with inheritance. In those concepts so called inheritance edges or hierarchy edges are introduced in the type graph. A reason to introduce inheritance in type graphs is the fact that it is a central concept in object-orientation and therefore it should be supported by graph transformations (cp.[Hau05]) when defining a VML. In [BEL+03, BELT04] they are introduced to obtain node type hierarchies. This has advantages like being able to abstract similar graph transformation rules (which are introduced in the next section) into one. Hence this leads to a more dense form of the graph transformation system.

\textbf{Definition 2.6. Type Graph with Inheritance}

A type graph with inheritance \(G_I = (TG, I, A)\) is a type graph \(TG\) with \(I \subseteq N^{TG} \times N^{TG}\) the set of inheritance edges which must not contain a cycle and \(A \subseteq N^{TG}\) the set of abstract nodes

Nodes in \(A\) are abstract which means that only nodes in rule graphs can be typed over these, but no nodes in DMM instance graphs. Unlike conventional edges, inheritance edges in \(I\) do not have a label. In addition it is not possible
to have multiple inheritance edges between two nodes. A node $n$ inherits from a node $m$ if there is an inheritance edge $< n, m >$. That means $n$ receives all properties of $m$, such as its connections or possible attributes. It is also possible for $m$ to have not only one but a whole subtree of descendants which inherit its properties. This subtree is called inheritance clan. Definition of the inheritance clan can be found in [EEPT05, BEL+03, BELT04, RT05, Hau05].

**Definition 2.7. Inheritance Clan**

For each node $n \in N$, the inheritance clan is defined by

$$\text{clan}_I(n) = \{ n' \in N | \exists \text{ path } n' \xrightarrow{\ell} n \text{ in } I \}.$$  

Note that $n \in \text{clan}_I(n)$.

To be still able to benefit from the graph theory which does not use inheritance, type graphs with inheritance can be flattened. That means in a flattened graph, also called closure, there are no inheritance edges left. Depending on whether working with the abstract closure or the concrete closure, abstract nodes are allowed respectively not allowed.

**Definition 2.8. Abstract Transitive Closure**

Let $TG_I = (TG, I, A)$ be a type graph with Inheritance with $TG = (N, E, l_N)$. The abstract transitive closure of $TG_I$ is the graph

$$TG = (N, E, l_N) with \quad \bar{E} = \{ < n_1, l_1, l_2, l_3, n_2 > | \exists \ e \in E :$$

$$n_1 \in \text{clan}_I(s_E(e)) \land n_2 \in \text{clan}_I(t_E(e)) \land < l_1, l_2, l_3 > = l_E(e) \}$$

**Definition 2.9. Concrete Transitive Closure**

Let $TG_I = (TG, I, A)$ be a type graph with inheritance with $TG = (N, E, l_N)$. The concrete transitive closure of $TG_I$ is the graph $\bar{TG} = (N \setminus A, E, l_N)$ with $E$ as defined above.

Below in figure 2.4 we can see an example of a type graph with inheritance. It has an abstract node B which is associated to a node A and a node C. In addition there are two nodes B1 and B2 which inherit from the abstract node B.

Figure 2.5 shows the Abstract Transitive Closure of the graph in figure 2.4. The abstract node B is still present, but the inheritance edges from node B1 and B2 were removed. Instead we can see the inherited associations to nodes A and C from B1 as well as from B2.

The Concrete Transitive Closure of the graph in figure 2.4 is shown in figure 2.6. Here the abstract node B is not present anymore neither are its associations. The inherited edges from B1 and B2 to A and C respectively are added in the same manner as in the Abstract Transitive Closure.
2.2.4 Graph transformation rules

Graph transformation is used to manipulate the structure of graphs. But usually not a single transformation is needed, but rules for graph transformations. Therefore graph transformation rules (GTRs) are required [Hau05]. Usually there are three parts a GTR consists of: two graphs and an embedding description. The two graphs are the left-hand side of the rule (L) and the right hand side of the rule (R).

A GTR is applied on a host graph G by first finding an occurrence of L in G (in other words finding a matching between L and G) and then replacing it with a copy of R which is connected to the host graph according to the embedding rule to derive a new graph H from it (cp. [EEKR99]).

In DMM the embedding rule is a common application context in L and R, which is not modified when applying the rule [BMST99]. This means that there are identical elements in both graphs related by a (production) morphism. This morphism is indicated by the equality of the labels of the elements which are neither deleted nor changed [Hau05].

Elements which are in L, but not in R are deleted, and elements which are not in L, but in R are newly created.

Later on, we will also give a formal definition of rules and their matchings, when we come to DMM Rules in section 2.3.2.

In figure 2.7 below we can see a rule $r$ in which an object $D$ is newly created. There is a matching $m$ between $L$ and the host graph $G$. In other words: there is an occurrence of $L$ in $G$ (which is emphasised in grey). After the rule $r$
Concepts of Dynamic Meta Modelling

is applied, a new graph $H$ is derived ($d$) from $G$. The application context is existent in matching $m'$ between $R$ and $H$ as well as in the matching $m$.

To apply a rule on a host graph, there must be a matching between the host graph and the rule. Certain structural conditions must be fulfilled. In our case, those conditions are part of $L$. But sometimes we want a specific structure not to be present in the host graph. To achieve this, so called Negative Application Conditions (NACs) are introduced to graph transformation. They check whether a certain structure is absent and a rule with a NAC can only apply when the NAC is not present in the host graph [HHT96]. NACs can be represented as extension of $L$.

In figure 2.9 we can see the graphical representation of a NAC in rule $r^2$. There is only one matching $m3$ for $A$ in the host graph. We can see the difference to the rule $r1$ in figure 2.8 where no NAC exists and where two matchings $m1$ and $m2$ to the same host graph exist.

2.3 Concepts of Dynamic Meta Modelling

2.3.1 Graphs in DMM

Graphs in DMM do not differ much from the graphs we defined in section 2.2.1. They are typed, node and edge labelled multi-graphs, with inheritance in the type graph. In addition they are attributed. We did not explain this concept in the last section as it is of no importance for this thesis. It is sufficient to say that we use a special subset of nodes to represent attributes, so called datatype nodes (DT). Those DT are connected to ordinary (class) nodes via special attribute declaration edges. There is a domain $D_{dt}$ for each data type $dt \in DT$ which contains all possible values of $dt$.

First we will define a DMM Type Graph, which differs from our former definition mainly in additional datatype nodes.
Definition 2.10. DMM Type Graph

A DMM type graph $G_{DMMT} = (G_I, DT)$ consists of

- $G_I = (N, E, l_N, I, A)$ a type graph with inheritance,
- $DT \subset N$ the set of data types,
- $l_N(N) \subseteq \Lambda \backslash \{\perp\}$
- $E \subseteq ((N \backslash DT) \times \Lambda \times \Lambda \times \Lambda \times (N \backslash DT)) \cup ((N \backslash DT) \times \{\perp\} \times \{\perp\} \times \Lambda \times DT)$

The set of all DMM type graphs is referred to as $G_{DMMT}$.

In this definition we can see that there is a distinction between the edges to datatype nodes and to other kinds of nodes. Edges to DT only have one label at the DT side which indicates the name of the attribute. We can also see that node names must not have a wildcard symbol or an undefined label.

Now we go on with the definition of typed graphs, which namely are the DMM instance graph and the DMM rule graph. When we have two or more graphs which are typed over a type graph, we assume that it is the same DMM type graph.
Definition 2.11. DMM Instance Graph
A DMM instance graph $G_{DMI}=(G, \text{type})$ consists of
- type: $G \rightarrow \overline{TG}$ the typing elp morphism
- $TG \in G_{DMM}$
- $l_N(N) \subseteq \Lambda$
- $l_E(E) \subseteq \Lambda \times \Lambda \times \Lambda$
- $\forall dt \in DT^{TG} : \text{type}^{-1}_N(dt) = D_{dt} \subseteq N^G$

The set of all DMM instance graphs is referred to as $G_{DMI}$.

A DMM instance graph is typed over the concrete closure $\overline{TG}$ of a type graph $TG$. We can see that the nodes in a DMM instance graph as well as in DMM rule graphs (in definition 2.12 below) may contain an undefined label in contrast to nodes in type graphs.

Definition 2.12. DMM Rule Graph
A DMM rule graph $G_{DMMR}=(G, \text{type})$ consists of
- type: $G \rightarrow \overline{TG}$ the typing elp morphism
- $TG \in G_{DMM}$
- $l_N(N) \subseteq \Lambda^*$
- $l_E(E) \subseteq \Lambda^* \times \Lambda^* \times \Lambda^*$
- $\forall dt \in DT^{TG} : \text{type}^{-1}_N(dt) = D_{dt} \subseteq N^G$

The set of all DMM rule graphs is referred to as $G_{DMMR}$.

A DMM rule graph differs from a DMM instance graph in the labelling alphabet $\Lambda^*$ as in the rule graph the extended alphabet is used. Furthermore the DMM rule graph is typed over the abstract closure $\overline{TG}$ of the type graph $TG$ which means that abstract nodes may occur. This and the possibility to use wildcard symbols makes the formulation of rules much more flexible.

2.3.2 DMM Rules
In chapter 2.2.4 we already gave an informal description of GTRs. Now we define them formally for DMM.

Definition 2.13. Graph Transformation Rule (GTR)
A Graph Transformation rule $r : (L,R)$ consists of two DMM rule graphs $L$ and $R$ and a name $r$. Note that for all following definitions, we assume that the different graphs are defined over a common base set such that $L \cap R, L \cup R$ are defined.

A graph transformation rule $r$ can be divided into three different partitions:
- elements to be deleted, elements to be newly created, and application context.
  - $r_{del} = \{N_{del}, E_{del}\}$ with $N_{del} = N_L \setminus N_R$ and $E_{del} = E_L \setminus E_R$
  - $r_{new} = \{N_{new}, E_{new}\}$ with $N_{new} = N_R \setminus N_L$ and $E_{new} = E_R \setminus E_L$
  - $r_{ac} = \{N_{ac}, E_{ac}\}$ with $N_{ac} = N_L \cap N_R$ and $E_{ac} = E_L \cap E_R$

Note that DMM follows the so called Single-Pushout-Approac h (SPO), not the Double-Pushout-Approac h (DPO) which both differ from each other.
mainly in the way dangling edges are handled. While in DPO rules are not allowed, which result in dangling edges (e.g. after deleting only one end node) such rules may occur in SPO. This is because in SPO dangling edges are automatically deleted. But theoretic results for each approach may be transferred (with restrictions) to the other one as it is shown in [EHK+97].

The matching of a rule under SPO is defined as follows.

**Definition 2.14.** Matching of a rule
For a rule $r=(L, R)$ a matching is an injective morphism $mt$ with
\[ mt : L \rightarrow G, \quad L \in G_{DMMR}, \quad G \in G_{DMMI} \text{ and} \]
\[ \forall <ln, gn> \in mt : type_N(gn) \in clan_I(type_N(ln)) \]

This means that the type of two matching nodes in $G$ and $L$ do not necessarily need to be the same. It is also possible to have a matching between $ln$ and $gn$, if in the type graph the type of a node $gn$ from $G$ inherits (directly or with intermediate nodes) from the type of the node $ln$ which occurs in $L$.

In section 2.2.4 we mentioned the concept of NACs. Now we will deal with them again to define them properly.

**Definition 2.15.** Negative Application Condition (NAC)
A NAC of a rule $r=(L, R)$ is $NAC(r) = \{\hat{L}\}$ with
\[ \hat{L} \in G_{DMMR} \]
\[ \forall \hat{L} : \hat{L} \in L \]

Each NAC is an extension of the left hand side of the underlying rule.

A matching $mt$ for rule $r$ satisfies its conditions, if for no $\hat{L} \in NAC(r)$ an extended matching $\hat{mt} : \hat{L} \rightarrow G$ can be found which is identical to $mt$ for elements of the underlying rule’s left hand side.

There is another concept which is supported by DMM, but which we just want to adumbrate as it is of no great importance for this thesis. Normally, elements of the left hand side of a rule are existential quantified, which means that they need to find exactly one matching in the host graph for the rule to be applied. However, sometimes such an element should have a matching with more than one element. This is realised with so called Universal Quantified Structures (UQS) in the left hand side of the rule. The UQS will match to the maximal number of elements in the instance graph which fulfil the given conditions.

The use of UQS in DMM is limited though. A UQS is formed by a special kind of node and all its adjacent edges. There can be several UQS per rule and when nodes marked as UQS are directly connected by an edge, they form a single UQS.

Rules with UQS can get unfolded to a good deal of simple rules (potentially an unlimited amount, which needs to be restricted to expected numbers). Universally quantified elements are either preserved or deleted, but they are not contained in $r_{new}$. The multinode of UML Collaboration Diagrams serves
as graphical representation of a UQS. For more information about UQS see [Hau05].

**Definition 2.16.** Deleted Edges

Let $Gr_{DMMI}=(G, \text{type})$ be an instance graph and $r$ a rule, such that $r$ matches $Gr_{DMMI}$ with a matching $mt$. The set of deleted edges $de(mt(r))$ is defined as

$$de(mt(r)) = \{ e \in E^G | \exists n \in N^r_{\text{del}} : s_E(e) = mt_N(n) \lor t_E(e) = mt_N(n) \}$$

As we can see there is no differentiation in the definition between explicitly and implicitly deleted edges. As soon as a node $n$ is deleted all adherent edges are deleted as well, whether $n$ was starting node or end node of the considered edge.

Now we can give a formal definition of the derivation from a graph $G$ to a graph $H$ when applying a rule as well after introducing the concept in section 2.2.4.

**Definition 2.17.** Graph Derivation

Applying a rule $r$ to a host graph $G$ yields a graph derivation:

$$d_r : G \to H$$

with $G, H \in G_{DMMI}$, $mt$ is the matching between $r$ and $G$ so that $mt'$ a matching between $r$ and $H$ exists with

$$mt(r_{ac}) = mt'(r_{ac}),$$

$$N^G \setminus (mt(N_{\text{del}})) = N^H \setminus (mt'(N_{\text{new}})),$$

and

$$E^G \setminus (mt(E_{\text{del}}) \cup de(mt(r))) = E^H \setminus (mt'(E_{\text{new}}))$$

This basically means that nodes and edges which are deleted from $G$ by rule $r$ are not present in the new graph $H$. Instead the nodes and edges newly created by $r$ are part of $H$. In other respects the both graphs $G$ and $H$ do not differ.

Now we should concern ourselves again with the graphical representation of rule graphs. We already gave some small examples on how they might look like, but what we did not mention yet is that we can merge the left hand side and the right hand side of a rule into one graph. Elements of $r_{\text{del}}$ then get the tag \{destroyed\}, elements of $r_{\text{new}}$ get the tag \{new\}. Elements of $r_{ac}$ do not get tagged.

In figure 2.11 we can see an example of the merging of the two sides of the rule graph of figure 2.10.

**Definition 2.18.** Graph Transformation System

A Graph Transformation System is formed by a set of Graph Transformation rules $GTR$s and a start graph $G_0$. All rules and the start graph must be typed over a common type graph.

From the start graph $G_0$ of a Graph Transformation System a sequence of other graphs can be derived by applying the rules of the rule set in the Graph
Transformation System repeatedly. An extension of such a Graph Transformation System hence is a set of derivation sequences gained by the application of rules on the start graph $G_0$.

**Definition 2.19.** Derivation Sequence

A derivation sequence $G \Rightarrow H$ is a sequence of derivations $G \xrightarrow{r_1} G_1 \xrightarrow{r_2} G_2 \ldots \xrightarrow{r_{n-1}} G_n \Rightarrow H$ with $r_i$ being rule applications, i.e., matchings of rules. A Graph Transformation System thus defines a set of such derivations with all applied rules originating from GTRs.

The choice which rule to apply next in the derivation sequence is non-deterministic, just as the choice which matching of a single rule to use (if there are several possible matchings). Therefore complex graph manipulations need to be packed into one single rule. Otherwise one could not ensure that the intended manipulations are executed completely and in the right order. This leads to complex rules which are hard to create and large rule sets as every possible scenario requires an own rule.

However, this is not very practical as it is easier to control smaller parts, i.e. less complex rules. Hausmann proposes in [Hau05] some mechanisms to control such complex manipulations which split the manipulations into smaller parts. This has the additional advantage that some parts can be reused for different manipulations thus not only making the rules smaller, but the rule set as well. We will introduce these mechanisms in the next sections.

### 2.3.3 DMM Rule Invocation

One of the main control mechanisms are so called invocations of rules. The idea is simple: a rule can invoke one or more other rules. That affects the derivation sequence as the invoked rule must be applied directly after the invoking rule.

The visual representation is evocative of UML Communication Diagrams (cp.[OMG07] p.512), as it is an arrow pointing at a so called target node or context node. The context node is a parameter for invocations of a rule. In an object-oriented way one could say that the context node or more precisely the type of the context node owns the behaviour expressed by the rule. The arrow is labelled with a sequence number (seq) when there is more than one invocation to assign the order and with the name of the rule to be invoked.
(name) including additional possible parameters (params) in parenthesis as we can see in the definition below.

With rule invocation it is also possible to model recursive loops, as we will show in a later example.

**Definition 2.20. Rule with Invocation**

A rule with Invocations \( r_{\text{invoc}} \) consists of

where \( r : L \rightarrow R \) a rule and

\( \text{inv} = <\text{seq}, \text{name}, \text{params}> \) a relation with \( \text{seq} \in N, \text{name} \in \Lambda \setminus \{\bot\}, \emptyset \neq \text{params} \subseteq N^R \)

Note that 0 as sequence number indicates an arbitrary sequence order. The rule name is part of the signature to which we will come in the next section. The parameters represent the information passed to the invoked rule. Hausmann defines the parameters as a subset of the set of nodes in \( R \) and says that "the first parameter always represents the target node of the invocation" (cp. [Hau05], p.87) whereas "target node" refers to what we mostly call "context node". As there is always a context node, the set of parameters cannot be empty. The problem with this definition is that a set is unordered and therefore it is not possible to determine the first parameter (unless there is only one). The parameters can therefore not be in a set, but must be in a tuple to get an order. In the following we want to use the notation \( \text{params}[i] = x \) if \( x \) is the \( i \)-th parameter in \( \text{params} \). Referring to \( \text{params}[1] \) means therefore referring to the context node.

Now we will give some additional details for a better understanding of rule invocation.

First, note that invoked rules are only applied after all graph manipulations of the invoking rule are accomplished.

Second, "deep" invocations have precedence. That means when a rule first invokes a rule \( A \), then a rule \( B \), and rule \( A \) invokes another rule \( C \), this sequence of \( A \) and \( C \) is applied before rule \( B \) is applied to the host graph.

Third, an invocation must be fulfilled by some rule, that means there must be a rule which can be applied. Otherwise the invocation is said to be failed as is the invoking rule. Hence whole invocation hierarchies can fail, even though the first actually applied successfully.

Now we have to consider the question, if every rule can be invoked. DMM distinguishes between so called **big-step rules** and **small-step rules**.

Big-step rules have the normal notion of graph transformation rules as we introduced in the last section. They can be applied to the host graph without invocation, but they might invoke other rules, which must be applied as well (or fail) before applying another big-step rule. If one of the invocations in the hierarchy fails, the original big-step rule fails as well. Therefore it is not considered successful before all invocations have been fulfilled. As the big-step rules have a transaction character, only one big-step rule may be active at a time. That means there must not be two big-step rules (including possible
invocations) which are applied to the host graph at the same time. Big-step rules are marked with a "***" at the end of their signature. Small-step rules on the other hand can only be applied when another rule invokes them, and can themselves invoke other rules as well, which may lead to an invocation hierarchy.

Invocations as we introduced them are carried out after the manipulations of the invoking rule are done. Therefore we can presume that they are an extension of the right hand side of a rule as they extend the invoking rule's effect on the host graph. To get a similar decomposition on the left hand side of the rule, so called premise rules are introduced, which are marked by a prefix "P_" in their names to be able to distinguish them from other rules. Premise rules can be used to avoid recurrences in the left hand sides of different rules. Using them has similar advantages as small-step rules, as they allow for less complex structures, smaller rule sets, reusing recurrent structures, and easier maintenance in an object-oriented sense. As they are an extension of the left hand side of the invoking rule, their right hand side must be identical to their left hand side. They cannot invoke small-step rules, but they may invoke other premise rules. They may contain NACs, but no UQSs. Premise rules are particularly useful, when a number of rules rely on a certain fact. This fact can then be modelled as a premise rule.

For more information about rule invocation see [Hau05].

2.3.4 DMM Rule Signature

When a rule contains an invocation, information is needed about which small-step rules can actually be invoked by this invocation. This information is called rule signature. A rule signature consists of four parts: the context node name, the rule name, parameters, and the big-step indicator.

In the last section we already explained that the context node is a parameter for possible invocations. The name of the rule does not need to be unique. Sometimes it is volitional to have several rules with the same name, for example to express the same behaviour under different conditions.

The notation for parameters consists of name, colon, and type, and they are separated by commas. They are parenthesised and "()" indicates that there are no other parameters besides the context node. Each parameter must be a node in the left hand side of the rule. When the rule is invoked, there must already be fixed matches in the host graph.

The big-step rule indicator indicates if a rule is a big-step rule or not. As already mentioned a "***" indicates that the rule indeed is a big-step rule. When this symbol is missing the rule is a small-step rule.

The graphic in figure 2.12 is from [Hau05] page 89. It illustrates how the signature and the invocation are displayed graphically.
With the definition of invocations and signatures of rules we now have the means to define a matching between an invocation and a rule.

**Definition 2.21. Invocation-Rule matching**

An invocation $i$ and a rule $r$ are said to match under the following conditions:

- $\text{matches}(i) \to r$ with $\text{matches}$ a bijective function and
- $\forall <p, p'> \in \text{matches} : \text{name}_p = \text{name}_{p'},$
  $\text{type}(p) = \text{clan}_I(\text{type}(p'))$, and
  $\text{name}_i = \text{name}_r$.

As you can see in this definition, invocations can have a matching to rules, which work on subtypes of their defined parameters.

### 2.3.5 DMM Final Rules

With the concepts of rule invocation, premise rules, NACs, and UQS, our definition of DMM rules does not suffice anymore. We need a new definition which combines all these concepts.

**Definition 2.22. DMM Rule with Invocations (final rule)**

A DMM Rule with Invocation $r$ is a six-tuple with $r \in r_{DMMInv}$ and

$r_{DMMInv} = \{<L, R, NAC, inv, Params, name>\}$, with

$L, R \in G_{DMMR}$ ordinary rule graphs,

$NAC = \{L\} \subseteq G_{DMMR}$ a set of negative application conditions,

$inv$ an invocation relation as described in definition 2.20,

$Params \subseteq L, \forall p \in (Params \cup \{x \mid x \in params^{inv}\}) : l_N(p) \neq \bullet$, name $\in \Lambda \setminus \bullet$.

As you can see all nodes which are used as parameters must not have the wildcard as label. This constraint is necessary to allow the parameters to be
represented textually in the head of the rule or together with the invocation arrow. The name of the rule must not be empty, either.

This definition of a rule is extended with information about the rule itself, or more precisely with its name and parameters, about its NACs, and the invocations which are specified in the rule. But it neither contains information about possible UQS, nor about possible premise rules. They can not be handled in a normal rule as they need preprocessing steps. Therefore Hausmann introduced the term of "Rule Schema" which has a so called core rule, which is a rule as defined in definition 2.22, and has an additional invocation relation for premise rules, and a set of universally quantified structures UQS which are a subset of \( r_{DMMInv} \). For a formal definition see [Hau05], pages 95 et seq.

A DMM Rule schema can be unfolded to final DMM Rules. First, the UQS need to be unfolded to simple nodes, edges and NACs. Second, the premise rules need to be glued with the core rules. As we do not need the formal definitions in this thesis, we settle with this informal description. For a more precise presentation see [Hau05].

2.4 DMM Example

Now we will give a small toy example. The type graph in figure 2.13 vaguely reminds of a very easy version of an activity diagram. We have Nodes which are connected via Edges. Each Edge has a start node and an end node, and for each Edge and Node it is distinguished whether it is an incoming edge or an outgoing edge. One Node can have an arbitrary number of Tokens which are stored in a queue.

![Type Graph](image)

Figure 2.13: Type Graph

In figure 2.14 we can see a rule which is typed over the graph in figure 2.13. This rule defines token flow from one node to another. Note that only the first Token in the queue can flow to another Node. This rule invokes two other rules: node.enqueue and node.dequeue, but besides that it does not change the graph. There are two rules with the same signature for each of the invoked rules which can be seen in figures 2.15, 2.16, 2.17, and 2.18 respectively.

node.enqueue appends a Token which was passed as parameter to this rule,
to *node* of the type *Node*. The two rules with the same signature are necessary to distinguish two cases. In the first case there is no *Token* in the queue yet and the new *Token* is the first element. In the second case there is at least one *Token* in the queue and the new one is appended to the end.

*node.dequeue* removes the first *Token* from *node* of the type *Node*. The first of the two *node.dequeue* rules is applied when there is at least one other *Token* in the queue left. The other one is applied when the removed *Token* was the only one in the queue.

In figure 2.19 we can see a possible instance graph. With our rule set and this graph as start graph, we get a derivation sequence which we can see in the following graphs. From state $S_0$ we get to state $S_1$ when applying the first rule invocation *n2.enqueue(t2:Token)* of rule *n1.flow()**. From there we get to state $S_2$ after applying the second rule invocation *n1.dequeue(t2:Token)* of rule *n1.flow()**. The *Token* $t2$ moved from *Node* $n1$ to *Node* $n2$. *Token* $t1$ is then enqueued to *Node* $n3$ (state $S_3$) and afterwards dequeued from *Node* $n2$ (state $S_4$). Finally, the same happens for *Token* $t2$. After enqueuing it to $n3$ we receive state $S_5$ and we reach our final state $S_6$ after dequeuing it from $n2$. 

![Figure 2.14: Rule node.flow()](image)

![Figure 2.15: Rule node.enqueue when there is at least one other token in the queue](image)

![Figure 2.16: Rule node.enqueue when there is no other token in the queue](image)
DMM Example

Figure 2.17: Rule node.dequeue when there is at least one other token in the queue

Figure 2.18: Rule node.dequeue when there is no other token in the queue

Figure 2.19: State $S_0$: Start Graph

Figure 2.20: State $S_1$: Graph after first rule invocation n2.enqueue(t2:Token) of the rule n1.flow(*)

Figure 2.21: State $S_2$: Graph after second rule invocation n1.dequeue(t2:Token) of the rule n1.flow(*)
Figure 2.22: State $S_3$: Graph after first rule invocation $n3.enqueue(t1:Token)$ of the rule $n2.flow()$ (First Token)

Figure 2.23: State $S_4$: Graph after second rule invocation $n2.dequeue(t1:Token)$ of the rule $n2.flow()$ (First Token)

Figure 2.24: State $S_5$: Graph after first rule invocation $n3.enqueue(t2:Token)$ of the rule $n2.flow()$ (Second Token)
Figure 2.25: State $S_6$: Graph after second rule invocation $n2.dequeue(t2:Token)$ of the rule $n2.flow()` (First Token)

In figure 2.26 we can see the transition system, with the start graph from figure 2.19. There is a fork in the transition system right at the beginning as both Tokens can flow from the Node where they are in the queue to the next Node. The derivation sequence in the upper half is the derivation sequence which we just visualized. The derivation sequence in the lower half occurs, when the other Token moves first. Then Token $t1$ moves from node $n2$ to Node $n3$ and only then does Token $t2$ move. It flows first from $n1$ to $n2$ and afterwards to $n3$.

It is noticeable that the last two states in both derivation sequences are the same, which does not need to be the case in general.

Figure 2.26: Transition System
Chapter 3

Inheritance in Object-Orientation

Nowadays object-orientation is an established method to handle complexity of software systems. It focuses on modular systems which help control complexity. It provides many tools and instruments for problems of various kinds. Techniques of object-oriented programming support the design of software which is upgradeable, testable, and maintainable (cp.[LaRa06]) and object-oriented programming languages are widespread in commerce and education. When modelling certain facts in the software development process, languages like UML are used, which too, support the concept of object-orientation.

Therefore it is only natural that a Meta modelling concept like DMM for such a modelling language should support object-orientation as well. Unfortunately, DMM only supports it insufficiently as we will see later. This may be unexpected for users and may therefore lead to unwanted side-effects. To clarify what we mean with "object-orientation" we will first give a small overview in section 3.1. Afterwards we will investigate the concept of inheritance further, which seems to be the most important concept in object-orientation. Then we will have a small look at the concept of polymorphism, which is linked with inheritance in a way.

3.1 Object-Orientation

In literature there are many different opinions what object-orientation is or what its most important concepts are. As Armstrong points out in [Ar06] the majority of authors does not give a proper definition, but enumerates the most important concepts.

Heide Balzert, for example, describes in [Bal99] six basic concepts: class, object, method, message, inheritance, and attribute, whereas Lahres and Rayman [LaRa06] think that there are only three concepts which are the least common denominator in object-oriented programming languages: inheritance, encapsulation, and polymorphism.
Cardelli even suggests in [Car88], p.2 "that inheritance is the only notion critically associated with object-oriented programming".

Armstrong investigated in [Ar06] 88 sources in which the authors claimed that a specific set of concepts characterize object-orientation. Altogether she could make out 39 concepts of which eight were identified by the majority of authors. These eight concepts are inheritance, object, class, encapsulation, method, message passing, polymorphism, and abstraction. Table 3.1 is an extract of the complete table in [Ar06]. In the latter some less enumerated concepts are displayed as well. As we can see in this table inheritance seems the most important concept, and polymorphism, too, is among the most important ones.

<table>
<thead>
<tr>
<th>Concept</th>
<th>Count</th>
<th>Percentage</th>
</tr>
</thead>
<tbody>
<tr>
<td>Inheritance</td>
<td>71</td>
<td>81 %</td>
</tr>
<tr>
<td>Object</td>
<td>69</td>
<td>78 %</td>
</tr>
<tr>
<td>Class</td>
<td>62</td>
<td>71 %</td>
</tr>
<tr>
<td>Encapsulation</td>
<td>55</td>
<td>63 %</td>
</tr>
<tr>
<td>Method</td>
<td>50</td>
<td>57 %</td>
</tr>
<tr>
<td>Message Passing</td>
<td>49</td>
<td>56 %</td>
</tr>
<tr>
<td>Polymorphism</td>
<td>47</td>
<td>53 %</td>
</tr>
<tr>
<td>Abstraction</td>
<td>45</td>
<td>51 %</td>
</tr>
<tr>
<td>Instantiation</td>
<td>31</td>
<td>35 %</td>
</tr>
<tr>
<td>Attribute</td>
<td>29</td>
<td>33 %</td>
</tr>
<tr>
<td>Information Hiding</td>
<td>28</td>
<td>32 %</td>
</tr>
</tbody>
</table>

Table 3.1: Count of mentioned concepts in object-orientation

3.2 Inheritance

In this section we will first introduce class inheritance and then we will explain subtyping. Afterwards we will point out the difference between subtyping and class inheritance.

3.2.1 Class Inheritance

An inheritance relation between two classes represents an is-a relationship between those two classes. The class which inherits is the subclass, the class from which is inherited is the superclass. An is-a relation means that an element of the subclass is also an element of the superclass (cp.[LaRa06], [Bre01], p.198 et seq.).

There are different reasons to use inheritance in object-orientation. It can be an instrument to structure given objects and classes in hierarchies (cp. [Car88]). When there is a number of classes which resemble each other in certain ways, e.g. have the same attributes, one can create superordinate
classes, also called superclasses. For example the classes cat, dog, and ferret can have the superclass predator or mammal, depending on which attributes are important. This relationship between the subordinate element (or subclass) and superordinate element is called generalisation. One could also start with the superordinate element and then refine it. The new element can have extensions, i.e. additional attributes or behaviour. This would be the inverted relationship to a generalisation, a specialisation.

Another reason to use inheritance is code reuse in object-oriented programming.

But it is not sufficient to have the same attributes to use an inheritance relationship. To use inheritance there should be an is-a relation between the superordinate and the subordinate element. A small example suffices to make this clear. Assuming that we have a sweater, we can model its colour. For a cat, a blossom, or a car we can model an attribute colour as well, but it would probably be rather useless to create a superordinate element for these four objects, just because of the attribute colour. There would probably be no common context where to use these classes. Therefore it would be bad style to use inheritance here.

Brey gives some further guidelines in [Bre01], p.198 et seq. when to use inheritance: the superclass should have a technical correspondent, the generalisation hierarchy should prepare a possible extension of the system, there is more than one subclass, classes in the generalisation hierarchy should have similar behaviour and are used in a common context.

Inheritance can be used in different ways. For example, single inheritance only allows one superordinate element, whereas multiple inheritance allows for more than one (cp. [Car88]). Examples for these two kinds of inheritance are shown in figure 3.1 and figure 3.2. Cat, dog, and ferret each inherit from one superclass. In contrast the amphibian vehicle inherits from two superclasses, from watercraft as well as from ground vehicle.

![Figure 3.1: Single Inheritance](image1)

![Figure 3.2: Multiple Inheritance](image2)

We can also distinguish between interface inheritance and implementation inheritance. When using implementation inheritance the subclass inherits the already implemented functionality of the superclass. Besides the obligations it therefore gets all methods, all data, and all abilities of the superclass (when
Inheritance

there are no other restrictions). The inherited functionality can be adopted unchanged or partly be overwritten (cp. [LaRa06]).

For interface inheritance it is important to know that there is a specification for every class, which consists of a precondition and a postcondition for every operation and additionally of an invariant for the class. Interface inheritance is based upon the fact that a subclass inherits the specification of its superclass. The subclass takes on all the obligations and covenants of the superclass, or in other words it inherits the interface of the superclass. This is a logical consequence of the idea that every element of a subclass should also be an element of the superclass (cp. [LaRa06]).

In this context the Liskov substitution principle is frequently mentioned. The Liskov substitution principle says that all properties of a superclass should be true for the subclass as well (cp. [LiWi94]), which includes the whole specification of the superclass.

This principle does not forbid changes in the subclass, although there are some constraints. First, a precondition for an operation which is defined in the superclass can be preserved or softened, but not tightened in the subclass. Second, a postcondition for an operation which is defined in the superclass can be preserved or extended, but not narrowed. And third, the invariants of the superclass must be valid in the subclass as well (cp. [LaRa06, Bre01]).

In programming languages the differentiation between the concept of implementation inheritance and interface inheritance is not uniform. In most languages it is not possible to just inherit the specification since the implementation is inherited as well. Mostly the only possibility to only inherit the specification are interface classes which do not have an implementation. In C++ it is also possible to inherit only the implementation by using private inheritance (cp. [LaRa06]).

In DMM either single or multiple inheritance can be used in type graphs between nodes which represent types. Currently, the subtype inherits the specification from its supertype. The rules which resemble methods in object-oriented programming languages, do not get overwritten, there can only be additional behaviour. Therefore the subtype can be used when the supertype is required. When we model a VML we cannot speak of implementation in the programming language sense. Nevertheless, when we use inheritance in a type graph in DMM the subtype does not only inherit the specification, but everything that was modelled for the supertype.

3.2.2 Subtyping

Programming languages like Java use data types. Khoshafian and Abnous define in [KhAb95] data types as a description of a set of objects with the same structure and the same behaviour. A data type with hidden implementation of user-defined operations is called abstract data type. Java has some built-in
data types like integers and characters, but it is also possible to define own data types.

A type $S$ is a subtype of a type $T$ when all instances of $S$ are also instances of $T$. For example prime numbers are a subtype of integers as every prime number is also an integer and can be used whenever an integer is needed (which is conform to the substitution principle mentioned above).

Cardelli uses the notion $S \leq T$ for $S$ is a subtype of $T$ in [Car88]. Subtyping is reflexive which means that every type is always a subtype of itself. It is also transitive, which means that if $S_2 \leq S_1$ and $S_1 \leq T$ then $S_2 \leq T$.

There are three possibilities to receive a subtyping relationship: 1. subsets (the set of prime numbers is a subset of the set of integers), 2. subtyping of structured types such as tuples: \[ \text{Name: String, Address: String, Age: Integer} \leq \text{Name: String, Address: String} \], and 3. subtyping of functions. Especially the last point is interesting for DMM when we consider that functions are methods and DMM rules behave similar to methods. We might consider the possibility that rules can only be overwritten by subtypes of them.

Cardelli notates a function $f$ with parameter $T_p$ and return value $T_r$ as follows: $f: T_p \rightarrow T_r$. He defines the subtype relation for functional types like this:

$T_p \leq S_p$ and $S_r \leq T_r$ then $S_p \rightarrow S_r \leq T_p \rightarrow T_r$.

Castagna adopts this notion and explains in [Cas95] that this function is covariant on the right argument as it preserves the direction $\leq$, and contravariant on the left argument as it reverses the direction of $\leq$. The conduct of the left hand side is taken characteristic for the function and therefore this rule is called contravariance rule. The covariance rule looks like this:

$S_p \leq T_p$ and $S_r \leq T_r$ then $S_p \rightarrow S_r \leq T_p \rightarrow T_r$.

He claims that the covariant and contravariant rule concern methods that are functions, that is methods that have one or more parameters and exactly one return value (which might be a tuple).

To increase the understandability of the contravariance rule, we visualise the subtyping of the function $T_p \rightarrow T_r$ and $S_p \rightarrow S_r$ in figure 3.3 (cp.[KhAb95], p.111). When a function is defined on a set, it is also defined on a subset. A function with a domain (here $S_p$) is therefore also a function on a subdomain (here $T_p$), and if the function’s range is $T_r$, it is also defined on range $S_r$ when $S_r$ is a subset of $T_r$.

Khoshaan and Abnous assert that for each method in a strongly typed language the parameter types and the type of the return value are specified and for strong typing the contravariance rule is needed to avoid run-time errors.

The notion of covariance and contravariance can be applied on pre- and postcondition of a method, if they reference only the method’s formal parameters (cp. [KhAb95], p.112). Then the parameters in the pre- and postcondition of the overwriting method can preserve or reverse the subtyping relation to the parameters in the overwritten pre- and postconditions. When the types in both
Inheritance

Figure 3.3: Subtyping of functions with the contravariance rule

pre- and postcondition preserve the subtyping relation, the covariance rule is applied. When the precondition reverses the relation, and the postcondition preserves it, we talk of the contravariance rule.

According to Khoshaian and Abnous the two concepts of subtyping and inheritance are often mixed up or used interchangeably. (They themselves often use the terms subtype and subclass synonymously, authors like Lahres and Rayman do not dwell on subtyping at all; they only speak of interface inheritance). As typing is defined to be the behaviour and behaviour is independent of the implementation it is possible to speak of a behavioural hierarchy when speaking of subtyping, whereas class inheritance is more an implementation hierarchy.

The both concepts differ among other things in the way they can be created. Class inheritance is always explicitly designated. Subtyping can be explicitly designated as well (to suggest that the subtype inherits from its supertype), but it can also be established implicitly in the manner described above (subsets, subtypes of structured types, subtypes of functions). There is also the possibility of assorting these both possibilities by sometimes declaring it explicitly and sometimes deducing it.

In figure 3.4 we see a figure from [KhAb95] p.94, showing the resemblance between abstract data types and subtypes, and classes and class inheritance. In programming languages classes are used to implement abstract data types and inheritance is used to express and support the subtype relationship between classes.

We could argue that inheritance in DMM is more explicit subtyping than class inheritance when we think of implementation in the way programming languages use the term. On the other hand the types in DMM resemble classes in Java as they can have attributes and functionality. The nodes in the typed graphs are similar to objects. Therefore it is possible to say that the inheritance concept in DMM is really more like class inheritance.
3.3 Polymorphism

The ability that different objects can behave differently when the same method is applied to them is called polymorphism. As already said, polymorphism is an important concept in object-oriented programming. One can distinguish between dynamic polymorphism and static polymorphism.

A precondition for dynamic polymorphism is the so called late (or dynamic) binding. Late binding identifies the right operation for a given operation call during run-time. Contrary to this is the so called early (or static) binding, where all operation calls are assigned to operations at compile-time.

To understand the coherences, it is of avail to know that for some programming languages it is not always clear at compile-time which type an object has when using inheritance. It can be an object of the denoted class or of a subclass. Therefore when a method is overwritten, that is when it is defined in the superclass and redefined in the subclass, it is not clear which method to use for the given object. With the help of late binding, dynamic polymorphism allows identifying types of deduced objects during run-time and then choosing the methods for these objects accordingly. Hence dynamic polymorphism is also connected with the overwriting of operations (cp. [LaRa06], [Oes04], p.67 et sqq.)). Some authors like John Hunt in [Hun98] even say that overwriting is a type of polymorphism just as static polymorphism.

Below we can see an example of dynamic polymorphism in Java in figure 3.5 and an associated UML class diagram in figure 3.6. The output of the main method would be "Class B" although the method whoAreYou is called with an object of type A. This is because of the concept of late binding generally used in Java, which allows to identify the given object during run-time. As we create an object of class B, the overwritten method of class B is chosen and not the one belonging to class A.
public class A {
    public void whoAmI() {
        System.out.println("Class A");
    }
}

public class B extends A {
    public void whoAmI() {
        System.out.println("Class B");
    }
}

public class Test {
    public static void main(String[] args) {
        whoAreYou(new B());
    }
    public static void whoAreYou(A a) {
        a.whoAmI();
    }
}

In [KhAb95] Khoshaan and Abnous categorise the possibilities of overwriting methods. They distinguish three possibilities.

- No redefinitions are allowed. That means a subclass cannot overwrite an inherited method. This alternative is quite inflexible. In Java one can enforce this possibility with the key word final in the method of the superclass. It is also possible to mark a whole class as final. Then it is not possible to inherit from this class at all.

- Arbitrary redefinitions are allowed. That means a subclass can overwrite the behaviour of the superclass completely, there is no need of any resemblance to the method of the superclass, and no restrictions on argument types. The only thing that must be in common is the method name. When the overwritten method is called, the most specialised method possible is executed.

- Only constrained redefinitions are allowed. Khoshaan and Abnous suggest that types and behaviour of the subclass must have signatures which are subtypes of the methods' signatures in their superclass. That means the contravariance rule is applied. The parameters of the overwritten method must be subtypes of the parameters of the overwriting method and the return values of the overwriting method must be a subtype of the return values in the overwritten method. Pre- and postconditions must conform to the one they overwrite.

Constrained overwriting guarantees strong typing as used in Java and C♯. Strong typing means that all expressions are type-compatible (of the
Polymorphism

defined type or one of its subtypes) at run-time. Constrained overwriting is necessary, when the Liskov substitution principle should be met. Otherwise one cannot use an object of the subclass whenever one of the superclass is required as it might behave completely differently without those constraints.

Constrained overwriting is also necessary to allow dynamic binding to subclass instances. Otherwise redefined method would not correspond to the superclass method and the decision which method is executed needed to be fixed at compile-time.

Static polymorphism is also known as overloading of methods or operations. A method can have several implementations with different parameters, different implementations can be invoked on objects of different types. It allows operations with the same name, but different semantics. The right one is identified by the types of the parameters passed to that method or by the operands used.

Khoshaian and Abnous state in [KhAb95] that in nearly every language one can add, subtract, and multiply integers and floating point numbers with arithmetic operators like "+", ",-", and "**", although the implementations of integers and floating point arithmetic are quite different. Which operation is applied depends on the types of the operands.

Other examples are I/O methods in programming languages, like the method `java.io.PrintStream.print()` in Java, which works for strings, integers, and other data types as well (cp.[LaRa06, KhAb95]). Although overloading is a quite useful concept in object-orientation, it is not confined to object-orientation as it can be used in C or Pascal for certain operations as well. But in contrast to C or Pascal overloading can be used in object-oriented languages for any operations and any data type, not only for built-in types. Overloaded operations can also be combined with dynamic binding to produce more compact code, e.g. in Smalltalk, as no distinction between datatypes needs to be done in the code (cp.[KhAb95]). Authors like Lahres and Rayman do not make a connection between static polymorphism and late binding. They only see it as an important means for dynamic polymorphism (cp.[LaRa06]).

There is another possibility to distinguish different kinds of polymorphism. Cardelli and Wegner differentiate in [CaWe85] between universal and ad-hoc polymorphism. Universal polymorphism works on an unlimited number of types having a given common structure. In an implementation a universal polymorphic function will execute the same code for different types. Ad-hoc polymorphism on the other side works only on finite number of types which can be unrelated in their structure. Ad-hoc polymorphism has different method implementations for different types.

Universal polymorphism can be further divided into parametric polymorphism, and inclusion polymorphism. Cardelli and Wegner write on page 5: "In parametric polymorphism, a polymorphic function has an implicit or
explicit type parameter which determines the type of the argument for each application of that function. In inclusion polymorphism an object can be viewed as belonging to many different classes that need not be disjoint; that is, there may be inclusion of classes. Inclusion polymorphism thus represents inheritance and the possibility to use an object of a subclass instead of an object of the superclass. Parametric polymorphic methods are generic methods which can work for many types, doing basically the same kind of work for each type.

Ad-hoc polymorphism can as well be further distinguished into two concepts. First there is overloading; the second kind is coercion. Coercion converts an argument into the type expected by a method. As Cardelli and Wegener state it is often difficult to distinguish between these two kinds of ad-hoc polymorphism. They illustrate this by giving an example like this:

\[
\begin{align*}
4 + 2 \\
4.0 + 2 \\
4 + 2.0 \\
4.0 + 2.0
\end{align*}
\]

There are three possible ways the operation is defined. First, the operator "+" could be overloaded with four different versions of the operation. Second, it could be overloaded with two versions, one for the addition of integer and one for the addition of real. If one argument was integer and one real, the integer would be coerced into a real number. The third possibility is that there is only one implementation for real numbers and that integers are always coerced into real numbers.

In figure 3.7 we can see an overview of these different versions of polymorphism mentioned above.

![Figure 3.7: Different kinds of polymorphism](image)

We can see that of the different types of polymorphisms overloading and overwriting (or inclusion) are of special interest to us. Overloading and in a way inclusion are already realized in DMM and overwriting we want to achieve.

In this chapter we have seen different perceptions on what inheritance, subtyping and polymorphism are. In the next chapter we will investigate to what extend these ideas can be found in DMM and how they can be used to improve DMM.
Chapter 4

Improving the inheritance concept of DMM

Now that we had an overview over DMM and the most important concepts of object-orientation, we will combine these two aspects to improve inheritance and polymorphism in DMM.

In this chapter we will first explain the current state in DMM concerning inheritance and overwriting which is an important part of polymorphism. Afterwards we will look at how a type graph can be extended. We will then investigate possible terms of overwriting and finally we will evolve an adequate concept for DMM. We hope to achieve this goal by drawing comparisons between DMM and a popular object-oriented programming language. These comparisons do not only include inheritance and overwriting, but other concepts as well. This is to show that there are many resemblances between DMM and object-oriented programming languages, and that an intuitive definition of overwriting should take this into account. As programming language we choose Java to compare it to DMM, as today Java is one of the most widespread object-oriented languages in education and industry.

4.1 Current State

In this chapter we first try to find some similarities between DMM and Java. Later on we will look at the differences between them. In the last chapter we mainly investigated inheritance and polymorphism. Therefore we start our comparisons in the next subsection with inheritance and will later come to polymorphism, or more precisely to static polymorphism (overloading) as we can find some resemblances here between Java and DMM. When we come to differences in subsection 4.1.2 between Java and DMM we will also have a look at overwriting. In between we will examine other concepts regarding their conversions in Java and DMM. Table 4.1 gives an overview of the concepts
Current State

<table>
<thead>
<tr>
<th>Concept</th>
<th>Java</th>
<th>DMM</th>
</tr>
</thead>
<tbody>
<tr>
<td>inheritance</td>
<td>class inheritance</td>
<td>explicit subtyping</td>
</tr>
<tr>
<td>abstraction</td>
<td>abstract classes,</td>
<td>abstract nodes</td>
</tr>
<tr>
<td></td>
<td>abstract methods</td>
<td></td>
</tr>
<tr>
<td>behaviour</td>
<td>methods</td>
<td>rules</td>
</tr>
<tr>
<td>controlling/structuring</td>
<td>method calls</td>
<td>rule invoking</td>
</tr>
<tr>
<td>complex behaviour</td>
<td></td>
<td></td>
</tr>
<tr>
<td>case distinction</td>
<td>if - else expression</td>
<td>different left-hand sides of the rule, but same signature</td>
</tr>
<tr>
<td>overloading</td>
<td>same method name required</td>
<td>same rule name required</td>
</tr>
<tr>
<td>extensibility</td>
<td>overwriting or adding methods, adding classes</td>
<td>adding rules, adding types</td>
</tr>
<tr>
<td>redefining behaviour</td>
<td>only one method is applied, guaranteed by dynamic binding</td>
<td>several rules can be applied, branch in the transition system</td>
</tr>
</tbody>
</table>

Table 4.1: Comparison of concepts in Java and DMM

and their conversions we examine in this section.

4.1.1 Resemblances between Java and DMM

In this subsection we will concern ourselves with resemblances between Java or object-oriented concepts and DMM. We will start with inheritance. As we saw in chapter 2, inheritance can be used in DMM Type Graphs. It is more an explicit subtyping than class inheritance as we pointed out in the last chapter. Class inheritance somehow includes an implementation which is not present in DMM in the way it is in programming languages like Java which actually have class inheritance.

Another concept we often find in combination with inheritance in Java as well as in DMM is that of abstract elements. Nodes in DMM Type graphs can be marked as abstract, but this makes only sense if there are elements inheriting from it, as otherwise there would be no use for it in instantiations of DMM models. As we remember from section 2.3.1, the instance graph is typed over the concrete closure and the rule graph is typed over the abstract closure of a type graph, which ensures that nodes marked as abstract in a DMM type graph cannot be used in instance graphs (as abstract elements cannot have instances), only in rule graphs. In both instance graph and rule graph, subtypes of nodes of the type graph can be used instead of the supertype (unless the subtype is marked as abstract, too). In Java, the use of abstract classes and methods is possible, but no instances of abstract classes can be created and abstract methods are methods without an implementation. Here too, it is used to let other classes inherit the methods or interfaces of the abstract class.

Although there are no abstract rules in DMM, there is a certain similarities between methods in a programming language and the rules in DMM. Rules as
well as methods are used to express behaviour (in contrast to structure). They both have a signature, which includes parameters and the name of the method and DMM rules respectively. One could say that the context node of a DMM rule, or more precisely its type, owns the behaviour of the rule (cp.[Hau05], p.89), just as objects and classes in object oriented programming languages.

In addition, DMM rules can invoke additional behaviour that is other rules. In programming languages methods can call other methods. Rule invocation allows for structuring DMM and controlling complex behaviour like an object-oriented program would do. It also allows for reuse and an easier maintenance, which are also goals in object-oriented programming (cp.[Hau05], p.85 et sqq.).

There are other concepts in DMM connected to rules which remind of object-oriented programming languages. In DMM different rules can have the same name and even the same signature. This might be useful when the language engineer wants to distinguish different cases, as DMM has no if - else expression like Java or C♯. The rules then may have the same signatures, but they differ in the left hand side of the rule and they may differ in the right hand side as well.

It is possible to use this technique as a form of overloading by using different parameter types in the signature, and rules doing the same thing for different types. But keep in mind that there is no need for a resemblance in the pre- and the postconditions of rules, although they have the same name or signature.

As we already saw, big-step rules can be applied when they match and the previous big-step rule was applied successfully. Hausmann gives coherence between applying the DMM big-step rules and object-orientation. He claims that "the notion of spontaneously occurring (or pro-active) behavior is used in a number of advanced OO concepts (e.g., agent-based systems or user interfaces)" (cp.[Hau05], p.118). That means that even if something like matching of big-step rules does not occur in Java, it can be compared with other object-oriented concepts.

Also extension is not an object-oriented concept as such, object-oriented means (like overwriting or inheritance) can be used to achieve extension. For now we want to be content to just compare the possibility of adding or deleting elements in Java or DMM. As we have seen, we are not allowed to delete any elements in DMM rule sets or type graphs, as this would not be reasonable. It is only possible to add elements. This is also true for Java. When we have a running Java application (or with other words have an existing instance) we cannot delete anything with hindsight as it would be very unreasonable. When adding new elements (or changing them) a new compilation is required, but it is possible to do so.
4.1.2 Differences between Java and DMM

Until now we essentially concerned ourselves with similarities between Java and DMM. In the following subsection we will look at the main differences between Java or object-orientation and DMM.

As we know, it is possible that there are several big-step rules which match a given instance graph and can be applied to it successfully. In this case the corresponding transition system branches (cp. section 2.4 for a branching example) which is quite unintuitive for somebody used to Java in this situation. When there are for example two methods with the same signature defined in the same class, a compilation error would occur. That means it is not possible that two methods of one class can be applied at the same time. Of course, the big-step rules do not need to have the same context node type to be able to match both as they are not invoked. That means it is possible that the behaviour of two different types can be applied to the instance graph at the same time. This cannot happen in Java as methods are always called and therefore it is clear of which class the called method is.

The language engineer can create rules for a specific context node, which is a supertype of another node. When he wants different behaviour for the subtype, he can add corresponding rules to the rule set. But the original rules will not get overwritten as we are used to from object-oriented programming languages. Instead both rules can be applied in case they both match, which again would lead to a branch in the transition system. In Java it is not possible to have several possible method applications when inheritance is involved. Late binding ensures that the search for the right method to be applied begins as far down in the inheritance hierarchy of the classes as possible. The search in the classes in the hierarchy goes upwards and the first method found which fulfils the calling conditions (such as the right signature) is applied.

The fact that the transition system branches when a rule is defined for the supertype and redefined in the subtype is the main disadvantage of the inheritance and polymorphism concept of DMM, as DMM has many similarities with object-oriented programming, but this important concept of overwriting is left out. For someone used to Java it might be quite unexpected and unintuitive. Generally, when someone uses inheritance and wants to change the behaviour of the subclass, it is sometimes undesired that the old behaviour can still occur in its original form, although the new behaviour might be very similar to it.

For example, we have a class "car" and a method prepareCarWashing(), which includes "winding up the window". We have a subclass "car with long antenna" which has the method prepareCarWashing() as well, but this method does not only include "winding up the window", but also "take off the antenna". It would not be a good idea to apply the method prepareCarWashing() of "car" on its subclass, as this would probably require a new antenna. To prevent the method of the superclass from being applied is not yet possible in DMM as there is no overwriting concept and the rule of the supertype might
Hausmann admits that this regulation can be uncomfortable, but he claims that a purely additive extension mechanism is beneficial as users who understand a specified construct once can rely on this knowledge ([Hau05] p.120 et seq.). We think he is right concerning the deleting part, but we also think that it would be much better, if overwriting was allowed for reasons we will point out later in this chapter.

Note that inheritance is not only a syntactic notion, it is a semantic concept. DMM as well as a programming language like Java can provide the means for the use of inheritance, but they cannot guarantee that the language engineer or the programmer use these means sensibly. How it is used is eventually left to the language engineer or the programmer.

4.2 Extension of a Visual Modelling Language (VML)

Sometimes it is desired to define the semantics of a modular or extensible VML. Maybe a new language is to be defined and the language engineer does not want to begin from scratch. He could use an already defined language and add the required elements, for example a domain-specific language based on UML. It is possible to define this kind of languages in DMM.

In this section we want to examine how we can extend the semantics of a VML. We want to see if we can only add elements to the semantics or if there are other possibilities. We will see that we can add elements to the type graph or rules to the rule set and that each of these possibilities may lead to certain problems. We will pick two problems for each of these two extension possibilities and will look if it is possible to solve them with the means available in DMM.

Before we actually look at what we can add to the semantics of a VML, we first want to refer to [Hau05], p.119 et seq., where Hausmann gives an extension mechanism which forbids deleting any element, not only in the type graph, but also in the rule set. It is only possible to add completely new behaviour or extend existing behaviour, not to overwrite or delete it. The extension of the type graph is also possible, whereas new syntactic elements often bring new behavioural elements. He claims that the restrictions give a necessary rigour to a language specification and this way the user can rely on the knowledge he has.

The restriction not to delete anything is reasonable, as users cannot accidentally use deleted elements. They can trust that in a form or another all their knowledge about a VML is still valid. The restriction that nothing should be overwritten is not so easy to understand as the argument that the user can rely on his knowledge still holds, although there might be elements and behaviour he did not get to know yet.
Adding elements to the VML is (as mentioned earlier) allowed. When we want changes in the structure of the VML, we add elements to the type graph as it imposes the structure on the instance graphs and on the rule graphs. If we want additional behaviour, we add rules to the rule set.

4.2.1 Structural extensions

To come to the first problem we want to examine, let us think of the following example. A language engineer wants to extend a type graph consisting of two types $A$ and $D$ connected to each other like shown in figure 4.1 with new types inheriting from $A$ and $D$. The nodes $B$ and $C$ shall inherit from $A$ and the nodes $E$ and $F$ shall inherit from $D$. The resulting graph looks like displayed in figure 4.2. But actually the language engineer does not want every subtype of $A$ being able to connect with every other subtype of $D$. $B$ should only be able to connect with $E$ and $C$ should be able to connect with $F$.

The graph in figure 4.3 is not sufficient to express this situation, as the connection from $A$ to $D$ is still inherited, so that for example $B$ can still connect with $D$, $E$, and $F$. A solution would be to allow inheritance not only between nodes, but also between edges. This would include a change of different definitions in DMM theory and would be rather elaborate. A more elegant solution would be to just include OCL (Object Constraint Language, cp.[OMG07]) constraints as they are already intended in DMM (cp.[Hau05]). These OCL constraints can ensure that the connection between $A$ and $D$ is for those two classes only, whereas $B$ can only connect with $E$ and $C$ only with $F$.

As we are engaged with extending type graphs with inheritance, this problem may occur at some time. But as using OCL solves it quite neatly, we do not need to bother about it for the rest of this thesis.
4.2.2 Behavioural extensions

The second problem we want to discuss can be identified with our example from chapter 2.4. First we want to extend the type graph by using inheritance. Thinking of an Activity Diagram, we want to have something resembling an ActivityNode, a DecisionNode, and later a ForkNode, all inheriting from our type Node. The type graph can be seen in figure 2.13 on page 22. These extensions would be no major problem. But the case is different when we also want to change the behaviour of certain elements of the type graph.

For this we will first go back a step. Assume that we have a type graph like in figure 4.4. It resembles the one in figure 2.13 on page 22, except that the Tokens which belong to the Node are not stored in a queue. The token flow is displayed in the rule node.flow()* in figure 4.5. As we can see, it is indeterministic which Token flows first when there is more than one. Any Token contained in node of type Node can flow to the next element of the type Node.

We now want to extend the type graph (again), so that Tokens are actually stored in a queue. For "normal" control tokens like in UML activity diagrams it would actually make no great difference. But if we think of Tokens associated with something (like object tokens in UML activity diagrams) it would make a difference in which order they would arrive at a node. (See [OMG07] p.314 et sqq. for more information about control and object tokens.) The new type graph with Tokens stored in a queue then looks like the one in figure 2.13. We have two new connections between Node and Token to be able to distinguish first and last element from the other elements. We also have an association from Token to itself to appoint the next element in the queue.

To benefit from this new structure we want to change the behaviour of the token flow. We only want the first Token to flow from one Node to another, not an arbitrary Token. In figures 2.14 to 2.18 on page 23 we can see the rule set we want to use.

Now we meet the problem that the old rule (without storing tokens in a queue) can still match as its left hand side is contained in the left hand side of the new rule (the one taking the token queue into account). Whenever the structure of the left hand side of the new rule occurs in an instance graph, the one of the old rule does as well. Therefore there will be a branch in the transition system whenever the new version of the rule can match as the old one can match as well. This is why we cannot be sure that the first Token in the queue will flow to the next Node. In addition if the old rule is applied at some time, it would destroy the queue where the Token was in. The new rule might not be able to match again.

The case is only a bit different if we do not only add means to store the Tokens in a queue, but also add inheritance to the type graph as we wanted in the first place. In addition we mark the type Node as abstract as it is not necessary to have instances of it. Every Node should either be a DecisionNode.
or an \textit{ActivityNode}. The type graph could look like the one in figure 4.6.

In this case we have the same problem, even if we would make special rules for \textit{DecisionNode} and \textit{ActivityNode}. The rule with an element of type \textit{Node} as context node could still match as well.

We could solve problems like the one above by overwriting rules, that is redefining rules for the subtype. In the example we could overwrite \texttt{node.flow()}\texttt{*} with rules with context nodes of the type \textit{DecisionNode} and \textit{ActivityNode}. The extension of a VML would be more precise and more cases would be expressible.

We see that although Hausmann suggests forbidding overwriting, it has several advantages to allow it nevertheless. Besides the argument above, that
we could solve the described problem and its like we have the argument of an intuitive definition. As already mentioned, DMM has many object-oriented concepts. Hausmann states in [Hau05] that he assumed object-orientation knowledge in the target audience of DMM (p.83). He also says "The prime goal of DMM is to enable the formulation of semantics for Visual Modeling Languages in a way that is easily perceived by our intended target group." (p.117). It would be much more intuitive and understandable if overwriting was allowed, as it is an important object-oriented concept used in object-oriented programming languages. And as we would not allow to delete anything either, the arguments that users cannot accidentally use deleted elements and that they can trust that in a form or another all their knowledge about a VML is still valid would still be true.

Of course, problems may also occur when the language engineer defines a VML in the first place. There must always be prudence when defining or extending a VML. But some of the problems the engineer may face would be easier to solve when overwriting was allowed.

4.3 Types of overwriting

In this section we want to examine different approaches for overwriting which are suggested in literature. We want to find out which approach fits best for DMM. First we will have a look at the three possibilities distinguished by Khoshaian and Abnous which we presented in chapter 3.3 on page 34 and the following. We remember that they proposed in [KhAb95] the following approaches: no redefinitions at all, arbitrary redefinitions, and constrained redefinitions. Afterwards we turn towards a different idea. We want to investigate to what extent the transition system should be included in the definition of overwriting in DMM.

4.3.1 No redefinition of inherited rules

No redefinition of inherited rules means that a node which inherits from another node cannot change the definition of an inherited rule for itself. The rule with the context node of the superordinate type is valid for the context node with the subordinate type as well. The language engineer can only add new behaviour, that is new rules, but he is not able to suppress the behaviour defined for the supertype for the subtype.

This is the same situation as we have in DMM now. We already argued that this possibility is not desirable for DMM; it is actually no overwriting at all. Therefore we will not consider it any further.
4.3.2 Arbitrary redefinition of inherited rules

Arbitrary redefinition of inherited rules means that there are no restrictions to the way an inherited rule can be overwritten. There is no need for resemblances between the rule with the context node of the supertype and the rule with the context node of the subtype. There can be completely different parameters in the rule signature. Neither the left hand side of the overwriting rule nor the right hand side need to resemble the ones of the overwritten rule.

Arbitrary redefinition implies that the inheritance used is neither interface inheritance nor implementation inheritance. For interface inheritance the specification of the supertype must be inherited by the subtype. The precondition of the subtype is similar to the left hand side, the postcondition to the right hand side of a rule, but these can be completely different from the ones of the supertype.

When using implementation inheritance, the subtype inherits all responsibilities, all rules, all data, and all abilities of the supertype. Behaviour can be adopted or partly be overwritten. But when the rules can be arbitrary redefined, the subtype has not the same responsibilities as its supertype and as the case may be not the abilities of its supertype.

We can also see that the Liskov substitution principle can not be fulfilled, either, as not all properties of the supertype need to be true for the the subtype. In addition the user of a VML can not be sure whether an inherited rule did similar things as the original rule, which could be confusing for him.

We can conclude that arbitrary redefinition in DMM is not what we want. If we wanted complete new behaviour, we should either not use inheritance or we should add the new behaviour to the rule set without overwriting. This makes sense when the left hand side of the supertypes’ rule is completely different from the one the language engineer wants to create. The original rule could still match for the subtype, but the new rule could as well under the right circumstances. This would be like an if - else distinction.

4.3.3 Constrained redefinition

The third alternative suggested by Khoshanian and Abnous is constrained overwriting. In chapter 3 we described one possibility to constrain the redefinition of a method. Transferring this possibility from methods to DMM rules we can see that it includes constraints on signature, pre- and postcondition, that is the left hand side and the right hand side of a rule. Signature, pre- and postcondition of an overwriting rule must conform to the ones of the overwritten rule. But we can also imagine other constraints, like only the signature must be conform to the overwritten one, or the signature and the left-hand side of the rule must conform to the ones of the overwritten rule.

In the next paragraphs we will first say what we mean when we talk of "conform". Later we will investigate the different possibilities of constrained
Types of overwriting

We will examine the possibility that only the new signature must conform in a way or another to the one which is overwritten. Then we will have a look at the alternative where the new signature and the precondition must conform to the overwritten ones, but not the postcondition. Finally, we will analyse the last possibility where all three, the signature, the precondition and the postcondition must be conform to the overwritten ones. To make investigating these possibilities easier, we will give an example beforehand and try to argue with it later on.

The term "conform"

Before we think about how reasonable possible constraints for overwriting rules are, we should first think about what "conform" means. Khoshafian and Abnous state in [KhAb95], p.110 et seq. that there are two possibilities when a signature of a function or method is conform (such as subtyping) to the signature of another. They distinguish between the covariance rule and the contravariance rule. We remember from section 3.2.2 that the covariance rule indicates that the parameters and return values of the subordinate function are subtypes of the parameters and return values of the superordinate function. The contravariance rule on the other hand indicates that the parameters of the subordinate function are supertypes and the return values are subtypes of the ones of the superordinate function. We also remember that the covariance rule is more intuitive and the contravariance rule guarantees strong typing as already pointed out in section 3.2.2, page 30 and the following. This is the reason why they propose to use the contravariance rule. Besides these two possibilities to make a signature conform to another, there is a third one which is for example used in Java. This possibility is identified by leaving the types of the parameters and return values as they were in the overwritten signature.

We will now get down to the conforming of the pre- and postcondition of the overwriting rule with the overwritten rule. Khoshafian, Abnous and Liskov say that the precondition of the overwriting method must be the same or weaker than the one of the overwritten rule and the postcondition must be the same or stronger than the one of the method which is overwritten (substitution principle). But we can also think of an alternative which might sound more logical at first: an alternative where both pre- and postcondition have to be stronger than the overwritten ones.

When we get back to DMM, we must ask ourselves the question when a pre- or postcondition is stronger (or weaker) than the original ones. In a graph we can add elements (extend it) or delete elements. When adding elements, e.g. on the left hand side of the rule, we have more elements which must be present (or must not be present, when adding a NAC), therefore we have more requirements we have to fulfil. Adding elements to the left hand side of the graph would therefore strengthen the precondition.

When we delete elements from the left hand side of the graph, those elements
Types of overwriting

can be present, but there is no need for them anymore (and for the NACs there is no need not to be there anymore). Deleting elements from the left hand side would therefore weaken the precondition. This applies of course to the right hand side, too. When adding elements the postcondition becomes stronger, when deleting elements the postcondition becomes weaker.

In the following sections we will investigate what "conform" actually means for DMM. We will have a look at the different possibilities and then decide which one fits best for DMM.

DMM example with extensions

In the following we will have a look at an example, where overwriting would be useful. We already know the type graph in figure 4.6 on page 44 from the last section. We now want to extend this type graph by a "ForkNode" as can be seen in figure 4.7.

![Figure 4.7: Extended Type Graph with ForkNodes and ForkEdges](image)

A ForkNode can have several outgoing edges and whenever a Token flows from this ForkNode, every outgoing edge should carry a copy of this Token to its adjacent node. To be sure that every edge carries a Token and not only an arbitrary one, we change the kind of storage for the edges. We store them in the queue like we do with the Tokens. This can ensure that the edges are all processed. But as this is only necessary for the outgoing edges, we create a new kind of edges, so called ForkEdges. Their startnode is a ForkNode, their endnode is an arbitrary Node.

For this new ForkNode we need to define new forknode.flow rules as the original one would only send a Token to one other Node, not to all adjacent ones. Instead of one new rule we need a rule set of four rules, two big-step and two small-step rules to allow recursion.
Types of overwriting

In figure 4.8 we see the rule when there is only one outgoing edge (although this case is not necessarily intended it should be covered). This rule bears likeness to the original rule. The first invocation makes sure that the Node $n_2$ gets a copy of the Token; the second invocation makes sure that the Forknode for $n_2$ does not have the first Token in queue anymore.

The second rule in figure 4.9 covers the case that there is more than one outgoing ForkEdge. The first invocation is the same as in the previous rule, but the second invocation starts a recursion, to assure that all adjacent Nodes to outgoing ForkEdges receive a Token.

The third and the fourth rule can be invoked by this rule. The third rule in figure 4.10 does the same as the second rule. It has an additional parameter in its signature, which is the ForkEdge handled next. The fourth rule in figure 4.11 covers the case that the actual ForkEdge is the last one in the queue. Here the invocation which dequeues the Forknode appears again instead of an additional recursion call. After dequeuing $tok$ from for $n_2$, it has no connection to any other object in the graph. Later we could add a rule which deletes such loose Tokens.
Types of overwriting

Figure 4.10: Rule Forknode.flow(e1:ForkEdge) with at least one edge left which is not processed yet

Figure 4.11: Rule Forknode.flow(e1:ForkEdge) with no edges left which are not processed

The two small-step rules node.enqueueCopy(originaltok:Token) in figures 4.12 and 4.13 create a copy tok of the original Token originaltok and enqueue it to Node node. One rule is for the case that there are already Tokens in the queue of node. Tok is then denoted last object and a link "next" is created from the Token which was the last one before. The other rule is for the case that there are no other Tokens in the queue and tok is the first one. Tok is then denoted first and last element in the queue.
Types of overwriting

![Diagram of node.enqueueCopy(originaltok:Token)](image)

**Figure 4.12**: Rule `node.enqueueCopy(originaltok:Token)` with at least one other token in the queue of node

![Diagram of node.enqueueCopy(originaltok:Token)](image)

**Figure 4.13**: Rule `node.enqueueCopy(originaltok:Token)` with no tokens in the queue of node

**Signature**

In this section we will investigate overwriting if our only constraint is that the new signature must be conform to the old one and how the conformance should actually look like. As rules in DMM do not have a return value, only parameters, we must only decide what types for the parameters can be chosen. That the name is still the same and the context node is in the inheritance clan of the original context node is understood.

We have the three possibilities that the parameters must either be covariant, contravariant, or invariant. As already mentioned, the covariant rule is more intuitive than the contravariant rule. But one can argue if covariance is more intuitive than invariance when looking at the parameters of a rule. Before deciding which possibility to take, we want to take a look at how overwriting is handled in Java.

In Java only the exact same type of the parameter in the overwritten rule can be chosen in the overwriting rule (the parameter type is invariant). When choosing a subtype or a supertype of the parameters of the original rule, the new rule does not overwrite the old one, but overload.

As overloading is also possible in DMM, we think this is a good solution
Types of overwriting

for DMM as well. When the language engineer wants rules for the subtypes or the supertypes of the parameters, too, he can simply overload the original rule. This solution is intuitive and does not require the distinction between co- and contravariance in the signature.

When we get back to our example, the overwriting of the rule node.flow()* works, as our new rule set does not violate the constraint that the signature must be the same aside from the new context node which is supposed to be in the inheritance clan of the old context node. We should add that only the two rules fornode.flow()* would overwrite node.flow()*: The two small-step rules with the parameter of type Forkedge would not overwrite node.flow()*, but would be simply added to the rule set without overwriting anything. They can only be invoked and cannot match without invocation like the big-step rules.

However, this possibility of constraining overwriting only by signature has certain disadvantages. It is possible to redefine the behaviour arbitrarily. We could for example create a rule like in figure 4.14.

![Figure 4.14: Bad example of overwriting when only constraining the signature](image)

This rule fornode.flow() has nothing in common with the original rule node.flow, except for the name. Instead of sending a Token over an adjacent Edge to another Node, a new ForkEdge and a new Node are created. When this rule replaces node.flow, it has a complete new behaviour. One could even say that ForkNode is no Node as it has different behaviour and therefore should not inherit from Node.

Although this alternative is the way overwriting is handled in Java, we claim that this constraint is to loose and we will look for a stern limitation.

Signature and Precondition

In this section we will determine if the constraint that signature and precondition of an overwriting rule must conform to the ones they overwrite is sufficient or not. In the last section we already decided that the signature of an overwriting rule is to be the same as the one in the overwriting rule apart from the context node.

Before we can proceed investigating the possibility of constraining, we first have to decide whether the new precondition should be stronger or weaker (or the same) than the old one. For this we will have a look at our example. In our
example we want a new rule (or more precisely a rule set) for node.ow which allocates copies of the first Token in queue of the Forknode over all outgoing ForkEdges to all adjacent Nodes. As already mentioned, we want to achieve this by using a queue as data structure. Therefore we certainly want to use this new data structure consisting of two new associations between Forkedge and Forknode in our new rule. If we would not use it, there would be no possibility to guarantee that each adjacent Node would get exactly one copy of the Token.

When weakening the precondition to overwrite the original node.ow, there must be less (or the same amount of) elements in the left hand side of the overwriting rule than there are in the overwritten rule. But those elements present must also be present in the original rule. Hence, we could not use our two new connections in the left hand side of the overwriting rule as this would strengthen our precondition, not weaken it.

We can conclude that changing the precondition like the Liskov principle suggests does not fulfill our requirements.

Now that we have resolved that overwriting preconditions should not be weakened, we should investigate if we can claim that they must become stronger (or as the case may be stay the same). When we allow overwriting preconditions to become stronger, the Liskov substitution principle is violated. Anyway, there are no other arguments at this point to forbid this possibility. In addition this alternative is intuitive and enables us to use the precondition we created for our new forknode.ow. Even if we would insert a premise rule, the case would still be the same. Therefore it is possible to ask for a precondition to become stronger or to stay the same. Back to the problem, if the constraint that the new signature and the new precondition must conform to the old one is sufficient, we can observe that we can use our intended rules forknode.ow to overwrite the old rule node.ow.

Nevertheless, without constraining the postcondition we have a similar problem as with only constraining the signature. The rule in figure 4.15 could for example be used to overwrite the rule node.ow. Here all elements which must be present are deleted except for the context node. No Token flows from one Node to another. The same argument we used in the previous paragraph holds for this rule as well: When this rule replaces node.ow, it has a complete new behaviour. One could even say that ForkNode is no Node as it has different behaviour and therefore should not inherit from Node.

We can conclude that the constraint described in this paragraph is still too weak, although it is a bit stronger than the one investigated before. We would like to get an even stronger constraint, if this is possible.

Signature, Pre-, and Postcondition

In the following we will examine our last constraint where all three, signature, precondition, and postcondition, must conform to the ones they overwrite.
When we transfer the results of the last two investigations to this one, we can say that the signature should not change besides the context node and the precondition should become stronger than the original one or stay the same. It remains to decide whether the postcondition of an overwriting rule should be stronger or weaker than the one of the overwritten rule. Here we can argue again as above: we might want to use some of the newly defined elements in the type graph which are not added through an inheritance relationship, but through a normal one. In our example we want to use the queue in which ForkEdges are stored, which is not possible when the postcondition becomes weaker. This means that the postcondition can be at most stronger but not weaker than the original one.

Now that we have determined that when we overwrite rules the postcondition should not be weakened, we should investigate if this constraining restricts the overwriting in our example too much. On first sight one might get the impression that none of the two forknode.flow()* rules are extensions of our node.flow()* rules, because in node.flow()* we invoke the rule node.enqueue(tok:Token), whereas in forknode.flow()* we invoke a node.enqueueCopy(tok:Token). In addition, one of the forknode.flow()* rules does not invoke node.dequeue(), but the new rule forknode.flow(e:ForkEdge).

But let us think about how the new rules look like, if we do not use any invocation. It is more elaborate to model our rules without invoking other rules, especially as we used invocations to gain recursion. We will have more new rules than just four as we need a rule for every number of ForkEdges leaving from a ForkNode. Therefore we need to restrict the number of Edges to receive a finite number of rules. Melting the invoked rules together with the invoking rules, we extend both the left hand side and the right hand side of the invoking rule.

In every rule’s right hand side the adjacent Nodes of ForkNode have a new Token, which is a copy of the one which was initially first in the queue of the ForkNode. This original Token is no longer enqueued to the ForkNode. Actually it has no connections to any other element in the graph as we pointed
out earlier. We can also model the rule \texttt{node.flow()} without invocations. And when we compare these rules without invocations, we can find that they look a lot more alike than before. The right hand side looks similar to the ones of \texttt{forknode.flow()}*. The \textit{Token} which was supposed to flow is no longer enqueued to \texttt{node}. Instead one arbitrary adjacent \texttt{Node} element has a new \textit{Token} (which happens to be the same as the one enqueued to \texttt{node}). These elements are all in the right hand side of \texttt{forknode.flow()}* as well, the difference is that in the latter all adjacent \texttt{Node} elements have a \textit{Token}, not only one, and that the original \textit{Token} has no connections to other elements anymore. That these new \textit{Tokens} are only copies of the original one is of no importance.

To make this explanation more understandable, we will show the melting of invoking and invoked rules. First we will show \texttt{node.flow()} and one of the \texttt{forknode.flow()}* rules again in figures 4.16 and 4.17 so that we can draw a direct comparison. In figures 4.18 and 4.19 we can see a case that can occur when melting the \texttt{flow} rules with the invoked rules. We picked the case when there is at least one other \textit{Token} in the queue of the start \texttt{Node} of the flowing \textit{Token} as well as in the queue of the end \texttt{Node}. The graphics in figure 4.20 and in figure 4.21 show the left hand sides only of the two rules in figures 4.18 and 4.19, whereas the right hand sides of these rules can be seen in the figures 4.22 and 4.23. Here we can see clearly that the left hand sides and the right hand sides of \texttt{node.flow()}* are contained completely in the ones of \texttt{forknode.flow()}* except for the names \textit{originaltok} and \textit{tok} in figures 4.21 and 4.20. But as the names in the instance graphs would probably differ from both as well, it seems that names of instances are not that important. In addition the types belonging to the \texttt{Forknode} rule set may not be the same types as in the rule set belonging to "normal" \texttt{Nodes}, but members of their inheritance clans.

To decide if the \texttt{forknode.flow()}* rules are extensions of \texttt{node.flow()}*, we must first show that \texttt{enqueueCopy} (as abbreviation we use the rule's name instead of the whole signature) is an extension of \texttt{enqueue}. We can ignore the fact that they have different signatures, as they cease to exist when we merge these rules with our \texttt{flow} rules. Therefore we will concentrate on the left and the right hand side of the graph.

As we can see the rule \texttt{node.enqueue()} in figure 2.15 on page 23 is contained completely in the rule \texttt{enqueueCopy} in figure 4.12 on page 51 and the rule \texttt{enqueue} in figure 2.16 on page 23 is contained completely in rule \texttt{enqueueCopy} in figure 4.13 on page 51, including each element's name. Therefore we can certainly say that the left and the right hand sides of each \texttt{enqueueCopy} rule are extensions of the left and the right hand sides of one of the enqueue rules. Ignoring other invocations for now, merging these small-step rules with their invoking flow rules would lead to the rule \texttt{forknode.flow()} being an extension of the rule \texttt{node.flow()}.

When we now consider the other invocations of \texttt{forknode.flow()}, we can observe that in every rule either \texttt{flow(e:ForkEdge)} or \texttt{dequeue(tok:Token)} is invoked. And as there is always a finite number of \texttt{ForkEdges} and each of them is
Types of overwriting

Figure 4.16: node.flow()* with invocations

Figure 4.17: fornode.flow()* with invocations

Figure 4.18: possible node.flow()* without invocations

Figure 4.19: possible fornode.flow()* without invocations
Types of overwriting

Figure 4.20: Left hand side of a possible fornode.flow(*) without invocations

Figure 4.21: Left hand side of a possible fornode.flow(*) without invocations

Figure 4.22: Right hand side of a possible fornode.flow(*) without invocations

Figure 4.23: Right hand side of a possible fornode.flow(*) without invocations
only once parameter in \textit{flow(e:ForkNode)} for a specific \textit{Token} at the \textit{Forknode}, at some time \textit{dequeue(tok:Token)} is invoked. This means that our new rules are extensions of our old \textit{flow} rule.

We can conclude that constraining the postcondition in addition to precondition and signature is possible. The constraint is quite severe, but it is an improvement to not being able to overwrite at all. Being a language engineer one has to be wary how to model certain aspects, especially when using invocations. Taking invocations into account is difficult, because there might be several overloaded rules with the same signature which get invoked eventually in the overwritten rule (or in one of its invocations). In this case there must be an according invocation for each of them in the overwriting rule (or in one of its invocations), which means either the same rule or one which is an extension of its left and right hand side.

\subsection{4.4 Labelled Transition System}

In the following we will investigate how the labelled transition system acts when defining overwriting. Before we start, we have to say that the constraining for signature, pre- and postcondition is still valid, as the argumentation still holds although we look at the transition system. That means that we want to keep the constraints we already found and want to look in this section whether we need additional constraints to gain a specific behaviour in the transition system.

We could consider constraining the transition system as well, but then we first have to think about in which way this is reasonable. The most intuitive choice would be to allow the extension of the transition system as we want to extend the behaviour of certain objects. This means that there can be additional states to already existing states and additional transitions.

This can have different effects. If there is no fork before overwriting a certain rule, there might be none afterwards, but maybe some new states in between. But there might also be a new fork in the transition system which leads to a new derivation sequence.

On the other hand, if there is a fork before overwriting it must not disappear afterwards. A problem we have to face when dealing with a transition system is that it looks always different for differing start graphs. When changing the rules we would have to check if the extension is correct for every derivation sequence and for every start graph.

Let us have a look at the two rules in figures 4.24 and 4.25 and the instance graphs in figures 4.26 and 4.27 to investigate what effects our constraints in the rule graphs have on the transition system.

In the first rule \textit{C.DoSth()} there are altogether three nodes of the types A, B, and C on the left hand side of the rule, whereat the node of type A
is connected to each of the other nodes. On the right hand side there is an
additional connection between the nodes of type $B$ and type $C$. This rule is the
original rule and we emanate that $ChildOfC$ inherits in the type graph from $C$.
The rule $ChildOfC.DoSth()$ is to overwrite the first rule and looks like it except
that on the left hand side there is an additional node of type $D$ connected to $B$
and instead of a node of type $C$ there is a node of type $ChildOfC$. On the right
hand side there is an additional connection between the nodes of the types $C$
and $D$.

![Figure 4.24: Rule C.DoSth()](image1)

![Figure 4.25: Rule ChildOfC.DoSth()](image2)

When we start our investigation with the instance graph in figure 4.26 we
can see that we can apply $C.DoSth()$ on this graph (before we overwrite the
rule). After overwriting we can apply $ChildOfC.DoSth()$ (but not $C.DoSth()$).
This means that the transition is the same (if we do not mind that the context
node is a descendant of the original one). The state is not exactly the same,
though, as the new rule does not do exactly the same thing as the old one. If
we hold on to the constraints we wanted to use, the new state is an extension
of the state we would get when we applied the old rule. When applying more
overwriting rules the extension could amplify.

Another problem can be seen when we look at the second instance graph in
figure 4.27. The rule $C.DoSth()$ would match as above, with no problems. But
the rule $ChildOfC.DoSth()$ cannot match as a necessary element of type $D$ is
missing. This means that when applying $C.DoSth()$ the transition system has
an additional state and an additional transition compared to the case when we
overwrite the rule with $ChildOfC.DoSth()$. The problem could only be solved
when the Liskov substitution principle would be fulfilled in the constraining,
because then the overwriting rule could be applied whenever the overwritten
rule could. But as argued above this is not desired.

We can say that we do not need to restrict the transition system as the
constraining of signature, left and right hand side already has certain effects
on the transition system. Additional constraints would either constrict too
much, they would be contradictory to the other constraints, or they would be
redundant and make the constraining unnecessarily more complicated.
Explicit overwriting and implicit overwriting

4.5 Explicit overwriting and implicit overwriting

As we saw in section 3.2, class inheritance is always explicit, whereas subtyping is either implicit or explicit. But what about overwriting? In programming languages like Java there is no need to explicitly say that a method in the superclass is overwritten by a method in the subclass. Overwriting is only possible when using the exact same signature, otherwise the method is overloaded. The most specialised method possible is used. As there can only be one method with the exact same signature in a class, it is always clear which method is overwritten, even if there are some overloading methods as well. Those have different parameters and therefore are not the ones which are overwritten or overwriting.

Unfortunately, the case is not that easy in DMM. We remember that there can be several rules with the same signature, for example as a substitution for an if-else expression. There can also be rules with the same signature and the same precondition as there is no necessity for determinism (although it may be wanted).

When overwriting we can distinguish different cases. The easiest possibility is that the signature of the overwritten rule is unique and that there is only one rule which ought to overwrite this first rule. This would conform to the case in Java described above. If only this possibility existed, there would certainly be no need for explicit overwriting.

The case is similar if there is a unique signature for the rule to be overwritten, but several overwriting rules. (This is like in the example we presented section 4.3.3 on page 48 et seq.) Here again it is clear which rule is overwritten.

But what if there are additional rules for the subtype (besides the overwriting rule) which are not supposed to overwrite the rule of the supertype, but to overload it without parameter change, for example to gain case distinction? Then it would still be clear which rule is overwritten, but not which rules are overwriting.

We will look at yet another case. Consider that there are several rules for the supertype with the same signature. Now there is only one rule for the subtype which is to overwrite all or only part of the rules of the supertype. We could argue that those rules certainly have different left-hand sides, but firstly, this does not need to be the case, and secondly the overwriting rule
can be an extension of more than one rule as can be seen in the example in figures 4.28 to 4.30. There are two rules $\text{A.accomplishTask()}$ with different left sides. We assume that $\text{ChildofA}$ is a descendant of $\text{A}$ in the type graph and that $\text{ChildOfA.accomplishTask()}$ in figure 4.30 is supposed to overwrite $\text{A.accomplishTask()}$. We can see that the left hand side of the rule is an extension of both the rules of $\text{A}$. Now we do not know whether it overwrites the one in figure 4.28, or in figure 4.29, or both.

Of course we could also look at the case where there are several rules for the supertype and several rules in the subtype, but this can be derived from the other cases.

![Figure 4.28: Rule A.accomplishTask() I](image)

![Figure 4.29: Rule A.accomplishTask() II](image)

![Figure 4.30: Rule ChildOfA.accomplishTask()](image)

We can conclude that unlike in Java, it is necessary in DMM to explicitly overwrite the rules. How this is done exactly will be shown in the next chapter.

### 4.6 Findings

We can resume that if a language engineer wants to use inheritance and overwriting of rules in his DMM specification of a visual modelling language, he must consider several restrictions.

A rule can be overwritten by another rule if the context node of the overwriting rule is in the inheritance clan of the overwritten rule’s context node.
Findings

The overwriting rule must have the same name and the same parameters as the rule it overwrites.

Additionally, the precondition of the overwriting rule must be stronger than the one of the rule to overwrite. This means that all elements which are in the overwritten rule’s left hand side must be in the left hand side of the overwriting rule as well. Nodes and edges can be added though. We assume that all premise rules are melted with their invoking rules. When there is a NAC in the rule which is to be overwritten, this NAC must be in the overwriting rule as well as NACs are part of the precondition. Since the precondition is to be strengthened, there can be additional NACs in the overwriting rule.

The postcondition of the overwriting rule must be stronger than the one of the rule which is supposed to be overwritten. To check if this is the case, all invocations need to be melted with their invoking rule. UQSs do not need to be unfolded to examine this. The same conditions which hold for the left hand sides of the rules hold for the right hand sides: all elements which are present in the overwritten rule’s right hand side must to be present at the overwriting rule’s right hand side.

A rule must explicitly be overwritten and a rule can overwrite more than one other rule and of course a rule can be overwritten by more than one rule.

The notion of a rule matching an instance graph changes with the introduction of overwriting. A rule can only match if there is no other rule which overwrites it (directly or transitive) and which also matches.

We finally need a graphical representation when a rule overwrites another one. We suggest an inheritance arrow like it is used in type graphs, but with a dashed line instead of a continuous one to connect the overwriting rule with the overwritten rule. In figure 4.31 rule $B.doSth()$ overwrites rule $A.doSth()$.

![Figure 4.31: Rule A.doSth() is overwitten by rule B.doSth()](image-url)
Chapter 5

DMM Formalisation

In the last chapter we determined how overwriting in DMM should look like. We decided that we want to constrain the signature of the overwriting rule insofar as besides the context node the signature must stay the same as in the overwritten rule. The context node has to be of a type contained in the inheritance clan of the original context node's type. Or in other words it has to be of a subtype of the context node's type. We figured out that the precondition and the postcondition of the overwriting rule can stay the same or be an extension of the ones in the overwritten rule. That means there can be additional elements, but none can be deleted. We also decided that invocations are treated as if they were melted with their invoking rules. In addition we found that overwriting must be explicit in DMM unlike in Java where it is implicit.

In chapter 4 we gave an informal description of our findings. In this chapter we want to formalise these findings to make them more precise and to make it easier to use them for possible tool support. We will first define a new kind of DMM rule such that explicit overwriting is taken into account. Afterwards we will redefine the term "matching" to adapt it to the overwriting in DMM. Finally we will describe the relationship between overwriting and overwritten rules.

5.1 DMM Final Rule

In this section we will redefine the DMM final rule which was introduced in section 2.3.5 on page 21. We remember that a DMM final rule is a DMM rule with invocations. It is a six-tuple, including the left hand side and the right hand side of the graph, the negative application condition, an invocation relation, parameters, and a name. To allow explicit overwriting we need a seven-tuple, which additionally includes a set of rules which are overwritten by the rule. This set can be empty, if the rule does not overwrite anything.
Definition 5.1. DMM Rule with Invocations and Overwriting (final rule)

A DMM Rule with Invocation and Overwriting $r$ is a seven-tuple with $r \in r_{DMMInOv}$ and

\[
\begin{align*}
L, R & \in G_{DMMR} \text{ ordinary rule graphs,} \\
NAC & = \{L\} \subseteq G_{DMMR} \text{ a set of negative application conditions,} \\
inv & \text{ an invocation relation as described in definition 2.20} \\
Params & \subseteq L, \forall p \in (Params \cup \{x \mid x \in params^{inv}\}): l_N(p) \neq \bullet, \text{ name } \in \Lambda \backslash \bullet, \text{ and} \\
overwriting: & \text{ is a set of DMM rules which are directly overwritten by } r_{DMMInOv} \\
\end{align*}
\]

The set overwriting contains all rules which are overwritten by the rule the set belongs to. As already mentioned there can be more than one rule in this set and different rules can have the same set. For the set overwriting we need to clearly identify the rules which are overwritten as there may be several rules with the same signature and maybe even with the same left or right hand side.

Overwriting is transitive. That means when a rule $r_2$ overwrites a rule $r_1$ and another rule $r_3$ overwrites rule $r_2$, $r_3$ also overwrites $r_1$. When defining rules like in definition 5.1, we can always check whether there is a rule which overwrites the actual one. And if there are several rules which match a given instance graph, we can exclude those rules which have been overwritten by another rule which also matches. But as we have a new rule definition, we also need a new definition for matching.

Beforehand we need to define a descendant of a rule and the overwriting clan $clan_{O}$.

Definition 5.2. Descendant

Given a set of rules $r_0$, ..., $r_{n-1}$, $r_n \in r_{DMMInOv}$, $r_n$ is descendant of $r_0$ if a sequence $(r_i, r_{i+1})$ exists with $i = 0, ..., n-1$ and $r_{i+1} \in overwriting_{r_i}$.

A descendant of a rule $r_0$ is therefore a rule $r_n$ which overwrites $r_0$ in one or more steps. We do not want $r_0$ to be a descendant of itself.

Definition 5.3. Overwriting Clan

For a rule $r \in r_{DMMInOv}$ the overwriting clan is defined by

\[clan_{O}(r) = \{r' \in r_{DMMInOv} \mid r' \text{ is descendant of } r\}\]

We can see that the overwriting clan of a rule consists of all its descendants. Now that we have this definition we can redefine matching for our new rule.

Definition 5.4. Matching of a rule

For a rule $r = (L, R, NAC, inv, Params, name, overwriting)$ a matching is an injective morphism $mt$ with

\[
\begin{align*}
mt: & L \to G, L \in G_{DMMR}, G \in G_{DMMI}, \\
\forall <ln, gn> & \in mt_N: type_N(gn) \in clan_I(type_N(ln)), \text{ and}
\end{align*}
\]
5.2 Relationship between overwriting and overwritten rule

In this section we want to clarify the relation between overwriting and overwritten rule.

First we will give a notation for the overwriting and overwritten rule. We will use the superscript \textit{sub} for the overwriting rule as its behaviour belongs to the subordinate type and we will use the superscript \textit{SUP} for the overwritten rule as it belongs to the superordinate type. To make it easier to distinguish between those two, we write \textit{SUP} in capital letters. Another notation we will use is \textit{context node} for referring to \textit{params}[1].

In the following we will first formalise the relation between the signatures of overwriting and overwritten rule, afterwards the left hand sides and finally the right hand sides of these rules.

5.2.1 Signature

As we pointed out earlier the signature must be the same in both rules except for the context node.

Definition 5.5. Overwriting Constraint for Signatures

Let \textit{SUP} and \textit{sub} be rules in \textit{rDMMinOv} whose signatures each have \textit{n} parameters. Concerning the signatures a necessary requirement for \textit{sub} overwriting \textit{SUP} is the following:

\begin{align*}
\text{name}_{\text{sup}} &= \text{name}_{\text{SUP}}, \\
\text{type}(\text{context node}^{\text{sub}}) &\in \text{clan}_1(\text{type}(\text{context node}^{\text{SUP}})), \\
\forall i = 2, \ldots, n: \text{type}(\text{params}[i]^{\text{sub}}) &= \text{type}(\text{params}[i]^{\text{SUP}})
\end{align*}

This property is called overwriting constraint for signatures.
We can see that both rules must have the same name and that the type of context node of the overwriting rule must be in the inheritance clan of context node’s type of the overwritten rule. In addition the other parameters in both rules must be of the same type.

5.2.2 Precondition

As we know the left hand side of a rule $L_{SUP}$ is part of the precondition. But NACs are also part of the precondition as they are an extension of a rule’s left hand side. Our definition 5.1 takes NACs into account and when we define the constraints for preconditions we must remember NACs as well. But $L_{SUP}$ and NAC can be treated likewise.

**Definition 5.6. Overwriting Constraint for Preconditions**

Let $SUP$ and $sub$ be rules $\in r_{DMIR_{MO}}$. Concerning the preconditions a necessary requirement for sub overwriting $SUP$ is the following:

There is an injective morphism (matching) $mt$ between the left hand side of the overwritten rule $L^{SUP}$ and the left hand side of the overwriting rule $L^{sub}$ with

$$mt : L^{SUP} \rightarrow L^{sub},$$

$L^{SUP}, L^{sub} \in G_{DMMR}$ and

$$\forall <l^{SUP}_n, l^{sub}_n> \in mt : type_N(t^{sub}_n) \in clan_I(type_N(t^{SUP}_n))$$

There is an injective morphism (matching) $mt'$ between the NACs of the overwritten rule $NAC^{SUP}$ and the NACs of the overwriting rule $NAC^{sub}$ with

$$mt' : NAC^{SUP} \rightarrow NAC^{sub},$$

$NAC^{SUP} = \{\hat{L}\}, NAC^{sub} = \{\hat{L}'\}$

$\hat{L}, \hat{L}' \in G_{DMMR},$

$$\forall <\hat{l}^{SUP}_n, \hat{l}^{sub}_n> \in mt'_N : type_N(t^{sub}_n) \in clan_I(type_N(t^{SUP}_n))$$

and

$NAC^{SUP} \subseteq NAC^{sub}$

This property is called overwriting constraint for preconditions.

When we look at the definition above, we can see that the left hand side of the overwritten rule must be part of the left hand side of the overwriting rule. This means that the new left hand side must contain all elements of the original one and can also contain some additional elements. In the last chapter we figured that it should be possible to strengthen preconditions, therefore NACs are handled in the same way as the left hand side. All NACs which were in the overwritten rule must also be in the overwriting one, but there may be some additional NACs as well. With additional NACs the precondition is stronger than before. With less NACs it would be weaker.
5.2.3 Postcondition

To examine the relationship between an overwriting and an overwritten rule, we do not only need to compare signatures and precondition of the two rules but also the postconditions. As we pointed out in the last chapter, we want to compare the postconditions of the two rules in a form where no invocations occur. To make this possible we need to melt the invoking rules and the invoked rules together.

In the following we will first formalise how to combine or "glue" invoking and invoked rule on the basis of Hausmann's combination of premise rule and rule schema which he calls unfolding (cp.[Hau05] p.98 et seq.). Afterwards we will be able to compare the right hand sides of the overwriting and the overwritten rule.

Before we can eliminate all invocations, we need to formalise the eliminating of a single invocation. The notation $|\text{matches}|$ indicates that all nodes of $r_{post}$ are renamed if need be according to the matching matches. The names of nodes in $r$ are adopted by the expanded rule $exp_{post}$.

**Definition 5.7. Invocation expansion**

The expansion of a rule $r \in r_{DMMInOv}$ by a small-step rule $r_{post} \in r_{DMMInOv}$, invoked by the invocation $i_{post}$ is the rule $exp_{post}$ with

$exp_{post}(r, i_{post}, r_{post}) = (L', R', NAC', inv', Params', name', overwriting')$ and matches $(i_{post}, r_{post})$ an invocation-rule matching as described in definition 2.21 such that

$L' = L^r \cup L^r_{post}|\text{matches}$

$R' = R^r \cup R^r_{post}|\text{matches}$

$NAC' = NAC^r \cup NAC^r_{post}|\text{matches}$

$inv' = (inv^r \setminus i_{post}) \cup inv^r_{post}|\text{matches}$

$Params' = Params^r$,

$name' = name^r$, and

$overwriting' = overwriting^r$

An invocation expansion combines the invoked small-step rule $r_{post}$ and the invoking rule $r$ and it produces a new rule $exp_{post}$. All the structures of invoked and invoking rule are melted whereas the passed parameters form the interface. By means of the parameters contained in the matching the union can be accomplished. We remember that a small-step rule can invoke other small-step rules. These invocations are thus transferred to the extended rule. We use $exp_{post}$ to indicate a transitive application of $exp_{post}$ until the point where no invocations are left in $exp_{post}$.

Before we can define the unfolding of invocations, we need a rule definition without invocations.
Definition 5.8. DMM Rule with Overwriting (and without Invocations)

A DMM Rule with Overwriting (and without Invocations) \( r \) is a six-tuple with \( r \in r_{DMMOv} \) and
\[
r_{DMMOv} = \{ < L, R, NAC, Params, name, overwriting > \}, \text{ with}
\]
- \( L, R \in G_{DMMR} \) ordinary rule graphs,
- \( NAC = \{ L \} \subseteq G_{DMMR} \) a set of negative application conditions,
- \( Params \subseteq L, \forall p \in (Params \cup \{ x \mid x \in params^{inv} \}); l_N(p) \neq \bullet \),
- \( name \in \Lambda \setminus \bullet \),
- \( overwriting: \) is a set of DMM rules which are directly overwritten by \( r_{DMMOv} \).

We see that this definition is exactly like definition 5.1 except that no invocations are allowed and therefore not stored.

Definition 5.9. Unfolding of Invocations

The unfolding of invocations \( ufi \subseteq r_{DMMInOv} \times r_{DMMOv} \times \mathcal{P}(r_{DMMInOv}) \) is a relation and for \(< r_1, r_2, Rules > \in ufi \) the following condition holds:
\[
r_2 = \bigcup_{i \in inv_{r_1}} \left( \bigcup_{ir \in Rules} (exp^*_post(r_1, i, ir)) \right)
\]
The unfolding of invocations is a subset of the Cartesian product of the set of rules with invocations and overwriting, the set of rules with overwriting (and without invocations), and the power set of \( r_{DMMInOv} \) which means the rule set given. The latter may yield one or more rules matching one invocation. The unfolding yields rules in \( r_{DMMOv} \) without invocations. A union over all invocation expansions is possible since the invoking rule is preserved in every new rule. The unfolding of invocations may lead to a rule set with a possibly infinite number of rules, for example if the invoked rules have been used to gain recursion.

It may look very cumbersome at first glance, if a language engineer has to melt all the invocations of a rule with the rule itself just to check if another rule can overwrite the first one. But we do not need to forget that the language engineer will in all probability not model everything by hand but use a tool. This tool should be able to help him check whether all properties between overwriting and overwritten rule are fulfilled.

Now that we have examined how we can glue the invoking rules and invoked rules together we can go on and concern ourselves further with the question about the relationship between overwriting and overwritten rule. We act on the assumption that the rules do not have any invocations left.

Definition 5.10. Overwriting Constraint for Postconditions

Let \( SUP \) and \( sub \) be rules \( \in r_{DMMOv} \). When \( sub \) overwrites \( SUP \) a necessary condition for the postconditions of the rules is the following:
\[
\text{There is an injective morphism (matching) } \text{mt between the right hand side of the overwitten rule } R_{SUP} \text{ and the right hand side of the overwriting rule } R_{sub} \text{ with } \text{mt} : R_{SUP} \rightarrow R_{sub}.
\]
\( R_{SUP}, R_{sub} \in G_{DMMR} \) and
\[
\forall < r_n^{SUP}, r_n^{sub} > \in \text{mt}_N : \text{type}_N(r_n^{sub}) \in \text{clan}_I(\text{type}_N(r_n^{SUP})).
\]
This property is called overwriting constraint for postconditions.
Relationship between overwriting and overwritten rule

The constraint for the postconditions states that the right hand side of overwriting rule $sub$ must completely contain the overwritten rule $SUP$ except that instead of a specific type in $SUP$ it is possible to use a subtype of it in $sub$. $sub$ may also contain additional elements.
Chapter 6

The GROOVE tool set

In this section we first want to introduce GROOVE, which was developed for graph transformations as basis for model transformation and operational semantics as well as for the use of simple graphs to model design-time, compile-time, and run-time structure of object-oriented systems (cp. [GroM07]). Hausmann uses the GROOVE tool set as basis for a prototype DMM interpreter in [Hau05] and gives a translation of DMM specifications into GROOVE specifications. We want to extend the translation by means for overwriting rules after giving a survey of the GROOVE tool set.

6.1 Introduction into the GROOVE Tool Set

GROOVE is the abbreviation for GRaphs for Object-Oriented VErification and is a project which deals with graph transformations as basis for model transformation and operational semantics and with the use of simple graphs to model design-time, compile-time, and run-time structure of object-oriented systems. It was developed at the University of Twente in the Netherlands by Arend Rensink: It is distributed under the Apache License, Version 2.0, developed in Java (cp. [GroM07]) and information about it is available under http://groove.sourceforge.net/groove-index.html.

6.1.1 Tools of the GROOVE tool set

The GROOVE tool set consists of five components: the Simulator, the Editor, the Generator, the Imager, and the Model Checker (cp. [GroM07]).

- Simulator: The simulator allows a step-wise investigation of generated transition systems. It has three different views, one for displaying rule graphs, one for displaying the different states, and one view for the LTS as far as it is generated.
• Editor: As the name suggests, the editor allows creating and editing graph transformation rules and graphs, or more precisely start graphs.

• Generator: The generator produces full or partial transition systems and is used by the other components.

• Imager: The imager can create images of graphs, graph transformation rules, and transitions systems. It is used by editor and simulator but is also a stand-alone tool. (cp. [Hau05])

• Model Checker: This is the newest part of the GROOVE tool set. It can verify properties over transition systems.

6.1.2 Graph Transformations in GROOVE

Graphs in GROOVE are directed graphs with labeled edges. Node labels are represented as labeled self-edges which additionally provides the possibility of multi-labeled nodes. GROOVE follows the Single-Pushout-Approach (SPO) just like DMM and it allows non-injective matches. That means for example that more than one node in the rule graph can match the same node in the instance graph. To prevent this it is possible to insert special edges called merge embargo edges. Two nodes which are connected via these edges must not match the same element in the instance graph.

The concept of Negative Application Conditions (NACs) is implemented in GROOVE. A NAC is formed by elements which are connected to each other and whose labels have the prefix \textit{not}:. Multiple NACs in a rule are possible. Universal Quantified Structures (UQS) on the other side are not supported in GROOVE.

The only control structure implemented in GROOVE is the one of priorities. The higher the priority number of a graph transformation rule the higher its application priority. This also means in particular that the DMM control structure of invocations is not supported in GROOVE (cp. [Hau05]).

6.2 Translation of the DMM Specification into a GROOVE Specification

Hausmann chooses GROOVE as basis for a prototype DMM interpreter in [Hau05], because it allows for editing graphs and graph transformation rules using a visual representation which resembles the one he uses. Additionally, with a rule set and a start state it is able to visualise the complete LTS and it can perform model checking. As the graph transformation approaches of DMM and GROOVE differ in certain points a translation between these two is necessary. Therefore we will first concentrate on the translation of graphs in this section, then we will turn our attention to the translation of rules and afterwards we deal with the encoding of application control.
6.2.1 Translation of Graphs

Graphs in GROOVE can either be rule graphs or state graphs (instance graphs in DMM). Type graphs like in DMM are not supported as typing is not designated.

Each node in a DMM graph is translated into a node of a GROOVE graph. Nodes in DMM can have a name (label), a type, and attributes. When looking at state graphs the name and the type of a DMM node are each transformed into a labelled self-edge. To distinguish these two, the name of the node receives the prefix "_". Note that the whole inheritance hierarchy of the DMM node's type is encoded in the GROOVE node's labels. Attributes in DMM are displayed as nodes and so are they in GROOVE.

In rule graphs nodes can only be labelled with their direct type only. As the encoding of the hierarchy is done in the state graphs, there is no need to encode it in the rule graph as nodes of the rule graph can match only with a node of the same type anyway.

In rule graphs we have to keep in mind that non-injective matches are generally allowed. We therefore need to explicitly disallow it by including merge embargo edges between all nodes that either have identical types or one is in the inheritance clan of the other one.

Edges in DMM graphs are each translated into edges in GROOVE graphs. Unfortunately, the threefold label structure of the edges is not supported by GROOVE. This could be rectified by using up to three labelled edges between two nodes. Hausmann prefers to fix a single label on the edges in DMM type graphs (cp. [Hau05]).

6.2.2 Translation of Rules

Some notations in DMM rules and in GROOVE rules are quite alike. Elements of $r_{\text{del}}$ are marked with $\text{del:}$, elements of $r_{\text{new}}$ are marked with $\text{new:}$, and elements of NACs are marked with $\text{not:}$. These elements have different colours in the GROOVE simulator: elements to be deleted are displayed with dashed blue lines, new elements are displayed with solid green lines, and elements in NACs are displayed with dotted red lines.

The rule signature of a DMM rule becomes the file name under which the GROOVE rule is stored. Big-step rules receive the suffix "#" instead of the asterisk character "*" as "*" is an illegal character in file names under Microsoft Windows.

Universally Quantified Structures need to be unfolded to encode them into GROOVE. For this, the number of rules to which the UQS is unfolded needs to be restricted as the number of potential occurrences of UQS in unlimited.

As premise rules affect the rule's ability to match a given state graph, they are merged with their invoking rule. The invoking rule is thus extended by the
information of the premise rule and all premise rules which may be invoked transitively.

6.2.3 Translation of Application Control

As already mentioned, invocations are not supported by GROOVE. But as invocations control the DMM rule application, they need to be encoded into GROOVE. The solution Hausmann chooses is the explicit modelling of the invocation stack in the state graph.

Each state graph receives a singleton node called \textit{DMMSystem}. When there is an open invocation, an invocation node is attached to \textit{DMMSystem}. This invocation node carries the label of the rule invoked and it has edges to the context node (with the label "\textit{self}") as well as to all parameter nodes (with the labels "\textit{param}").

The last node on the stack is an invocation node called \texttt{\_bottom} which indicates an empty stack. Only when the node \texttt{\_bottom} is the first element (and therefore the only one) on the stack of \textit{DMMSystem} a big-step rule can match. This condition is encoded into the big-step rule.

A given small-step rule on the other side can only match when an adequate invocation node is the first node on the stack. The invocation node needs to have the correct name and the node connected to it by the self-edge needs to be of the right type. When the invocation is fulfilled, the invocation node will be deleted and the stack must be reordered accordingly.

In figure 6.1 we can see a state graph with an invocation stack. This stack contains one invocation called \textit{execute} which is connected to the \texttt{\_bottom} element. \textit{execute} has a context node of type \textit{A} and a parameter of type \textit{B}. A normal DMM instance graph would only contain the nodes \textit{A}, \textit{B}, \textit{C}, and \textit{D} and their connections to each other.

As there is no guarantee that all invocations are successful, there must be a mechanism to check this. For this purpose a rule \texttt{CheckInvocation} must be added to the GROOVE rule set which is applied when no other rule can match anymore although there are still invocations on the stack, that is when

![Figure 6.1: GROOVE state graph with a non-empty invocation stack](image-url)
the system is in a final but failed state. When the system reaches a final state where no open invocations are left, the state is called stable and the rule CheckInvocation can not be applied. To ensure that this rule can only match when the system attained a final state, it gets a lower priority than all the other rules. If it can be applied the reached state is failed, otherwise it is stable.

6.3 Translation of the overwriting concept

In this section we want to translate our DMM concept of overwriting into GROOVE. For this translation we see three basic possibilities which we want to introduce. We will see that only one of them is really realisable and we will use it to first translate the overwriting of small-step rules and afterwards the overwriting of big-step rules. Finally we want to compare the number of rules in the GROOVE rule set before and after we introduced our concept.

6.3.1 Ideas for the translation of the overwriting concept into GROOVE

When we think of the overwriting concept in DMM, there are three possibilities which immediately occur to us to translate the concept into GROOVE. As the notion of overwriting in DMM resembles the one in Java in certain ways, it would be nice if the conversion into a tool would resemble it as well. But then the tool had to be able to handle dynamic binding as this is how it is done in Java. GROOVE is not able to do so and to change this, the tool needed to be rewritten. Of course, this is not possible within this thesis and we have to look for another alternative.

The second possibility is to use the control mechanism supported by GROOVE. This means we use priorities to determine which rules in the hierarchy of the overwriting and overwritten rules (short: overwriting hierarchy) are allowed to match. At first this idea sounds promising, but when we look at the realisation we soon see the problems which occur.

We assume that each overwriting rule receives a priority. This priority exceeds the highest priority of all rules this first rule overwrites by at least one. This entails that the overwriting rule automatically has a higher priority than the rules which were on the same priority level as the overwritten rules, such as all the rules which overload the overwritten rule. But these rules are neither overwritten by the first rule nor should the first rule be favoured over them. We cannot adjust these priorities as the rules would be favoured over the overwritten rules should the overwriting rule not be able to match. The overwritten rules would not be applied then, even if they were able to match. But it is also problematic to keep the priorities as then the overwriting rule would be favoured over the other rules. This means should the overwriting
rule match in addition to some rules which are not overwritten and which have lower priorities, only the overwriting rule would be applied.

Therefore only the third possibility remains, which means we have to include auxiliary elements into the state and rule graphs just like it is done when translating invocations into invocation stacks. We need to add information to the rule graphs and to the state graphs so it is possible to decide which rule is to be applied.

In the state graph we need to encode a representation of all rules and their overwriting hierarchies in addition to the invocation stack.

We also need some auxiliary rules which create control nodes connected to the representations of the lowest rules in the overwriting hierarchy which are allowed to match. Other auxiliary rules must ensure that the rules of the next hierarchy level are allowed to match by connecting control elements to them when no rule of the actual level can match.

In the rule graphs we need to add the condition that they can only match, if a control node is connected to a rule's representation in the state graph. This way we can control which rules in the hierarchy are allowed to match and which are not.

In the following we will explain this idea in detail, first for small-step rules and afterwards for big-step rules.

6.3.2 Encoding overwriting for small-step rules

In the DMM specification for each rule is stored which rules it overwrites. We can go through the overwriting hierarchy by recursively checking which rule the overwritten rule overwrites itself. This is not possible in GROOVE; but nevertheless we need to know which rule overwrites another or if it does not overwrite anything. Therefore we need to explicitly store all rules and the whole hierarchy of overwriting and overwritten rules somewhere. This is done like the translation of invocations into the invocation stack; the information is encoded into the state graph. Each rule has a representation in the state graph. When a rule overwrites another rule, these two are connected by an edge with the label "overwrites" whereas the arrowhead points to the overwritten rule. This way we can explicitly store the overwriting hierarchies.

In figure 6.2 we can see a GROOVE state graph where all rules in the system are displayed together with the information about which rules overwrite other ones. We can see that there are four rules with the name doSth. The first element in the hierarchy is the root element. Root is connected to the rule which belongs to the most general context node which again is connected to the rules it is overwritten by. Each of the rules has a number to be able to distinguish them later on in the rule graphs. Rules which have a unique signature receive a number as well, although this is not necessary. But for uniformity reasons we choose to give them one. We can also encode other
things in the nodes. If, for example, there are many rules with the same signature but with different parameters, the parameters can be stored as well to make it easier to distinguish the rules. This might be more intuitive than giving numbers to the rules. In addition the types of the context nodes can be stored hence it can easily be seen which rule in a hierarchy belongs to which type. We will do this when we encode our running example at the end of this chapter. Note that we did not connect the small-step rule not in a hierarchy to a root-element as this would make the graph more obscure and confusing than it is already.

Note that the GROOVE tool set sorts the labels by itself. The order in which they appear in a node cannot be defined. Also remember that elements in rule graphs with dashed blue lines are going to be deleted, elements with solid green lines are going to be newly created, and elements with dotted red lines are NACs.

Now we found a possibility to explicitly store the overwriting hierarchies. But we still did not find a way to prevent the rules in the higher hierarchy levels to match. As we cannot change the way rules match in GROOVE, we need control elements which are connected to the representatives of those rules which are allowed to match. If they cannot, the control elements need to be deleted and rules in the next higher hierarchy level need to get permission to match. For this we need certain auxiliary rules.

In figure 6.2 we can see that the first invocation in the invocation stack is not for the rule we actually want to invoke but for such an auxiliary rule which is called `help_doSth` in this case. We could alternatively model certain NACs in this auxiliary rule as we must do when translating big-step rules as we will see in the next chapter, but this would result in a graph much more complex.

The rule `help_doSth` adds a control element connected to a representative

![Figure 6.2: GROOVE state graph](image)
Translation of the overwriting concept

of \texttt{doSth()} in the state graph. We can see the rule in figure 6.3. The control element is connected to the rule which is last in the overwriting hierarchy. It identifies the rule which is allowed to match. In this rule we can also see that the invocation of the auxiliary rule is replaced by the invocation of the rule we wanted to invoke in the first place. Of course this is not mandatory, but it helps understanding what is happening in the rule and state graphs.

![Figure 6.3: GROOVE auxiliary rule graph for the small-step rule doSth() hierarchy level 3](image)

Now we have to check whether the rule we allowed to match is actually able to match. When the rule can be applied, the invocation node is deleted from the invocation stack, and additionally the control element is deleted from the graph. If, for example, the rule in figure 6.4 tries to match our state graph, it can be applied as it only requires an $A$ in the state graph besides the auxiliary elements: the right invocation on the stack and the control node connected to its representative. The next invoked small-step rule can try to match or if the stack is empty the next big-step rule.

For the case the rule is not able to match (if for example the rule in figure 6.4 also required an $E$ besides an $A$), we need another auxiliary rule which deletes the control element connected to that rule in the state graph and which creates new control nodes for the rules which are allowed to match afterwards. To ensure that this rule is only applied when the "proper" rule is not able to match the latter needs a higher priorit y than the auxiliary rule. We can see such a rule in figure 6.5.

If a rule overwrites more than one other, every overwritten rule gets a control element when the overwriting rule failed to match. And in case one of these overwritten rules in the same hierarchy could not match, either, we need to introduce an additional element to clarify that a rule has actually been checked. We do this by replacing the control node by a node of the type \textit{processed} as can be seen in figures 6.6 and 6.7. Otherwise it is possible that the hierarchy level is changed without checking whether really none of the rules is
Translation of the overwriting concept

If all rules are checked and none could match, the rules of the next hierarchy level are allowed to match.

If on the other side more than one rule on the same hierarchy level can match, there will be a fork in the transition system. The control nodes are all deleted then.

If there are at least one rule that matches and one that does not on the same hierarchy level, we need to take care that the processed node is deleted. In case that one rule on this level was able to match, no rule on the levels above will be allowed to match.

The auxiliary rules in figures 6.8 and 6.9 show this for our example. Only if neither of them can match, we will allow the rule in the next hierarchy level to match. The latter can be seen in figure 6.10.
Translation of the overwriting concept

Figure 6.6: GROOVE auxiliary rule for the small-step rule doSth() when doSth() could not match

Figure 6.7: GROOVE auxiliary rule for the small-step rule doSth() when doSth() could not match
Translation of the overwriting concept

Figure 6.8: GROOVE auxiliary rule for the small-step rule doSth() when doSth() could match only one time

Figure 6.9: GROOVE auxiliary rule for the small-step rule doSth() when doSth() could match only one time
If a control node is finally connected to the root element, we know that the invocation failed. We then allowed every single rule in the hierarchy to match, but none of them was able to. For this case we need a rule which checks whether a root element is connected to a control element, similar to the rule CheckInvocation whose application also indicates a failed invocation hierarchy.

6.3.3 Encoding overwriting for big-step rules

The overwriting of big-step rules can be modelled similar as the overwriting of small-step rules. The representation of every big-step rule is stored in the state graphs as well as their overwriting relationships. Control elements are used to indicate which rules in an overwriting hierarchy are allowed to match. We need auxiliary rules which handle the composition of the control and processed nodes.

But there are problems that occur to us when we have a closer look on how big-step rules work. The biggest problem seems to be that they are not invoked and therefore we do not know which of them actually is supposed to match. In addition we cannot invoke an auxiliary rule to set the control elements in the way we did with small-step rules, or more precisely for one overwriting hierarchy only. The third problem apparent is that we do not want to call a rule failed, just because the control node reaches the root element. In this case the rule was just not able to match, but this is not required.

The latter is the easiest problem to solve. We just assign a different kind of root element to the overwriting hierarchy, e.g. an element called big_root. Then the rule which checks if an invocation failed cannot be applied although
the control element has reached the root element. If each hierarchy gets an own root element or if there is just one of them for all rules is of no importance. We choose the first possibility to make it easier to tell the hierarchies apart.

The other two problems we solve by introducing an auxiliary rule which matches like a big-step rule. Whenever there is no invocation except for the _bottom dummy invocation in the invocation stack, this rule can match like we are used to from big-step rules. But instead of only handling one overwriting hierarchy, it handles all overwriting hierarchies of big-step rules. Every element lowest in a hierarchy gets a control node. In figure 6.11 we can see such an auxiliary rule which creates a control element for each of the two existing big-step rule hierarchies. As we do not have invocations to prevent this first auxiliary rule to match again while we check the rules in the hierarchy we have to add several NACs as can also be seen in figure 6.11. Otherwise this rule might be still able to match when we actually wanted other rules to do so and the rules’ representaties in the lowest hierarchy level would get more and more control nodes connected to them.

With these two differences to small-step rules in mind we will have a closer look on how overwriting of big-step rules is realised in GROOVE. Big-step rules which are in an overwriting hierarchy can still only match if the dummy invocation _bottom is the only element in the invocation stack. But we have the additional requirement that they need to have a control element connected to their representation in the state graph.

When encoding big-step rules we need to give every big-step rule a control element, even when it is not in a hierarchy. This is unlike the encoding of small-step rules where rules which are neither overwriting nor overwritten do not necessarily need such an element. We need to take this precaution as otherwise these rules can match while we actually check a certain hierarchy as they have a higher priority than most of the auxiliary rules. When we add the control elements they can only match if they have such an element. As they are deleted when we start checking a certain hierarchy, they cannot match accidentally. As these rules get a control node, they need to be connected to a big-root element as well, thus it is possible to check whether the rule cannot be applied.

Of course we have to make sure that the control elements for the other hierarchies are deleted when the first rule in a hierarchy tries to match the state graph. Otherwise the application of the rules of two or more hierarchies might get mixed up.

The rest of the encoding nearly works exactly as the encoding for small-step rules. If a big-step rule in the hierarchy matches, the control node is deleted. If it cannot match and it is the only rule on this hierarchy level, an additional auxiliary rule is applied which deletes the token and creates new ones for the rules the first one overwrites. If there are other rules on the same level, we need to replace the control node by a processed node. Then we need additional rules which delete this node when any other rule on the same level was able to
match, just as we need when handling small-step rules. If all rules are marked with a \textit{processed} node, the rules on the next hierarchy level get permission to match.

When the \textit{control} node is attached to the \textit{big\_root} element we know that no big-step rule in this hierarchy was able to match. Then other big-step rules can try to match, but not the one we just tried, as it still has the \textit{control} node connected to \textit{big\_root} element. This rule might only be able to match when the state graph changed and this is only possible when another big-step rule was able to match. Therefore we need to make sure that if the state graph indeed changed, the rules of this first hierarchy are again allowed to match. We then must delete the \textit{control} node. We suggest we do this by adding an invocation to each big-step rule which systematically deletes all of these \textit{control} nodes in other hierarchies. This invocation calls one of two auxiliary rules which checks if a specific hierarchy has such a node connected to its \textit{big\_root} element. If it has one, the first auxiliary rule is applied which deletes the node and invokes another auxiliary rule checking the next hierarchy. If the \textit{big\_root} element is not connected to a \textit{control} node, the other rule is applied which just invokes the new rule without deleting anything. This whole process is of course unnecessary, in case that there is only one big-step rule hierarchy.
6.3.4 GROOVE Example

In this section we want to show parts of our running example which we encoded in a GROOVE rule set. We will not show every rule here as our encoding consists of 18 rules plus the start graphs. We will only show the big-step rules and the auxiliary rules. As in the last section we included graphics from the GROOVE simulator.

In figure 6.12 we can see a possible but small start graph where all rules are encoded. In the left upper corner we can see the DMM invocation stack. Below we can see the rules’ representations. On the right hand side there is the DMM start graph.

In figure 6.13 we can see the rule node.flow which is overwritten by the forknodex.flow rules.

In figures 6.14 and 6.15 the two forknodex.flow() rules which overwrite the rule node.flow are displayed.
Translation of the overwriting concept

Figure 6.13: GROOVE rule node.flow()

Figure 6.14: GROOVE rule fornode.flow()

Figure 6.15: GROOVE rule fornode.flow()
In figure 6.16 we can see the first auxiliary rule to be applied when we check, if any rule in the hierarchy can match.

![Figure 6.16: GROOVE auxiliary rule for the rule flow() in the lowest hierarchy level](image)

In figure 6.17 the auxiliary rule is shown which is applied when the corresponding forknode.flow() rule was not able to match. The new node *processed* indicates that it was checked and that it is already clear that the rule cannot match. In figure 6.18 the case for the other forknode.flow() rule is displayed.

![Figure 6.17: GROOVE auxiliary rule for the rule flow() when the rule could not match](image)

![Figure 6.18: GROOVE auxiliary rule for the rule flow() when the rule could not match](image)
Translation of the overwriting concept

In figures 6.19 and 6.20 the possibility is taken into account that the first of the rules tried from the GROOVE system has matched, but the second has not.

Figure 6.19: GROOVE auxiliary rule for the rule flow() when the first rule matched and the second did not

Figure 6.20: GROOVE auxiliary rule for the rule flow() when the first rule matched and the second did not

In figure 6.21 a control node for the next hierarchy level is created as none of the forknode.flow rules could match. Finally figure 6.22 shows the rule for the case that none of the flow() rules could match.

Figure 6.21: GROOVE auxiliary rule for the rule flow() when none of the forknode.flow() rules matched

Figure 6.22: GROOVE auxiliary rule for the rule flow() when none of the flow() rules matched
If there were more big-step rule hierarchies we would need to make sure that the control node would be deleted from the flow()-hierarchy whenever a big-step rule of another hierarchy matched. In this case the state graph might have changed and one of the rules in the flow()-hierarchy which could not match before might be able to match with the changed state graph.

6.3.5 Increase of rule numbers

We suggested using a number of auxiliary rules. These rules are not necessary, if we do not use overwriting in DMM and in GROOVE. We now want to look how the number of rules increased when we use overwriting compared to the case we do not. We assume that the number of non-auxiliary rules is the same in both cases.

We first have to define certain values before we can do this:

- number of small-step rule hierarchies: \( h_S \)
- number of big-step rule hierarchies: \( h_B \)
- number of all hierarchies in the rule set: \( h \), with \( h = h_S + h_B \)
- maximum number of rules in a hierarchy: \( n \)
- maximum number of rules in a hierarchy level: \( e \)
- maximum number of hierarchy levels in a hierarchy: \( m \)

We can clearly see that the number of rules must be \( h \cdot n \), if we do not overwrite any rules. We assume then of course that even rules which are not overwriting or overwritten are in a one-rule hierarchy. Earlier we did not call this a hierarchy when the rules were small-step rules, but to find out the increase of rules this perception eases our work.

Now we can think about how many auxiliary rules we need. We assume the worst case that there is not only one rule per hierarchy level, but several. This increases the number of additional rules. When each rule is only overwritten by one other rule, we do not need so many auxiliary rules as we do not need to introduce processed nodes.

1. for each overwriting hierarchy of small-step rules we need a rule which creates and connects control nodes to the rules’ representatives in the lowest hierarchy level, altogether \( h_S \) auxiliary rules
2. one rule for all big-step rule hierarchies altogether to set the control nodes at the right place
3. for each big-step rule hierarchy we need a rule to delete the control nodes of all other hierarchies, altogether \( h_B \) rules
4. one rule for checking if an invocation failed (when a control node is connected to the root element)

5. delete control nodes connected to big_root elements, 2 for each hierarchy to check if a control node is there or not, altogether $2 \cdot h_B$

6. one rule for each change of the hierarchy level, altogether $m \cdot h$

7. for each rule which is supposed to match we need one which matches when the actual rule cannot, altogether $n \cdot h$

8. if there are several rules on a hierarchy level we have to make sure that a processed node is deleted when another rule of the hierarchy level matched, hence we need to check for every rule on this level if any other rule has already matched. We need an auxiliary rule for each of this rule as we cannot model this with NACs all in one rule. So we have $e \cdot (e - 1)$ auxiliary rules per level, $e \cdot (e - 1) \cdot m$ per hierarchy and altogether $e \cdot (e - 1) \cdot m \cdot h$

9. besides these auxiliary rules we have $h \cdot n$ non-auxiliary rules

In the following we will consider the following inequalities:

$$n \geq e, n \geq m, \text{ and } h \geq h_B$$

The first one indicates that the number of rules in a hierarchy is greater or equal than the number of rules in one of the hierarchy levels as there can be rules on other levels as well. The second inequalities indicates that the number of hierarchy levels cannot be greater than the number of rules in the hierarchy. It is equal when there is only one rule on each hierarchy level. The last inequalities is obvious as $h = h_B + h_S$. That means that the number of hierarchies is equal to the combined numbers of small-step and big-step rule hierarchies.

Keeping all this in mind we get the following formula:

$$\begin{align*}
(h + 2) + 2 \cdot h_B + m \cdot h + n \cdot h + e \cdot (e - 1) \cdot m \cdot h + n \cdot h \\
\leq 2 + (3 \cdot h) + 3 \cdot n \cdot h + n^4 \cdot h \\
= O(n^4 \cdot h)
\end{align*}$$

This seems to be a huge increase of rules compared to the rule set without overwriting. But we also have to see that $n$ will probably not be very big. $n$ is the maximum number of rules in a hierarchy. In a given rule set most rules will probably not be overwritten at all and within a hierarchy there will only be a limited amount of rules, not all hierarchies will contain $n$ rules. As the overwriting hierarchy levels symbolise the number of inheritance levels in a DMM type graph we can also add that the number of levels is probably not that big either.
When we look at our running example from the last section and section 4.3.3 we will see that we have 11 non-auxiliary rules. We could either say that the rules which overload each other belong to one hierarchy, then we have 5 hierarchies, or they each are in an own hierarchy, then we can count 9 hierarchies. \( n \) is three here as the only hierarchy where rules actually are overwritten is the one of \texttt{flow()} and there we have one \texttt{node.flow()} rule and two \texttt{forknode.flow()} rules. So using our formula, we would either receive \( 81 \cdot 9 = 729 \) or when we use the smaller number \( 81 \cdot 5 = 405 \). That is both much bigger than the actual number of rules which is 18. In our case the number of rules has not even doubled.
Chapter 7

Summary and Outlook

In this chapter we will briefly summarise the results of this thesis and afterwards we will give an outlook on further work possible in the area of overwriting in Dynamic Meta Modelling.

7.1 Summary

Visual Modelling Languages are often used to model certain aspects, e.g. in software development processes, as their graphical notation is relatively easy to comprehend. These VMLs need to be defined precisely and intuitively for the intended target audience and their definitions must be easy to understand as well. Therefore we need a Meta modelling approach which has all three of these qualities.

Dynamic Meta Modelling is a Meta modelling approach by Hausmann which is based on graph transformation and is therefore easily comprehensible. It can define the static semantics of a VML as well as the dynamic semantics precisely. For the intended target audience which is supposed to be accustomed to object-orientation it might not necessarily be intuitive though. Inheritance can be used in DMM, but polymorphism is only partly implemented. Or more precisely: the overloading of graph transformation rules is possible, overwriting is not. In this thesis we developed an approach where rules belonging to a certain type can be overwritten for types inheriting from this first type. This should make the DMM approach more intuitive.

To give a basis to understand DMM we gave an overview of graph transformation and the different graphs it requires in the chapter 2. Afterwards we introduced the concepts of DMM and its formalisations. We also presented an example including a rule which was to be overwritten later in the thesis.

As the target audience is accustomed to object-orientation we stressed the importance of inheritance and polymorphism in object-orientation and therefore in DMM in chapter 3. We then dwelled on what we understand by inheritance and polymorphism.
We investigated different possibilities how to define overwriting in DMM in chapter 4. We concluded that the signature of the overwriting and the overwritten rule must be the same and that left and right hand sides of the overwriting rule must be extensions of the overwritten rule. To check this it is necessary to unfold all invocations. We also found that the overwriting in DMM must be explicitly unlike in programming languages like Java where overwriting is implicitly. We also investigated the transition system to look if there is need of any other restrictions, which is not the case.

In chapter 5 we then formalised the findings of chapter 4 to make them more precisely and easier to use for possible tool support. We redefined DMM rules and the notion of matching and gave definitions for the unfolding of invocations as well as for the relationship between overwriting and overwritten rules.

In chapter 6 we introduced the tool set GROOVE which was used by Hausmann as a basis for a DMM interpreter. We encoded the overwriting of rules in GROOVE by introducing auxiliary rules and structures within rule and state graphs. We distinguished between the overwriting of big-step rules and small-step rules. Finally, we translated our DMM example into a GROOVE rule set.

7.2 Outlook on further work

There are some improvements that can be made concerning the tool support of DMM and the concept of overwriting within DMM. In the following we will name some of prospects of future work that could be done.

First, one could investigate, if the defined constraints for overwriting prove of value. The question is whether it is the intended mixture of flexibility to gain expressiveness and of restriction. For this an evaluation method would be needed.

When we have a look at the GROOVE tool support there is especially room for improvement. The encoding of DMM graphs into GROOVE graphs can be enhanced. It would for example be nice, if the auxiliary rules would not be displayed in the transition system as it bloats with these rules. Maybe it is also possible to find a better way to translate a DMM specification into a GROOVE specification to reduce the number of necessary auxiliary rules. This might then improve the performance of the tool.

The encoding of DMM rules in GROOVE was done manually in this thesis. Something that would make modelling of overwriting much easier is the automatic generation of the auxiliary rules. During our work we found it very difficult to debug these rules in GROOVE and the generation would be time saving. It would be even better when a DMM specification could be translated automatically into a specification for GROOVE or some other tool.

It would also be nice, if the representatives of the rules and their hierarchy
Outlook on further work

could be automatically generated into the start graph. This should not be that
difficult if the tool knew which rule is to overwrite another rule.

As the constraints on overwriting might make the whole process rather
complex when the rule to overwrite includes several invocations, the possibility
to actually let the computer check if the constraints are fulfilled would be very
helpful. The ideal solution would include that the constraints are checked
while creating the overwriting rule and not only when it is finished.
Bibliography


BIBLIOGRAPHY


[RT05] A. Rensink and G. Taentzer. Ensuring Structural Constraints in Graph-Based Models with Type Inheritance. In Fundamental Approaches to Software Engineering (FASE), Pages 64-79, Lecture Notes in Computer Science 3442. Springer-Verlag, Edinburgh, Great Britain, 2005