Enhancing the Dynamic Meta Modeling Formalism and its Eclipse-based Tool Support with Attributes

Bachelor Thesis

Eduard Bauer

edebauer{at}mail.upb.de
6383047

Paderborn, October 2008

Supervisor: Prof. Dr. Gregor Engels
Co-Supervisor: Christian Soltenborn

UNIVERSITÄT PADERBORN
Die Universität der Informationsgesellschaft

Table of Contents

1 Introduction........1
1.1 Current State........2
1.2 Goals of this Bachelor Thesis........2
2 Dynamic Meta Modeling........4
2.1 Graph Transformations........4
2.2 A Running Example........5
2.3 Design of Dynamic Meta Modeling........5
2.3.1 Runtime Metamodel........6
2.3.2 DMM Ruleset........7
2.4 Further Usages of DMM........12
3 Extension of DMM........13
3.1 Formal Extension of DMM Rules........13
3.1.1 Introduction of an Expression Language........19
3.1.1.1 Lexical Tokens........20
3.1.1.2 Possible Expressions........22
3.1.1.3 Well-formed Expressions........27
3.1.2 Extension of the DMM Metamodel........32
3.1.2.1 Conditions and Assignments........32
3.1.2.2 Expression Package........33
3.2 Visual Extension of DMM Rules........35
4 Implementation of the Expression Language........37
4.1 Introduction to JavaCC........37
4.2 Implementation........39
4.2.1 Regular Expressions........40
4.2.2 Grammar........41
4.2.3 Well-formedness Rules........43
4.2.3.1 Overview........44
4.2.3.2 Result Package........44
4.2.3.3 Constraints Package........46
4.2.3.4 Validator Package........50
5 Enhancement of the Graphical Editor........52
5.1 Introduction to Eclipse........52
5.1.1 Eclipse Modeling Framework........53
5.1.2 Graphical Modeling Framework........56
5.2 Implementation........59
5.2.1 Access to the Validation Results........59
5.2.2 Creation of an Unparser........60
5.2.3 Addition of Conditions and Assignments........62
5.2.4 Access to the Expression Parser and Unparser........64
5.2.5 Keeping Conditions and Assignments up to date........65
6 Extension of the Transformation to Groove Rules........67
6.1 Introduction to Groove........67
6.1.1 Some Groove Constructs........68
6.2 Comparison between Groove and DMM........71

6.3 Current State of the Transformation....72
6.3.1 Preliminary Actions....73
6.3.2 Transformation of basic constructs....74
6.3.3 Transformation of quantified nodes....75
6.3.4 Transformation of Invocations....76
6.3.4.1 DMM Invocation Stack....77
6.3.4.2 Invocations performed on non-quantified nodes....78
6.3.4.3 Invocations performed on quantified nodes....80
6.4 Implementation of the Augmentation....81
6.4.1 Transformation of Expressions....82
6.4.2 Transformation of Conditions....83
6.4.3 Transformation of Assignments....86
6.4.4 Additional Treatments....87
6.4.4.1 Handling Nodes with the Role “create”....87
6.4.4.2 Handling Nodes with the Role “not exists”....88
6.4.4.3 Handling Quantified Nodes....90
7 Conclusion and Outlook....92
7.1 Conclusion....92
7.2 Outlook....92
List of Figures....94
List of Tables....95
List of Code extracts....95
List of Transformation directives....96
Bibliography....97

1 Introduction

Today models are often used in order to describe complex situations or difficult problems. The use of models increases the comprehension and reduces the time needed to grasp these situations or problems. These models are based on some concepts describing how models have to be composed. Similar to sentences that are composed in a natural language, models are said to be composed in a modeling language.

On the whole it is possible to distinguish between two kinds of modeling languages. Structural languages comprising models for describing structures and behavioral languages covering models used to describe the behavior and the sequence of actions that are performed in a particular situation.

Just like in natural languages, the usage and the meaning of a modeling language have to be explained. Such specifications address two parts of modeling languages–their syntax and their semantics [1].

The syntax of a language describes which constructs can be used to compose models in a modeling language. These constructs are also called syntactical elements. The description of the syntax is often done with the help of a metamodel. A metamodel itself is a model–which can be presented by a UML class diagram [2]–that describes all the possible models composed in a modeling language. Thus, a particular model is an instance of its metamodel.

Sometimes all the constraints defined within a metamodel–like for example cardinalities–do not suffice in order to limit the possible instances of the metamodel to all the well-formed models that can be composed in the modeling language. In this case the definition of well-formedness rules is required. These are constraints that can only be checked by considering the semantics–the meaning, this is described below–for particular syntactical elements. This kind of constraints can be given in a special language like for example OCL [3] in order to specify the amount of well-formed models properly. The well-formedness rules limit the possible instances of a given metamodel to the well-formed ones.

The visual representation of the constructs is usually given by a special mapping between the classes that are defined in the metamodel and geometric figures that consist of lines and closed elements–for example arrows and boxes.

The semantics describe the meaning of constructs that are used within a model. Semantics are usually specified by a mapping between the modeling language, containing all the possible models, and the semantic domain. This mapping is called the semantic mapping. The semantic domain is usually an already formalized notation where the meaning of all its constructs is unambiguously defined. Thus, a semantic mapping attaches a meaning to each syntactical element used within a modeling language.

Because structural languages describe the structure of situations and problems, a correct semantic mapping between syntactical elements and their meaning suffices for a full specification of their semantics. Nevertheless, this is not the case for behavioral languages. Models that are used to describe behavior can be regarded as being executed while the described behavior takes place. In that case the semantic mapping has to be extended in order to describe how such models can be executed. This kind of semantics is called dynamic semantics.

Usually these semantic mappings and sometimes even the descriptions of the semantic domain are given in natural languages. But, since natural languages are not specified in a formal way, specifications in natural languages can lead to various inconsistencies and ambiguities. In order to exclude these ambiguities in the case of dynamic semantics, a special approach, called Dy-

namic Meta Modeling *(DMM)* [4], can be used.

Dynamic Meta Modeling is a technique used to formally describe the dynamic semantics of behavioral modeling languages that make use of a metamodel to define their syntax. DMM itself is a modeling language. It consists of two counterparts–one for the creation of the foundation in order to specify the dynamic semantics and a set of graph transformation rules, called DMM ruleset, to complete the specification.

The first counterpart is an enhancement of the original metamodel of the behavioral modeling language. It introduces elements–additional pieces of information–for the description of runtime aspects. That is why such an extended metamodel is called runtime metamodel or model of the semantic domain.

The second counterpart describes how instances of the runtime metamodel change in time and thus attaches a meaning to the different constructs of these instances. These changes are specified by a set of graph transformation rules, called DMM ruleset. This can be done since instances of metamodels can be interpreted as graphs; accordingly, changes of these instances as graph transformations.

A profound introduction to graph transformation and DMM is given in chapter 2. At this place the reader should only have an abstract view of DMM in order to understand the current state [4] that builds the foundation of this bachelor project and the goals that shall be accomplished by this work.

1.1 Current State

This bachelor project addresses DMM and its possibility to describe changes of instances of runtime metamodels by the use of graph transformation rules. In a recent bachelor thesis [5] a graphical editor was developed with the help of the Graphical Modeling Framework *(GMF)* [6] to visually model DMM rulesets. Besides this, there currently exists a possibility to transform these DMM rulesets to another representation of graph transformation rulesets which can be processed by a special graph transformation framework called Groove [7]. The necessity to transform the DMM rulesets results from the fact that DMM does not process the specified graph transformation of its own, but uses Groove to perform these graph transformations.

This chain of tools used together to specify the dynamic semantics of behavioral languages are currently not able to describe all the parts needed to specify the dynamic semantics of complex languages fully, easily, and intuitively. There are cases where metamodels of modeling languages contain attributes for the specification of all the possible models. In these cases the values of attributes can change while the described behavior takes place and therefore these changes have to be specified for instances of the according runtime metamodel.

But for now, there is only a brief concept of attributes within DMM and above all there is no possibility to specify the changes of the values of such attributes. Therefore, the dynamic semantics of modeling languages whose metamodel makes use of attributes can not be specified with the help of DMM and its currently available tools.

1.2 Goals of this Bachelor Thesis

Because of this lack of support for attributes, the idea comes up to extend DMM rulesets and the currently available tools by the introduction of attributes and according concepts, enabling the possibility to specify the changes of instances of runtime metamodels that contain attributes. In detail, two different constructs concerning the usage of attributes can be defined and introduced

into DMM rules that affect the dynamic semantics of modeling languages.

The first construct is called condition. Conditions have to be fulfilled such that a specific DMM graph transformation rule matches[1]. A condition consists of an expression defined over attributes and literals. This expression must evaluate to a `boolean` value. If, besides the normal matching constraints, all conditions defined within a DMM rule are evaluated to the value `true` the DMM rule matches and can therefore be applied.

The second construct states a possibility to change the value of an attribute. This construct is called assignment. Assignments specify new values of attributes and assign these new values to attributes if the DMM rule is applied. Such an application can update the value of many different attributes at the same time. An assignment itself consists of an attribute to which the new value is assigned and an expression that specifies how the new value is calculated. The latter part is defined over attributes and literals.

Now the overall goal of this bachelor project can be formulated precisely as follows: First of all, the two parts, conditions and assignments, have to be introduced to DMM such that DMM rulesets can make statements about the values of attributes defined within runtime metamodels. Secondly, the currently available graphical editor has to be enhanced in order to provide the possibility to enter conditions and assignments. Finally, the transformation of DMM rules to Groove rules has to be extended in order to incorporate these new constructs into the transformation process.

Similarly to the sequence of elaborations that are performed during this bachelor project, this document contains the following sequence of chapters. First of all, chapter 2 introduces graph transformations and DMM profoundly. Thereafter chapter 3 contains all the pieces of information concerning the specification of expressions that can be used within conditions and assignments. The next chapter, chapter 4, covers topics about the creation of a parser that accepts all the expressions defined in the third chapter and converts these expressions to an abstract, processable representation. Thereafter the extensions to the currently available graphical editor are described in chapter 5. Chapter 6 contains the augmentation of the transformation from the DMM rulesets to a set of rules that can be processed by the graph transformation framework Groove. Finally, chapter 7 sums up the elaborations of this bachelor project and gives a brief overview over further possible extensions to Dynamic Meta Modeling.

Chapters 4, 5 and 6 contain a brief introduction into the frameworks used in order to achieve the tasks described above.

1 The concepts of matching and other familiar graph transformation concepts will be depicted in detail in subsection 2.1.

2 Dynamic Meta Modeling

This chapter of the bachelor thesis at hand introduces Dynamic Meta Modeling profoundly. But, first of all, before starting the introduction to Dynamic Meta Modeling two aspects have to be introduced and explained to the reader. Because DMM builds heavily on the concepts of graph transformation–as one counterpart of DMM consists of graph transformation rules–section 2.1 contains some aspects of the topic of graph transformations. Thereafter, a running example is introduced that is used throughout the whole bachelor thesis. This example is given in section 2.2. It is used many times to describe the design of Dynamic Meta Modeling and other familiar concepts vividly. The actual description of Dynamic Meta Modeling is given in section 2.3.

2.1 Graph Transformations

The overall goal of graph transformations is, like implied by the name, the specification of changes within graphs. Graph transformations are performed on graphs that contain nodes and edges in between the nodes. These graphs are called host graphs. The description of a graph transformation is given by a graph transformation rule.

In general, graph transformation rules consist of two sides–a left-hand side and a right-hand side. The left-hand side of a graph transformation rule is a structure that is mapped into the host graph. This means that the structure defined in the left-hand side is searched within the host graph. If this structure is found, the graph transformation rule is said to match the host graph. In the case of such a matching the graph transformation rule can be applied to the host graph. How the graph changes on application is defined on the right-hand side of the graph transformation rule. Therefore, it specifies the structure that replaces the structure of the left-hand side on the application of the rule.

A matching is technically achieved by the mapping of all the nodes and edges used within the left-hand side of the graph transformation rule to nodes and edges present within the host graph. This is called a morphism between the structure defined on the left-hand side of the graph transformation rule and the host graph.

Besides the definition of structures that must be present within the host graph, the left-hand side of a graph transformation rule can specify the absence of a certain structure. Such a structure is called a negative application condition. It limits the matching by the specification which structures are not allowed to be present within the host graph.

Different graph transformation rules together are called a ruleset, or a set of rules. There might be situations where more than one rule of the ruleset matches the host graph, of course. Then, the application of all these different rules can lead to different successive host graphs. All these host graphs and all the applications of rules together can accordingly form a transition system. Transition systems therefore consist of different states of the host graph that is being transformed and transitions–that are created by the application of a graph transformation rule–between these states. Thus, a transition system is a graph itself. Different host graph are represented by different vertexes and transitions–applications of graph transformation rules–are represented by edges within this graph. The name of an edge is always the name of the graph transformation rule that has been applied in order to create the new host graph.

Since a host graph can be seen as an object structure resulting from object oriented programming, different concept of this domain can be introduced to graph transformations. One concept is the definition of types for the different nodes. Therefore, a special type graph is used that states all the possible types and the relation between these types. This type graph can be seen as a

kind of metamodel that states how all the different models, namely host graphs, have to look like. As the host graphs are then typed over the type graph, they are accordingly called typed graphs. Since the corresponding graph transformation rules make statements about these typed graphs, they have to be typed over the same type graph, as well.

There are different efforts to add attributes and other related concepts to type graphs and accordingly to typed graphs [8]. Attributes can be added to the different types defined within the type graph. Thus, nodes having that type would have particular values for these attributes.

2.2 A Running Example

Throughout the whole bachelor thesis the following running example will be used to explain relations and facts vividly and comprehensibly. The example is a modeling language that should be pretty familiar to the average computer scientist. This is the modeling language of petri nets [9]. This modeling language is a behavioral language for specifying situations like traffic light indicators or workflow descriptions and thus can be perfectly used for the exemplary definition of the dynamic semantics of this language.

A petri net is a directed graph consisting of vertexes connected by arcs and arbitrary many tokens. There are two different kinds of vertexes that are called transitions and places. A place is a storage point for tokens. Places themselves are only allowed to be connected to transitions. Transitions are vertexes that consume and produce tokens if they fire. Thus, a firing transition can be seen of as a possibility to reorder the tokens in some places. Similarly to places, transitions are only allowed to be connected to places. This leads to a bipartite directed graph.

A transition can only fire if every place that has an outgoing arc to the transition carries at least one token. These places are called input places. While the transition fires it consumes one token from each such place and produces a token for each place that has an incoming arc from the transition itself. Such places are accordingly called output places. Within this bachelor thesis, it is assumed that the set of input place and output places of each transition is disjoint[2].

A deployment of tokens over the different places is called a marking. Thus, when a transition fires it changes the marking of the petri net, resulting in a new deployment of the tokens.

Simple petri nets can be extended by the introduction of capacities for the places and weights for the arcs. Thus, the capacity constraint of each place has to be obeyed at every point in time. The weights determine how many tokens must exist on each input place of a transition such that the transition can fire, and how many tokens are added to each output place when the transition fires.

2.3 Design of Dynamic Meta Modeling

As DMM [4] was developed based on different object orientation concepts it takes several approaches of the traditional graph transformation and reuses them in the context of object orientation.

As already mentioned, DMM achieves the specification of the dynamic semantics of modeling languages by two different counterparts. The first one consists of the runtime metamodel that is based upon a metamodel for the description of the syntax of the modeling language. The second counterpart consists of a set of graph transformation rules to complete the specification. Such a set of graph transformation rules is called a DMM ruleset in the domain of Dynamic Meta Modeling. These two parts are described in detail in the following two subsections 2.3.1 and 2.3.2.

2 By this, the following examples can be kept more simple since many special cases for firing transitions can be avoided.

On the whole, DMM achieves the specification of the dynamic semantics of a behavioral modeling language by precisely specifying the changes of an instance of the runtime metamodel, thus stating what is changed and in which way it is changed, if some particular action of the described behavior takes places. This can, for example, be a firing transition in case of petri nets that changes the marking of the petri net.

Due to the fact that DMM is used to describe the dynamic semantics of a modeling language in a formal and precise way, DMM as a modeling language itself is defined formally with the help of a metamodel and a set oriented formalism.

2.3.1 Runtime Metamodel

The first counterpart extends or even replaces the metamodel that is used to describe the syntax of a modeling language. Thus a further metamodel, called runtime metamodel, is introduced. In order to create a connection between the original metamodel and this runtime metamodel a mapping between these two kinds of metamodels is used. The runtime metamodel contains some additional pieces of information regarding the state of a model at the time when the described behavior takes place and the model can somehow be seen as executed. Accordingly, as already defined, this is the reason for the name of this kind of metamodel. Besides this term, the term model of the semantic domain is used to describe the extended or replaced metamodel.

Figure 1 gives a rough overview over the Dynamic Meta Modeling, its counterparts and their relationships. The different relationships are depicted by associations within this diagram. All the constructs and terms used within this figure are going to be described in the following paragraphs. Some parts of the figure, like nodes and edges, will be explained in the following subsection 2.3.2.

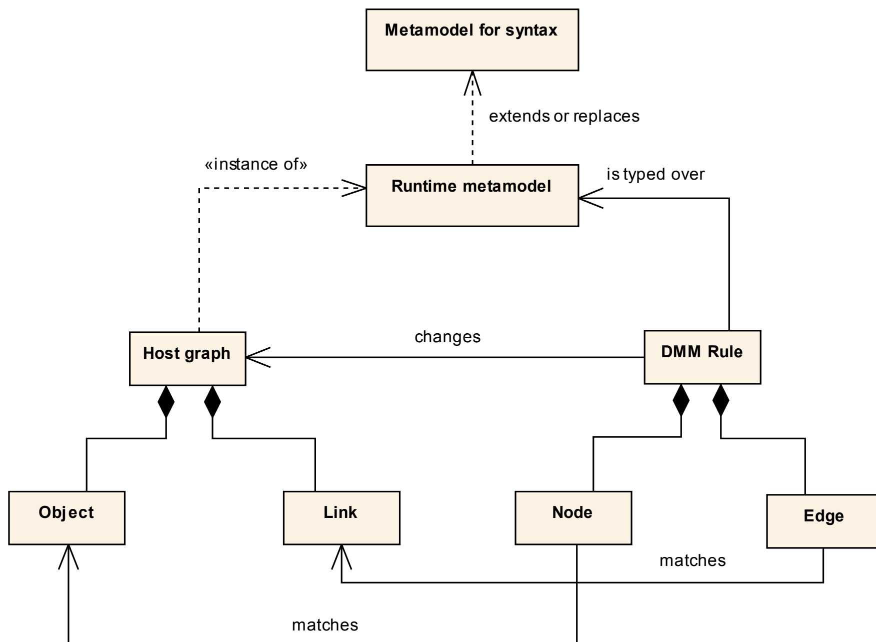

Figure 1: Relationship between the different DMM and related elements

The runtime metamodel can be seen as a type graph as it was defined in the domain of graph transformation in section 2.1. Instances of this metamodel can accordingly be compared to host graphs or respectively typed graphs. Because DMM evolved in the domain of object orientation the terms runtime metamodel and instance of a metamodel will be used from now on, instead of type graph and typed graph. Nevertheless, the term host graph will be kept and used as a syn-

onym for an instance of a runtime metamodel. This relationship is visualized in Figure 1.

The described elements–the runtime metamodel and instances of the runtime metamodel–are usually visualized with the help of the Unified Modeling Language *(UML)* [2]. The runtime metamodel is visually presented by the use of a UML class diagram containing classes, associations, and attributes. Sometimes classes are called types, especially in the context of an instance of a particular class. Then the class of a particular instance is called the type of that instance. Similarly to the relationship between UML object diagrams and a given class diagram, instances of the runtime metamodel are accordingly related to the runtime metamodel. Thus these instances are visually presented by object diagrams.

In order to avoid ambiguities in subsequent chapters, the nodes within host graphs are called objects and the edges between these objects are called links from now on.

Let us give an example for the runtime metamodel and how it relates to the normal metamodel that describes the syntax of a modeling language. Therefore, think of a simple petri net. A petri net itself is nothing else than a model composed in the modeling language for petri nets. This modeling language has the metamodel that is shown in Figure 2.

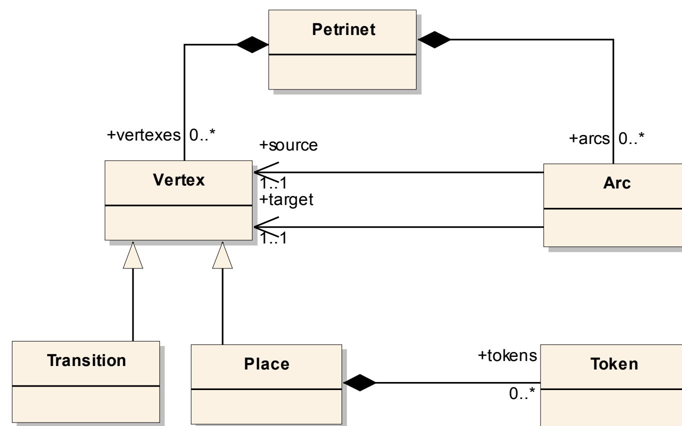

Figure 2: Metamodel and runtime metamodel for petri nets

This metamodel states that each petri net consists of instances of the type `Transition` and `Place` that are connected by instances of the type `Arc`. Furthermore it shows that each instance of the type `Place` can link arbitrarily many instances of the type `Token`. For simplicity reasons it does not ensure the constraint that places are only connected to transitions and vice versa.

In this special case the metamodel that describes the modeling language is equal to the runtime metamodel. Therefore, a direct mapping between the metamodel for the specification of the syntax and the runtime metamodel can be created. It already contains all the different pieces of information needed for the definition of the dynamic semantics of petri nets. Sometimes, in the case of petri nets, the type `Token` is regarded as a piece of information that does only belong to the runtime metamodel, since it describes a state of the petri net at runtime. Accordingly, it would not be used within the metamodel that describes the syntax of petri nets, but added if the runtime metamodel would be created.

2.3.2 DMM Ruleset

The latter counterpart of DMM cares about the change of the state of a model at runtime. Therefore, it specifies the changes of an instance of the runtime metamodel, becoming another instance of the same runtime metamodel. These changes define the meaning of constructs of the model that is described by the model composed in that modeling language. Because an instance

of a metamodel can be seen as a host graph, DMM is based upon the concepts of graph transformation to describe the changes between two directly successive instances of the runtime metamodel. These changes, as already mentioned, are given by DMM rulesets.

Each DMM rule within DMM ruleset is typed over the runtime metamodel that was described in the preceding section. This means that the constructs used within the definition of a DMM rule relate to the elements of the underlying runtime metamodel. Figure 1 visualizes this fact.

As a contrary to the description of traditional rules within the graph transformation domain, the visual representation of DMM rules only consist of one single side and not of a left-hand and a right-hand side. But this single side contains all the pieces of information that are needed in order to define the graph transformation properly. Nevertheless, the formalization of DMM rules given in [4] assigns the different constructs expressed within DMM rules to the two different sides of normal graph transformation rules. Therefore, the following paragraphs contain hints, to which side a particular construct would belong, if the rule would consist of two different sides.

Rules are presented visually by DMM. The visual representation of a single DMM rule resembles strongly to the notation of object diagrams and can therefore be intuitively used by the average programmer that is familiar with the concepts of object orientation. This resemblance was chosen, because these rules are matched and applied to host graphs and because host graphs are visually presented by object diagrams.

Running Example. Within the following descriptions of the different constructs of DMM rules, a graphical representation of one exemplary DMM rule is used. This graphical representation of the DMM rule is shown in Figure 3[3]. The exemplary rule is typed over the runtime metamodel visualized in Figure 2.

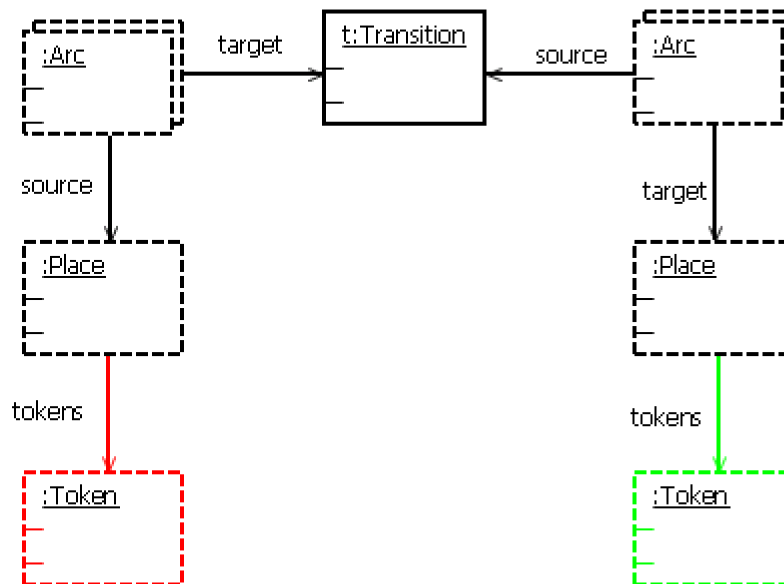

Figure 3: Exemplary DMM rule that specifies the firing of a transition

The exemplary DMM rule makes statements about one firing transition. If a transition fires, one token is consumed from each input place and put on all the output places. If there is at least one input place that does not contain a token then this rule cannot match the transition and therefore the transition cannot fire. Thus, the following example specifies the ability of a transition to fire and the changes performed if such a transition fires. This is one aspect of the dynamic semantics of the modeling language for petri nets.

3 This rule was already created with the extended graphical editor. The two small lines on the left-hand side of each node will be explained in section 3.2 on page 35. Currently, these lines are not important.

Nodes and edges. A DMM rule consists of nodes that are matched to objects and edges that are matched to links. Nodes and edges are together called elements. Matching means that they are mapped to each other and that the actions described by the particular node or edge are applied to the attached object or link within the host graph.

Similar to objects of the host graph that are typed over the runtime metamodel, the nodes of the DMM rule must have a type as well. In order to match successfully, these two kinds of types must fit to each other. This means that the type of the node must be the same or a supertype of the type of the object for a successful matching.

Nodes are visualized by a rectangle within DMM rules. The border and the color of the border define some properties of the node that will be described in the following paragraphs. An example for a node is the rectangle having the label `t:Transition` shown in Figure 3. The suffix, beginning at the colon, states the type of the node. The prefix is the name of the node. This name is used to identify the particular node within a DMM rule. As a result, the name of a node must either be unique or empty. Names of nodes do not have any impact on the matching.

Edges can only exist between nodes if there is an association between the two types of the nodes in the runtime metamodel. Thus, similar to the node's type, each edge has a corresponding part in the runtime metamodel–an association. Edges can only be matched to links if the adjacent nodes can be matched to the according objects and the link is an “instance of” the association that is assigned to the edge.

Edges are presented by arrows within DMM. The direction and the name are determined by the direction and the name of the according association defined within the runtime metamodel. The color of the edge expresses some properties that will be described below. Different edges can be seen in Figure 3. For instance, the edges with the name `tokens` refer to the association with the name `tokens` between the types `Place` and `Token`, given in Figure 2.

Each rule within the DMM ruleset has one special node–the context node. This node determines somehow the context to which the DMM rule is applied. The context node can be compared to the object on which a particular method is invoked within object oriented programming languages. The context node is not visually highlighted in the given example in any way.

Element role. Each node and each edge has exactly one element role. The element role is a property of the elements that specifies the according elements in more detail. For example, the role states whether a particular object within the host graph to which the node is mapped must be absent or present within the host graph such that the rule matches the host graph.

One value for the property role is called `exists`. It describes the duty of the existence of an object or link within the host graph. If, for instance, a node has this role, one object with a suitable type must be present in the host graph in order to be matched successfully. This kind of role can normally be found on the left-hand side or on both sides of the rule.

DMM uses the color black for these elements. Thus, the border of nodes is painted black and the arrows representing edges are painted black. In Figure 3, for example, the node with the label `t:Transition` must be mapped to an existing object within the host graph such that the whole rule can match the graph.

The second role that can be assigned to a node or an edge is called `destroy`. Similarly to the `exists` role this role claims for the presence of the according object or link within the host graph. But, furthermore, this role describes the action that is performed during the application of the rule. If an element does have this role, the according object or link is destroyed on application and does therefore no longer exist within the host graph after the application of the rule. Normally, as described above, graph transformation rules consist of two sides. The `destroy`

role would then be situated on the left-hand side of such a rule.

If an element has this role, it is painted red. This can be seen in Figure 3. There the node with the type `Token`–shown on the left side of the figure–is matched to an object that is removed on the application of the rule. Furthermore, the link connecting the according object of the type `Place` and the object of the type `Token` is deleted on application as well.

Thirdly, an element can have the role `create`. For each such element, an according link or object is created on application of the rule. These elements would normally appear only on the right-hand side of a rule because they describe which objects to create on application of the rule.

All elements that state that the according object or link should be created on the application of the rule are painted green: the nodes with a green border and the edges by a green arrow. Within the example rule given above such a node can be found on the right side of the figure. This green sub part of the rule states that a new object of the type `Token` should be created and connected to an existing object of the type `Place` via a newly created link called `tokens`.

Last but not least, elements can have the `not exists` role. This role is a Negative Application Condition *(NAC)*. If a node or an edge has the `not exists` role, an according object or link is not allowed to be present in the instance of the runtime metamodel such that the rule can match the graph. Thus, it must not be possible to match an object or link by the node or edge. If more than one element–having this role–is connected to each other, such a structure is called a not exists cluster. Then, the whole structure is not allowed to be present within the host graph. If only parts of the structure are present within the host graph, then the rule can still match the host graph. The direction of the edges that achieve the connection does not matter. Elements with this role can be found on the left-hand side of a rule, if the rule consists of two sides.

This kind of role is represented visually by a prohibition sign, indicating that this element is not allowed to be present. A node having this role is not shown within Figure 3. But Figure 10 on page 36 contains a node with this role.

Quantification. Besides these roles, there is a special property for nodes, called quantification. Basically, this property enables DMM to create rules that are much more generic and expressive. This means that by the use of this property, it is possible to define nodes within DMM rules that might match more than one object within the host graph, thus enabling the possibility to perform changes to arbitrary many objects or links within the application of one rule.

The property quantification can have four different values. One value, denoted as `1`, is the default quantification of a node. By this, the node under consideration matches only exactly one object within the host graph. Nodes with this quantification are called non-quantified nodes. If the quantification is not given explicitly, it is implicitly meant that the particular node is a non-quantified node. The set of nodes that contains all the non-quantified nodes is called root cluster. The root cluster does not necessarily have to be connected.

Nodes within the root cluster are visually presented with a solid border. For example, the node with the label `t:Transition`–given in Figure 3–is a non-quantified node. It matches only exactly one object of the type `Transition` within the host graph. The root cluster contains only this node, as no other non-quantified nodes exist within the given example.

The second and third value of the quantification property have a pretty similar meaning. They are denoted as `0..*` and `1..*` and state that this particular node can be mapped to arbitrary many objects. The second one additionally specifies that the DMM node with this quantification has to match at least one object within the host graph. If a node has this quantification, it is called a universally quantified node–or simply a uqs node. A connected set of universally quantified nodes is called a uqs cluster. Such a set must be as large as possible. Thus, a uqs cluster must not

be connected to a uqs node that does not belong to it. Connected, in this case, does not depend on the direction of the edge. The DMM rule is interpreted as an undirected graph in this case. All nodes within an uqs-cluster must have the same `quantification`, either `0..*` or `1..*`. If a uqs cluster contains more than one node, the whole structure of the uqs cluster is mapped to parts of the host graph–including the edges that connect such a uqs cluster.

Depending on the quantification–either `0..*` or `1..*`–the border of universally quantified nodes is painted differently. In the former case, every node is bound by two dashed borders, like it can be seen in Figure 3, indicating that this node can match arbitrary many objects within the host graph. If the node has a quantification `1..*`, the node is bounded by two borders as well, but where the inner border is solid and the outer one is dashed. The nodes of the type `Arc` are universally quantified nodes with the quantification `0..*`. Thus, every incoming arc and every outgoing arc of one matched transition within a petri net is matched by the two universally quantified nodes. These two nodes belong to different uqs cluster.

Finally, the quantification property might have the value `nested`. Nodes with this quantification are called nested nodes. A nested node can also match arbitrary many objects within the host graph but has a different meaning. While a uqs node, respectively a uqs cluster, tries to match as much sub graphs of the host graph as possible, the nested node states which structure has to be present, not present, destroyed or created for each such subgraph.

Thus, every nested node has to be connected to exactly one uqs cluster. This means that the nodes in between the nested node and the uqs cluster can only have the quantification `nested` itself. The greatest possible set of nodes that have the quantification `nested` and are connected to the same uqs cluster is called nested cluster. Thus, every uqs cluster has exactly one nested cluster–possibly an empty one. A uqs cluster and its according nested cluster together is called a quantification cluster. Nodes with a quantification different than `1` are called quantified nodes.

Uqs cluster and nested cluster together enrich the expressiveness of DMM to a great extent. With a uqs cluster, repeated structures within the host graph can be matched and–with the help of nested clusters–constraints for the matching of the rule formulated for these structures. By this, it is possible to express that a universally quantified node must have a nested node of a particular type. This results in the meaning that every object that is matched by the universally quantified node has to be connected to another node with a type suitable to the nested node's type.

Nested nodes are bordered by a dashed border like shown within Figure 3. The nodes of the type `Place` and of the type `Token` are such nested nodes. Let us focus on the left side of the figure. Together with the according universally quantified node of the type `Arc`, this part of the rule specifies that every incoming arc must be connected to a place–an input place–where the place must have at least one token. On application of the rule, one token is deleted from each input place. The right side of the rule specifies accordingly that a token is created for each output place. Therefore, the rule given above specifies when a transition is allowed to fire, and which changes are performed to the petri net by this firing transition.

Kinds of Rules. On the whole, three different kinds of rules can be distinguished. These three different kinds are called Bigstep rules, Smallstep rules, and Premise rules. In order to explain the different kinds of rules, a special concept, the concept of invocations, has to be introduced first.

Invocations in the domain of Dynamic Meta Modeling can be compared to method calls on a particular object within object oriented programming languages. They specify which rule has to be invoked on which given node. The node on which the rule is invoked is called target node. This target node is bound to the context node of the invoked rule.

By this concept, vast changes of the host graph or checks of the host graph can be split into many smaller changes and checks that are described in separate rules. Similarly to normal method calls, parameters can be passed to the invoked rule. Parameters are nodes within the invoking rule. The parameters of the invoking rule are called actual parameters. The parameters of the invoked rule are accordingly called formal parameters. In order to perform a correct invocation the actual parameters have to fit to the formal parameters and the type of the target node has to be compatible to the type of the context node of the invoked rule.

Invocations are visually presented by an arrow pointing to the target node and a label containing pieces of information about the invocation. This label contains the name of the invoked rule and the names of the actual parameters, represented by a single string. Every node that is used as actual parameter must have a name such that it can be uniquely identified. An example for the visual representation of an invocation can be found in Figure 34 on page 79.

Now, the distinction between the different rule types can be given. Let us start with Bigstep rules. A Bigstep rule cannot be invoked. It can only match the host graph directly. The rule presented within Figure 3 is a Bigstep rule. The rule itself does not make use of the invocations concept.

Smallstep rules, on the other hand, can only be applied to the host graph if they are invoked by a Bigstep rule or another Smallstep rule. Smallstep rules can contain exactly the same constructs like Bigstep rules. But if an invoked Smallstep rule does–erroneously–not match the host graph, an exceptional situation results. This normally indicates specification failures, as the Smallstep rule was not allowed to be invoked in this particular situation.

Finally, a DMM rule can be a Premise rule. Premise rules can only be invoked by Bigstep rules and Premise rules themselves. Premise rules only affect the matching of a rule and must completely consist of elements that would normally be situated on the left-hand side of a graph transformation rule. They are not allowed to change the host graph in any way[4].

The different rules that are defined within a DMM ruleset can overwrite each other [10]. But, because the concept of overwriting is not affected by the changes made to DMM rulesets in the course of the introduction of attributes and because this introduction should only give a brief overview over DMM, this concept is completely omitted here.

All in all, these descriptions were given rather informally. A formal specification of the different concepts can be found in [4], [10] and [11]. At this place, the reader should only get an overview along general lines.

2.4 Further Usages of DMM

The attentive reader might already have recognized that DMM is not limited to the specification of the dynamic semantics of modeling languages. It is much more able to describe the changes of an arbitrary instance of a class diagram–namely an object diagram–formally by the usage of visually represented graph transformation rules. Thus, it is possible to program within DMM and therefore specify the changes of object structures of a running program.

4 Although nodes that have the role `destroy` belong to the left-hand side of a normal two sided rule, they are not allowed to be used within Premise rules as well, since they would change the host graph.

3 Extension of DMM

This chapters elaborates all the new constructs and concepts based on the original version of the Dynamic Meta Modeling. Therefore, it divides the extensions into two parts–one affecting the formal extension and the other the visual one.

As already mentioned, there is only a brief concept of attributes within DMM. This thesis introduces attributes completely and profoundly into DMM and the available tooling. Section 3.1 contains an introduction of conditions and assignments to DMM rules in a partly formal and partly informal way. Furthermore, an expression language used for the definition of conditions and assignments is elaborated and described in this section. Finally, this section covers the changes made to the DMM metamodel that itself describes all the possible DMM models–meaning all the possible DMM rulesets. The whole section is interleaved by a lot of different visual examples describing the new constructs and concepts of DMM.

The visual representation used within these examples is summed up in section 3.2 where all the newly invented graphical elements like attributes, conditions and assignments are presented.

3.1 Formal Extension of DMM Rules

Some aspects of the fully introduction of attributes into DMM rules were already given in section 1.2 on page 2. Nevertheless, these concepts are repeated and analyzed in detail in order to create a solid foundation for the implementation of the extensions of DMM. Because only the rules itself have to be modified for the introduction of attributes and as the global view provided by DMM rulesets, comprising these rules, is not affected in any way, no descriptions are given in the following sections about the modification of whole DMM rulesets.

Attributes in the runtime metamodel. As DMM rules are typed over a given runtime metamodel, attributes within DMM rules can only be used, if they are used within the runtime metamodel as well. But, because attributes are a common part of UML class diagrams and as runtime metamodels are usually visually presented by UML class diagrams, the formal addition of attributes to the runtime metamodel is straightforward and clear. Therefore, the introduction of attributes to runtime metamodels is avoided at this place.

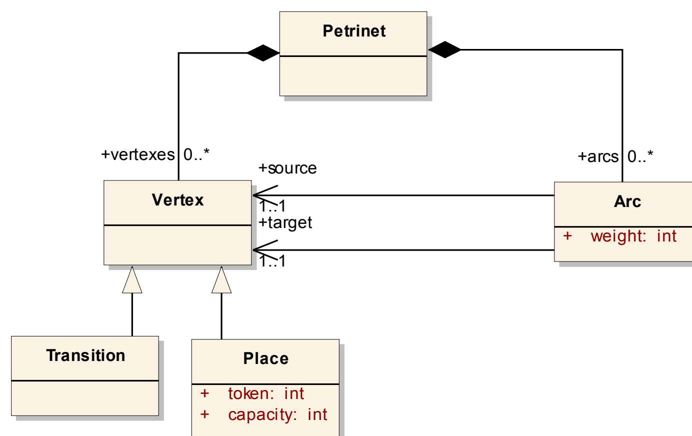

Figure 4: Runtime metamodel for petri nets, including attributes

In order to give examples for the vivid explanation of the new constructs and concepts a runtime metamodel is introduced that contains attributes. Therefore, the running example is modified and extended such that it makes use of attributes.

Figure 4 shows the modified runtime metamodel for petri nets. It now expresses the extensions of petri nets that were describe in section 2.2 on page 5. Henceforth, it is possible to define capacities for the places and weights for the arcs. One main difference is the elimination of the type `Token`. This class is replaced by the variable `token` that holds the number of tokens that are currently stored at the given place.

New constructs of DMM rules. The following paragraphs contain three constructs that are added to DMM rules properly such that the dynamic semantics of behavioral languages whose runtime metamodels contain attributes can be specified. These three new constructs are attributes, conditions, and assignments.

All different kinds of expressions are defined thoroughly in subsection 3.1.1. There, all parts, starting with tokens crossing the grammar and ending by the definition of well-formedness rules, are specified, partly formally and partly informally. For now, these expressions are used intuitively in order to give examples for the newly added constructs.

Attributes. As already mentioned, DMM rules are typed over the runtime metamodel. Instances of this runtime metamodel are called host graphs. These host graphs can now contain objects that have attributes and values for these attributes, resulting from the addition of attributes to the runtime metamodel. By this, the attributes within the host graph belong to particular objects. They are defined within the type of the object in the runtime metamodel.

As nodes of DMM rules are mapped to objects of the host graph, it makes sense to arrange DMM rules in such a way that attributes used within DMM rules belong to nodes. More correctly speaking, the attributes within a DMM rule belong the type of the node. For simplicity, this is meant if it is said that a node contains attributes or if it is stated that an attribute belongs to a node.

Attributes have a type themselves that defines all the possible values of an attribute. From now on the types of attributes and the according literals are called data types in order to reduce ambiguity.

There is no special part of the node used for the visual representation for attributes of its own. They do not have to be explicitly selected, but can be directly used within DMM rules. An attribute can be referenced by a string called identifier. The identifier refers uniquely to the target attribute, by the usage of the name of the attribute and the name of the node to which the attribute belongs. This node is called target node[5], in the context of the given identifier.

Conditions. Secondly, conditions can be added to nodes. The addition of conditions to nodes is more reasonable than the addition of a set of conditions to a whole DMM rule, since it increases the relationship to the notation of UML object diagrams. Therefore, conditions have some type of context–the node to which they belong. Conditions, like already mentioned, consist of exactly one expression that must evaluate to a `boolean` value. Since conditions affect the matching of a rule, these constructs would normally be situated on the left-hand side of a rule if the rule would consist of two sides.

Conditions are represented visually within a special container that is situated directly under the underlined label of the node. Figure 5 shows such a condition that belongs to the node with the label `p:Place`. This rule states that the place under consideration must obey its capacity constraint. Therefore, the expression defining the condition uses the identifiers `token` and `capacity` that refer to attributes defined within the type `Place`, and the identifier `weight` that refers to the target attribute that belongs to the type `Arc`.

5 Although the term target node is already defined in the context of invocations, no ambiguities result if this term is overloaded by a second meaning.

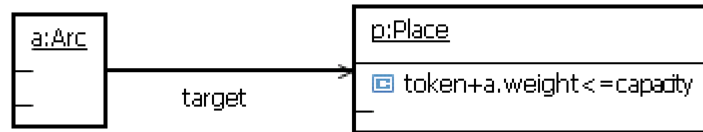

Figure 5: DMM rule that demonstrates the usage of conditions

Assignments. Finally, the usage of assignments must be explained. Similar to conditions, assignments are added to a node within a DMM rule in order to create some kind of context for each assignment. Assignments are only allowed to define the new value of attributes belonging to the same node as the assignment itself.

Assignments consist of two parts: An identifier–referring to an attribute to which the new value is assigned–and an expression that defines how this new value is calculated. Similar to expressions that can be used for the definition of conditions, the same kind of expressions are used for assignments. Nevertheless, they do not have to be evaluated to a `boolean` value since the identifier on the left-hand side of the assignments might refer to an attribute having another data type than `boolean`.

Figure 6 shows an exemplary assignment to the attribute with the name `token`. For the set of assignments that can be defined for each node, a special container that is situated directly under the container for the conditions was created. The small character `'` after the identifier `token` will be explained below. The given assignment states that the number of tokens belonging to the object that is matched by the given node is increased by the weight of the incoming arc.

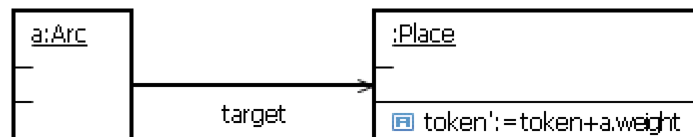

Figure 6: DMM rule that demonstrates the usage of assignments

Two Value Concept. One interesting aspect resulting from the introduction of assignments is that by assigning a new value to an attribute there might be the possibility to refer to two different values of attributes. Let us give this characteristic or respectively concept a name. It is from now on called the Two Value Concept *(TVC)*.

The first value is the value of the attribute before the rule has been applied and before the assignments have been evaluated. The second value of an attribute is the value after the evaluation of the assignments. But, although the second value of the attribute is the one after the application, it is possible to use both kinds of values in the definition of assignments.

However, then the constraint must be given that if the new value of the attribute is used within any assignments, it has to be defined prior to the usage during the same rule application. Although it might be possible to define the old value to be the new value, if the new value is not defined, the additional constraint ensures a more robust usage of the TVC. One great advantage of the introduction of the TVC is that it enables to express something like sequential assignments during the application of a DMM rule, although a graph transformation is performed instantaneously.

The TVC was already used in Figure 6, where the new value of the attribute `token` was calculated by an expression that uses the old value of the same attribute. The new value of attributes is denoted by a `'` after the identifier that refers to the particular attribute.

Adjustments. By the introduction of these three different parts the meaning of other concepts that are currently present within DMM have to be adjusted and precisely reformulated. There-

fore, the following part consists of adjustments and customizations of the meaning of the different constructs and concepts. The word printed out boldly at the beginning of each part states which part of DMM rules will be described and reformulated.

Restrictions resulting from the orthogonal use of these new constructs and old constructs are formulated within the following paragraphs and later on recapped as well-formedness rules. These well-formedness rules ensure that valid DMM rules are created that do not have ambiguities or violations against the specification of the DMM. They are summed up and formulated precisely in section 3.1.1.3.

Inheritance. Because the runtime metamodel can make use of inheritance, a node can have different indirect types at the same time. Therefore, every attribute can be used that belongs to the direct type of a node or one of its supertypes. It is said for simplicity that all these attributes of the supertypes and all the attributes of the type of the node belong to the node.

In order to avoid ambiguities, this imposes an additional constraint on the runtime metamodel. Every attribute name must be unique with respect to all attributes that belong to one node.

Element Roles. There are some restrictions to the usage of assignments, conditions and attributes resulting from the node's role. The different roles that can be assigned to a DMM node are `create`, `exists`, `not exists` and `destroy`.

Table 1 shows whether a particular construct that was newly introduced into DMM can be used within nodes of different roles. The table consists of four rows that represent the four different node roles and four columns which hold the different constructs that were added to DMM rules–these are assignments, conditions and–resulting from the TVC–the new and the old value of an attribute.

An entry within the last two columns states whether the new or the old value of an attribute that belongs to a node with the role given in the column can be used used within an expression defined on any node within the same DMM rule. For example, the first value in the third column states that attributes that belong to nodes that have the role `exists` can be used within any expressions across the DMM rule.

Role	Assignment	Condition	New Value	Old Value
`Exists`	yes	yes	yes	yes
`Not Exists`	no	yes	no	yes[6]
`Destroy`	no	yes	no	yes
`Create`	yes	no	yes	no

Table 1: Interdependence between the node's role and the newly introduced constructs

Nodes that have the role `exists` can make use of conditions and assignments and every expression can refer to attributes that belong to these nodes.

There are some exceptions concerning nodes that have the role `not exists`. This kind of nodes are checked for non-existence such that a rule can match the host graph. Since the according objects are not allowed to exist within the host graph, it does not make sense to assign new values to their attributes. Henceforth, it is not allowed to define assignments on nodes with this role.

Nevertheless, there is one reasonable construct that is supported by DMM rules in order to enrich the expressiveness of DMM. By the permission to use conditions within nodes with this role, it is possible to check if particular objects do not exist within the host graph that fulfill the specified

6 Not all expressions can refer to the old value of attributes that belong to a node with the role `not exists`. There are some exceptions that are given in the following paragraphs.

conditions. Thus, it is possible to limit the kind of objects that are not allowed to exist within the host graph. That is why these conditions can make use of the old value of attributes that belong to nodes with the role `not exists`.

Besides this usage of attributes that belong to not exists nodes there is another possibility to refer to attributes that belong to these nodes. Although this might at first seem as a contradiction–to refer to a node's attributes that do not exist–the usage can be reasonable justified. If more than one node with the role not exists builds a not exists cluster, it is possible to use all the attributes that belong to nodes of the same not exists cluster. Thereby, the DMM rule expresses which structure is completely not allowed to be present within the host graph. Thus, the rule can still match the graph if only one part of the not exists cluster could not be matched.

To sum up the given ideas, this paragraph states exactly which nodes can make use of the old value of attributes that belong to nodes with the role `not exists`: old values of attributes of `not exists` nodes can only be referenced by conditions that belong to the same or other nodes with role `not exists` that belong to the same not exists cluster. Besides, nevertheless, it is not possible to use attributes belonging to this kind of nodes within other expressions, like for example expressions that belong to conditions defined within nodes that have the role `exists`.

Nodes that have the type `destroy` cannot have assignments because this kind of nodes are destroyed on application of the rule. As a result, it is not possible to use the new value of attributes belonging to these nodes.

Finally, some reasonable restrictions must be given for nodes that have the role `create`. This kind of nodes cannot contain conditions because they are created on application of a rule and thus are only situated on the right-hand side of a rule–if a rule would consist of two sides. But, because conditions belong to the left-hand side, it is not possible to check a condition of a node that describes that a particular object in the host graph will be created on application. Additionally, the old value of attributes belonging to these nodes cannot be used within expressions defined on any arbitrary node, since it does not exist within the host graph.

Quantification. By the introduction of attributes, and hence the introduction of conditions and assignments, the meaning of quantified nodes has to be adjusted and refined. Mainly, different restrictions of usages of the attributes has to be given, in order to create an unambiguous and clear meaning of these constructs.

On the whole, it is possible to define conditions and assignments on quantified nodes, just like they are defined and used on non-quantified nodes. Therefore, conditions and assignments defined on quantified nodes look exactly the same as conditions and assignments defined on non-quantified nodes. An example for their visualization can be found in Figure 7.

Especially conditions affect the meaning of uqs nodes and accordingly of uqs cluster. As described in subsection 2.3.2 on page 7, uqs cluster match whole structures within the host graph. Now, by the definition of conditions within these uqs nodes, the structure is only matched if all conditions defined within this uqs cluster are fulfilled by the structure under consideration. Thus, conditions now impose an additional constraint that has to be fulfilled by structures within the host graph in order to be matched by the uqs cluster.

Conditions used within a nested cluster that belongs to a uqs cluster state which constraints have to be fulfilled by the structures matched by the nested cluster such that the rule matches the graph successfully and can therefore be applied. The usage of conditions within nested clusters can be seen in Figure 7. This figure shows a DMM rule that is typed over the extended runtime metamodel given in Figure 4 on page 13. If this rule matches the host graph, the transition that is matched be `t:Transition` can fire and accordingly consume and produces tokens. The condition within the quantification cluster on the left-hand side states that every input place must have

at least as much tokens as the weight of the arc that connects that place to the transition `t`. The condition that belongs to the nested node visualized on the right-hand side, ensures that the capacity constraint of each output place is obeyed.

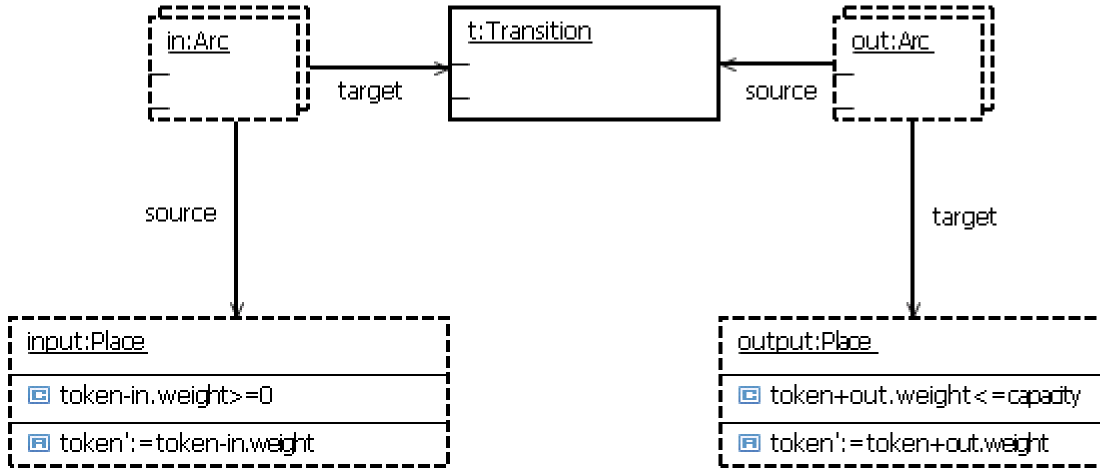

Figure 7: DMM rule for the specification of a firing transition

But, as always in life, some restrictions concerning the usage of attributes of nodes have to be given. Especially the usage of attributes that belong to quantified nodes has to be constrained. The possible situations that can occur for two different quantification cluster `A` and `B` are summed up in Table 2. The source node is the node that contains an expression which refers to the given attribute that belongs to the target node. The target node, as already defined, is the node that contains this particular attribute.

Source Node / Target Node	uqs node within quantification cluster A	uqs node within quantification cluster B	nested node within quantification cluster A	nested Node within quantification cluster B	non-quantified node
uqs node within quantification container A	yes	no	yes	no	no
nested node within quantification cluster A	no	no	yes	no	no
non-quantified node	yes	yes	yes	yes	yes

Table 2: Possible references to attributes that belong to quantified nodes

There is one special case that should be mentioned separately. This is the usage of attributes that belong to nested nodes within expressions defined on uqs nodes of the same quantification cluster. This is not possible, since first the uqs cluster defines which structures within host graphs a particular uqs cluster matches and thereafter the nested cluster states which additional structures must be present within the host graph such that the rule matches the host graph successfully. Therefore, when the conditions for the uqs cluster are checked for the according structures within the host graph, the structures matched by the nested cluster do not exist, and accordingly their attributes are not visible.

All in all, the usage of attributes that belong to quantified and non-quantified nodes–which affect the quantification property–can be summed up by the following two well-formedness rules. First, attributes that belong to universally quantified nodes can only be used by all nodes of the same quantification cluster. Secondly, attributes that belong to nested nodes can only be used by nodes that belong to the same nested cluster.

Kinds of rules. The introduction of assignments affects the different rule types slightly. Because Premise rules can only contain constructs that are situated on the left-hand side of two sided graph transformation rules and because assignments can be found on the right-hand side, there is no other possibility than to limit the usage of assignments to Smallstep rules and Bigstep rules.

Overwritten Rules. Overwritten rules [10] are nearly not affected by the introduction of attributes, conditions and assignments. There is only one constraint resulting from the kind of overwriting of DMM rules that has to be obeyed. Because the left-hand side of the overwriting rule must consist out of a subgraph of the left-hand side of the overwritten rule and similarly the right-hand side must contain the right-hand side of the overwritten rule as a subgraph, this relation has to be conveyed on conditions and assignments. Thus, all the conditions within the overwritten rule must exist in the overwriting rule and all the assignments defined within the overwritten rule must, similarly, be present in the overwriting rule.

Matching and Application. By the introduction of the different constructs that were described above the term of matching and the meaning of applying a rule to a host graph has to be refined.

Now, a DMM rule matches a host graph if all the old constraints–like the presence or absence of a link or a object–hold and additionally all conditions that are defined within the DMM rule are evaluated to the `boolean` value `true`. Only in this situation the DMM rule can be applied to the host graph.

On application of a DMM rule, beside all the old changes that could be expressed, like the deletion or the creation of an object or respectively a link, the values of the attributes have to be changed accordingly to the assignments defined within the DMM rule. The expression–as a part of the assignment–is evaluated by the usage of all the current values of the attributes within the host graph and the new value is calculated according to the calculation instruction given by this expression.

Furthermore, all the matching and application characteristics that are affected by quantified nodes and nodes that have the role `not exists` have to be obeyed. These two constructs– enriched by attributes, conditions and assignments–limit the matching of a rule by the addition of new constraints that have to be fulfilled such that a rule matches the host graph. These additional constraints were described above within the according paragraphs.

3.1.1 Introduction of an Expression Language

This section defines and elaborates an expressions language that is used to express conditions and assignments within DMM rules.

One main driver of the development of the expression language was that it should be easily extendable. This driver results from the fact that there would always be possible dynamic semantics that can not be fully specified by the already elaborated expression language. Therefore it should be possible to enhance the expression language such that it can express new kinds of expressions that are needed for the specification.

As an example for the modifiability and extensibility of the expression language the modifications are given that would result from the introduction of the power operator `^`[7]. Thereby, it should be shown how the definition of the expression language can be expanded. This example is used within this chapter at different places.

In parallel to the usual way of creating a formal textual language, this subsection describes these actions in the same order. This order is visually presented in Table 3. In this table lexical tokens–

7 This operator is not introduced into the expression language because it is currently not needed in order to specify the dynamic semantics of behavioral languages.

the smallest part of a language–are visually presented by a circle containing the string representation of the token. Expressions are visualized by a square. If an expressions consists of subexpressions the subexpressions appear as a square inside the other square.

Lexical Tokens	12 23 3 + true ==
Possible Expressions	12 + 23 == true
Well-formed Expressions	12 + 23 == 3

Table 3: Exemplary elements on each layer of the definition of the expression language

The current subsection starts by the definition of all the possible lexical tokens in subsection 3.1.1.1. Then subsection 3.1.1.2 continues with the presentation of all kinds of possible expressions and an elaboration of a grammar that formally describes all these kinds of expressions. Finally, it states all the constraints that could not be expressed by the grammar in subsection 3.1.1.3. These constraints–called well-formedness rules–limit all the possible expressions that are accepted by the grammar to the well-formed expressions with respect to a given rule, its nodes, and their attributes.

3.1.1.1 Lexical Tokens

This subsection defines all the possible tokens that can be used within expressions–like literals, attribute names and operator symbols. In order to reduce ambiguity the tokens that refer to an attribute are called identifier from now on. This means that an identifier refers to, or respectively identifies, an attribute uniquely within a whole rule. Furthermore, the current subsection states which data types are supported by DMM.

Let us start with the definition of data types. DMM limits the possible data types of attributes and literals to `integer`, `boolean`, `double` and `string`. Further data types that might be familiar to Java programmers like `float`, `byte`, `short` and `long` are not supported by DMM directly. But there exists still the possibility to use these data types within runtime metamodels. Therefore everything concerning the data type `float` is mapped to the data type `double` and similarly everything related to the data types `byte`, `short`, and `long` is mapped to the data type `integer`.

The following paragraphs use regular expressions [12] for the construction of the literals and identifiers. Therefore, a special notation is used that differs a little bit from the original notation of regular expressions in order simplify the resulting regular expressions. Let `1-9` or `a-z` denote a regular expression where the meaning is similar to the alternation of all the elements located in between the border characters. Thus, the regular expression `1-9` means the same as the regular expression `1|2|3|4|5|6|7|8|9`.

Table 4 contains all the different regular expressions used to define the possible tokens that build the foundation of the expression language. The colors that are given in Table 4 are from now on used throughout the rest of the section for the four different data types.

Name of the Regular Expression	Regular Expression
`boolean_literal`	`(true|false)`
`integer_literal`	`(0-9)+`

<table>
<tr><th>Name of the Regular Expression</th><th>Regular Expression</th></tr>
<tr><td>double_literal</td><td>(0-9)+.(0-9)*(exp)?|.(0-9)+(exp)?|(0-9)+exp
with
exp as (e|E)(+|-)?(0-9)+</td></tr>
<tr><td>string_literal</td><td>“(~(”|\)|\(\|”))*”</td></tr>
<tr><td>identifier</td><td>(subident.)?subident(')?
with
subident as (_|a-z|A-Z)(_|a-z|A-Z|0-9)*</td></tr>
<tr><td>set of operators</td><td>{+, -, *, /, %, !, &, |, min, max, abs, <, <=, ==, >=, >, :=}[8]</td></tr>
</table>

Table 4: Summarization of all possible tokens and their regular expressions

Boolean Literals. Let us start with the definition of literals of the data type `boolean`. They are only allowed to be `true` or `false`. As a result, these special words are not allowed to be used as identifiers. The regular expression for such literals is therefore easily defined by `(true|false)` and from now on called `boolean_literal`.

Integer Literals. Literals of the data type `integer` are described within this paragraph. Negative literals of this type can be noted by a special prefix operator `-`, but as this symbol is an operator and is accordingly used within an operation, negative literals can be omitted in the definition of the regular expression for literals of the data type `integer`. The regular expression for a positive `integer` literal is `(0-9)+`. This regular expression is called `integer_literal`.

Double Literals. For the creation of the regular expression for literals of the data type `double`, an intermediary regular expression must be defined such that the overall regular expression can be expressed easier. This intermediary regular expression is called `exp` and states how the exponent of literals of the `double` data type look like. The `exp` regular expression is `(e|E)(+|-)?(0-9)+`.

It is possible to distinguish three different looks of literals of the data type `double`. Consequently, the whole regular expression for these literals consist of three sub regular expressions. The first sub regular expression covers all the literals that start with a number contain a dot and end with some numbers after the dot. This sub regular expression is `(0-9)+.(0-9)*(exp)?`. Another opportunity to write down a `double` literal is to skip the numbers in front of a dot, but write the dot explicitly and continue with the positions after the decimal point. This results in the sub regular expression `.(0-9)+(exp)?`. Finally, it is possible to leave out the dot and only write the numbers in front of the decimal point. But then an exponent has to be explicitly given such that `double` literals of this kind can be distinguished from `integer` literals. This sub regular expression equals `(0-9)+exp`.

This results in an overall regular expression for literals of the data type `double` of `(0-9)+.(0-9)*(exp)?|.(0-9)+(exp)?|(0-9)+exp`. This regular expression is called `double_literal`.

String literals. Strings, or to be more precise, literals of the data type `string` have to be enclosed in quotation marks “. They are allowed to contain any sequence of letters and numbers and special characters between the two inverted commas. But, if a quotation mark should be used within the string, it must be escaped by a backslash \.

8 This is no regular expression, but rather a normal set containing all the possible operators. Likewise, the set of operators is not the name of a regular expression as shown in the table.

In order to specify the regular expression easily without writing down all the different special characters, let us modify the notation of regular expressions a second time. Let the expression `~(a|b)` denote all the characters without `a` and `b`. Thus, the regular expression `"(~("|\)|\(\|"))*"` defines all the possible values of literals of the data type `string`. Let us call this regular expression `string_literal`.

Identifiers. Because it is possible to refer to attributes that belong to another node, the identifier must be able to identify the target node and the according target attribute uniquely. Therefore, the identifier consists of two sub identifiers which together resemble strongly to a notation that is used within object oriented programming languages. The first sub identifier is optional and defines the target node. The second sub identifier specifies the target attribute that must belong to the target node. These two sub identifiers are separated by a dot. If the first sub identifier and the dot are missing the whole identifier refers to an attribute that belongs to the same node as the expression containing this identifier.

Therefore, each sub identifier has to be specified by a separate sub regular expression. In conjunction these two sub regular expressions build the regular expression for identifiers that is called `identifier`. Both sub regular expressions for the sub identifiers are absolutely similar as they specify names of nodes or respectively names of attributes. Thus, a special regular expression for a sub identifier is introduced.

Possible sub identifiers of attributes must start with a letter or an underscore and can consist of an arbitrary amount of letters, numbers and underscores. Therefore, the sub regular expression `(_|a-z|A-Z)(_|a-z|A-Z|0-9)*` defines all the possible sub identifiers of attributes and is called `subident`. If the character `'` is used after the name of an attribute, the identifier denotes the new value of the attribute, like defined by the TVC that is described in section 3.1 on page 15. Without this character the old value is referenced. Together with the first optional sub identifier that refers to a node separated form the attribute sub identifier by a dot the regular expression `(subident.)?subident(')?` results.

Finally, the following set contains all the possible operators: `{+, -, *, /, %, !, &, |, min, max, abs, <, <=, ==, >=, >, :=}`. In this set, the symbol `,` is only used to separate the different operators. The meaning of all the operators is given in the following subsection.

3.1.1.2 Possible Expressions

This subsection defines all the possible expressions and elaborates a grammar for these expressions. Note that not all the constraints to the elaborated expression language are given by a grammar. Further well-formedness rules that limit the possible expressions to the well-formed expressions are defined in section 3.1.1.3.

The reason for the elaboration of an expression language is to build conditions and assignments for the use within DMM rules. Therefore, some kinds of expressions are supported that are commonly used for the specification of the dynamic semantics of behavioral modeling languages. All the different kinds of expressions are parted into four major groups according to the data type of their operands[9]. These data types are `boolean`, `integer`, `double` and `string`. Each kind of expression that contains subexpressions is from now on called operation, consisting of arbitrary many input operands, an operator and exactly one result. Operands can be literals, identifiers or even other operations. Let us distinguish the term expression from the term operation. An expression is much more generic. Every literal, identifier or operation itself is an expression. In the case of an operation, the operands are sometimes called subexpressions.

9 All the input operands must always have the same data type.

Table 5 contains all the different operators–and their priority–that are supported by the elaborated expression language. The priority decides which operation is going to be evaluated first and which thereafter. The higher the priority of an operator, the more associative it is and the more it “holds its operands together”. For example, the expression `a & b | c` would be evaluated by first calculating the result of `a & b` and thereafter calculating the disjunction of the results and `c`.

Priority	Operator
1	`{ | }`
2	`{ & }`
3	`{ == }`
4	`{ < , <= , >=, > }`
5	`{ + , - }`
6	`{ * , / , % }`
7	`{ - , ! }`
8	`{ min , max , abs }`

Table 5: Priorities for all the different operators

The priorities in Table 5 were taken from math expressions that can be defined within the programming language Java. Therefore, the Java's grammar was examined and the priorities extracted.

This priority list of operators is easily extendable. Let us look at the exemplary addition of the power operator ^ to this table of operators. This operator can be put in a separate set between the operator sets with the priorities 6 and 7.

All the different operations are summed up in Table 6. There, the name, the signature, the notation, the operator and the priority is given for every possible operation. The signature describes the expected data type of the operands and the result data type. The notation states if the operation is written in the infix or in the prefix notation. The priority is extracted from Table 5. The priority is used later on in order to build a grammar that obeys the priority rules. Similar to Table the colors are used to distinguish between the different classes of operations. All the operations that belong to the same class have operands of the same data type.

Operation name	Signature	Notation	Operator	Priority
Conjunction	`boolean x boolean → boolean`	infix	`&`	2
Disjunction	`boolean x boolean → boolean`	infix	`|`	1
Negation	`boolean → boolean`	prefix	`!`	7
Equality	`boolean x boolean → boolean`	infix	`==`	3
Addition	`integer x integer → integer`	infix	`+`	5
Subtraction	`integer x integer → integer`	infix	`-`	5
Multiplication	`integer x integer → integer`	infix	`*`	6
Division	`integer x integer → integer`	infix	`/`	6
Modulo	`integer x integer → integer`	infix	`%`	6
Minus	`integer → integer`	prefix	`-`	7

Operation name	Signature	Notation	Operator	Priority
Minimum	`integer x integer → integer`	prefix	`min`	8
Maximum	`integer x integer → integer`	prefix	`max`	8
Absolute Value	`integer x integer → integer`	prefix	`abs`	8
Greater-Than	`integer x integer → boolean`	infix	`>`	4
Greater-Or-Equal	`integer x integer → boolean`	infix	`>=`	4
Equality	`integer x integer → boolean`	infix	`==`	3
Less-Or-Equal	`integer x integer → boolean`	infix	`<=`	4
Less-Than	`integer x integer → boolean`	infix	`<`	4
Addition	`double x double → double`	infix	`+`	5
Subtraction	`double x double → double`	infix	`-`	5
Multiplication	`double x double → double`	infix	`*`	6
Division	`double x double → double`	infix	`/`	6
Minus	`double → double`	prefix	`-`	7
Minimum	`double x double → double`	prefix	`min`	8
Maximum	`double x double → double`	prefix	`max`	8
Absolute Value	`double x double → double`	prefix	`abs`	8
Greater-Than	`double x double → boolean`	infix	`>`	4
Greater-Or-Equal	`double x double → boolean`	infix	`>=`	4
Equality	`double x double → boolean`	infix	`==`	3
Less-Or-Equal	`double x double → boolean`	infix	`<=`	4
Less-Than	`double x double → boolean`	infix	`<`	4
Concatenation	`string x string → string`	infix	`+`	5
Greater-Than	`string x string → boolean`	infix	`>`	4
Greater-Or-Equal	`string x string → boolean`	infix	`>=`	4
Equality	`string x string → boolean`	infix	`==`	3
Less-Or-Equal	`string x string → boolean`	infix	`<=`	4
Less	`string x string → boolean`	infix	`<`	4

Table 6: Summarization of all possible operations and their characteristics

`Boolean` operations can be–amongst others–used for creating complex conditions that are evaluated in order to check if a rule matches the host graph. That is why the main operations like conjunction, disjunction, negation and equality are supported by the expression language.

The second great class of operations are the `integer` operations. These operations are used for calculations and comparisons between operands of the data type `integer`. For calculations the main four basic arithmetic operations can be used. In addition the modulo operation is appended to the list of calculations. The relationship between two operands of the data type `integer` can be checked in any arbitrary way. Finally, there are three additional operations–the maximum, the minimum and the absolute value operation. These operations were introduced into the expression

language because otherwise it would not be possible to find out the maximum, the minimum or the absolute value directly within one rule and the necessity would come up to create at least two rules only to express these simple operations.

Nearly all the described integer operations are supported for operands of the data type double, too. Only the module operation does not make sense for this data type as real numbers ℝ have a trivial divisibility calculation, where each real number can by divided by each other real number–except the number 0–without remainder. Therefore, it does not make sense to define the modulo operator for real numbers.

The last class of operations are string operations. In this case the operands are of the data type string. They can be concatenated or their relationship to each other can be checked in any arbitrary way.

The attentive reader might have recognized that the operator - has two different meanings, and is therefore contained within two different priority sets that are defined in Table 5. The reason for this is that the operator - is overloaded. The first meaning is the simple subtraction between two expressions of the data type integer or double. The second one is the negative value of a literal or an expression of the data type integer or double. These two different operators–that have the same symbol to represent them–have two different meanings and accordingly two different priorities.

In the case of the addition of the power operator ^, two additional operations have to be added to the Table 6, since it is possible to exponentiate numbers of the data type integer and double. One operation having the signature integer x integer → integer and the other one accordingly having the signature double x double → double.

Grammar. Now that all the operations are known and all the priorities of the operators are determined, a grammar for the expression language can be given.

The productions of the grammar are given in a notation that adheres the Backus-Naur-Form *(BNF)* [12]. This means that all the productions of the grammar are formed by a left-hand side and a right-hand side. The left-hand side is always a nonterminal. In BNF nonterminals are enclosed by angle brackets. The right-hand side is separated from the left-hand side by the operator ::=. This side contains pieces of information on how the nonterminal on the left-hand side is replaced–or as it is called in the domain of formal languages–how the nonterminal is derived. The right-hand side consists of a list of terminals and nonterminals. All strings that are enclosed in single quotation marks ' are terminals composed in the expression language. Finally, the parenthesis enclose opportunities of terminals and nonterminals. Exactly one of the terminals inside the parenthesis has to be chosen on derivation.

The grammar for the elaborated expression language can be seen in Table 7. The table consists of three columns. The first column only identifies each production or respectively set of productions that is given in the second column. All productions that derive the same nonterminal belong to the same set of productions. The last column shows which priority for each operator results from the ordering of productions. These priorities were given in Table 5.

No	Productions of the grammar	Prio
(1)	<assignment> ::= identifier ':=' <orExp>	-
(2)	<orExp> ::= <orExp> '|' <andExp> ::= <andExp>	1

No	Productions of the grammar	Prio
(3)	<andExp> ::= <andExp> '&' <equalityExp> ::= <equalityExp>	2
(4)	<equalityExp> ::= <equalityExp> '==' <inequalityExp> ::= <inequalityExp>	3
(5)	<inequalityExp> ::= <inequalityExp>('>=' | '>' | '<=' | '<')<additiveExp> ::= <additiveExp>	4
(6)	<additiveExp> ::= <additiveExp> ('+' | '-') <multiplicativeExp> ::= <multiplicativeExp>	5
(7)	<multiplicativeExp> ::= <multiplicativeExp> ('%' | '/' | '*') <unaryExp> ::= <unaryExp>	6
(8)	<unaryExp> ::= ('-' | '!') <unaryExp> ::= <primaryExp> ::= <mathPrefixExp>	7
(9)	<mathPrefixExp> ::= ('max' | 'min') '(' <orExp> ',' <orExp> ')' ::= 'abs' '(' <orExp> ')'	8
(10)	<primaryExp> ::= <literalExp> ::= identifier ::= '(' <orExp> ')'	-
(11)	<literalExp> ::= integer_literal ::= double_literal ::= boolean_literal ::= string_literal	-

Table 7: The grammar for the elaborated expression language

The given priorities are achieved by defining a nonterminal to be derivable from other nonterminals. Let us look at an example for this. The nonterminal <orExp> with the set of productions given at (1) can either be derived to an expression that contains the operator |, having the priority 1, or to the nonterminal <andExp>, increasing the priority by 1. Now, if this nonterminal is derived to an expression containing the & operator, this operator has priority 2. Nearly the same concept to find out the priorities is used for the implementation of the grammar. How exactly this is done is described in subsection 4.2.2 on page 41. How these priorities are reused in the process of unparsing is described in subsection 5.2.2 starting on page 60.

Extensions. In the case of the addition of the power operator ^, the given grammar can be enhanced easily. This is done by an addition of a new nonterminal, two new productions and a modification of the productions given at (7). The modified productions are:

```
<multiplicativeExp>
    ::= <multiplicativeExp> ('%' | '/' | '*') <powExp>
    ::= <powExp>
```

The newly added production looks as follows. The structure of this production is pretty similar to

nearly all other productions.

```
<powExp>
    ::= <powExp> '^' <unaryExp>
    ::= <unaryExp>
```

Because of these modifications, it would now be possible to use the power operator ^ within the conditions and assignments of DMM rules. This shows that the main goal of extensibility is attained by this grammar.

As described so far, the grammar limits the expressions to a set of expressions–called possible expressions–where still different constraints do not hold. It is, for example, now possible to define expressions that contain identifiers that refer to attributes which are not defined in the runtime metamodel. Furthermore, there are several possibilities where the data type of the different literals and attributes do not fit to the expected data types of an operator. Therefore, some additional constraints are needed that limit the possible expressions to the well-formed expressions. All these constraints are elaborated and given in the following subsection.

3.1.1.3 Well-formed Expressions

The kinds of constraints that are needed to limit the set of possible expressions to the set of well-formed expressions are called well-formedness rules. These constraints depend on the semantic–the meaning–of a particular construct and its relationship to the environment. They forbid all the possible expressions that are not well-formed, but accepted by the grammar.

Many well-formedness rules were already described briefly by the introduction at the beginning of section 3.1 starting on page 14. But these constraints have not been formulated precisely yet.

Well-formedness Rules. The well-formedness rules can be divided into six main groups. These groups are formed according to the DMM constructs about which the constraints make statements. All constraints that belong to one group specify the well-formedness of the same DMM construct.

The first group contains constraints formulated against identifiers. The second one comprises constraints for expressions. The next two groups have constraints that are checked upon conditions and assignments. Well-formedness rules that impose constraints for each node are given within the fifth group. Finally, the last group contains well-formedness rules for DMM rules.

For all groups, the constraints that belong to these groups and a short description defining each constraint are given in Table 8. The number given within this table is used later on to directly refer to a particular well-formedness rule. The last column of this table contains some comments on why this well-formedness must be fulfilled. Furthermore some comments about the dependencies of the different well-formedness rules are given. Each constraint itself is formulated informal but precise. The according implementation specifies each implementation unambiguously.

No	Well-formedness Rule	Comments
(1)	The first group contains well-formedness rules that are checked on identifiers.	*An identifier refers to an attribute that belongs to a node. Refers means in this case that it points to or identifies an attribute that belongs to a particular node. The attribute is called target attribute and the node to which the attribute belongs is called target node.*
(1.1)	An identifier must refer to an existing attribute.	*This is the most important well-formedness rule. If the target node or the target attribute cannot be*

No	Well-formedness Rule	Comments
	If the identifier `x.y` of an attribute consists of two sub identifiers, the first sub identifier `x` must refer to a node with the name `x` that must exist within the rule. The second sub identifier `y` must denote an attribute `y` that belongs to the type of the node with the name `x`. If the identifier `y` of an attribute does not consist of two sub identifiers, then the attribute `y` must belong to the type of the node, to which the expression containing this identifier belongs.	*uniquely identified, all further checks could not be performed.*
(1.2)	If the new value of an attribute should be used within an expression of a rule, there must be an assignment defining the new value of the attribute within the same rule.	*The new value of an attribute does only make sense if the new value is defined somewhere in the rule. Therefore, an assignment defining the new value must exist. This means that an attribute only has two values if there is an assignment defining the new value of the attribute.* *Although there might be the possibility to take the old value, if the particular assignment cannot be found, this well-formedness rule is kept in order to make the DMM rules less error prone.*
(1.3)	An identifier within an expression cannot refer to an attribute that belongs to a universally quantified node. This well-formedness rule does not affect identifiers that are used within expressions that itself belong to a quantified node of the same quantification cluster like the universally quantified node to which the referenced attribute belongs.	*A whole quantification cluster is matched to one particular structure within the host graph. Therefore, the expressions defined on nodes that belong to the same quantification cluster like the universally quantified node, to which the attribute belongs, can still refer to this particular attribute.* *The attribute is unique within the matched structure found within the host graph. But it is not possible to determine which attribute is meant, if the expression that contains the identifier is defined on a non-quantified node or a node that belongs to a different quantification cluster.*
(1.4)	An identifier within an expression cannot refer to the value of an attribute that belongs to a nested node. This well-formedness rule does not affect identifiers that are used within expressions that itself belong to a nested node of the same nested cluster like the nested node to which the referenced attribute belongs.	*Similarly to (1.3), the whole nested cluster is matched to one particular structure within the host graph–or, of course, may be to several of these structures. Thus, within a structure the referenced attribute is unique and can be used for calculations.* *Nevertheless, it is not possible to refer to an attribute that belongs to a nested node from expressions defined on universally quantified nodes that belong to the same quantification*

No	Well-formedness Rule	Comments
		cluster. This results from the fact that first the uqs cluster matches the host graph and thereafter the statements defined within the nested cluster are checked for the already matched structures from the host graph. Accordingly, the value of an attributes of an object that is matched by a nested node is not available when the uqs cluster is matched.
(1.5)	An identifier within an expression can not refer to attributes belonging to a nodes with the role `not exists`. This well-formedness rule does not affect identifiers–that refer to the old value of attributes–used within expressions on nodes that belong to the same not exists cluster like the target node itself.	*It cannot be possible to refer to attributes, respectively to objects that do not exist. The exception results from the possibility of checking, if a particular structure matched by the not exists cluster does–on the whole–not exist. Therefore, conditions can be expressed on not exists nodes and checked if any structure in the host graph fits the given not exists cluster and the according conditions. If such a structure can be found then the rule is not applicable.* *Actually this well-formedness rule is partially ensured by other well-formedness rules and therefore is therefore ensured for identifiers referring to the new value of attributes because of the well-formedness rule (5.2) and (1.2).* *The first one states that nodes with the role not exist are not allowed to have assignments and the second one that new values of attributes might only be used if there is an assignments defining the new value of the attribute. Nevertheless, this well-formedness rule specifies that neither new– nor old–values of attributes of not exists nodes are allowed to be referenced if the node that contains the expression and the target node do not belong to the same not exists cluster.*
(1.6)	An identifier within an expression cannot refer to the old value of attributes belonging to nodes that have the role `create`.	*This well-formedness rule results from the simple fact that objects defined by nodes with the role create are created on application and do not have an old value for their attributes in the host graph.*
(1.7)	An identifier within an expression cannot refer to the new value of attributes belonging to nodes that have the role `destroy`.	*Because the objects to which nodes with this role are matched, are destroyed on application of the rule, it is not possible to refer to the new value of attributes of these nodes. Similarly to (1.5) this well-formedness rule is ensured by the well-formedness rules (5.3) and (1.2). But, just like in (1.5), it is explicitly given.*
(2)	This group consists of well-formedness rules that are checked upon operations.	*An operation is an expression that contains at least one subexpression and an operator.* *These well-formedness rules can only be checked for fulfillment if the data type for all the attributes*

No	Well-formedness Rule	Comments
		is clear. Thus, all the identifiers must refer to an existing attribute such that this well-formedness rule can be checked. This results in a dependency between (1.1) and (2.1).
(2.1)	For each operation used within an expression the data types of the operands must fit to the expected data type of the operator.	*The signature of the possible operations can be seen in Table 6 on page 24.* *Basically, this well-formedness rule states that there must be a legal operation given in Table 6 having the same operator and the same data types of their input operands as the operation used within the entered expression.*
(3)	Well-formedness rules that are checked upon conditions are given in the following group.	*A condition consists of exactly one expression. Therefore, the well-formedness rules regarding conditions have to be checked after the well-formedness rules for expressions have been checked successfully.*
(3.1)	Identifiers used within conditions can only refer to the old value of attributes.	*This well-formedness rule results from the fact that conditions are checked before the application of a rule and new values are not known prior to the application of the rule.*
(3.2)	The result data type of the expression used within a condition must be `boolean`.	*This follows from the fact that conditions can only be evaluated to the boolean values true and false.*
(4)	This group consists of well-formedness rules that are checked upon assignments.	*Assignments consist of an identifier on the left-hand side and an expression denoting the new value of the attribute on the right-hand side. Therefore assignments can only be checked after the successful evaluation of the well-formedness rules given in group (2)*
(4.1)	An assignment must have an identifier referring to the new value of an attribute on its left-hand side.	*This means that every identifier referring to an attribute must end with the ' sign on the left side of an assignment.* *Although this well-formedness rule could also be ensured by the grammar it is defined to be a well-formedness rule. The reason for this is the simplicity of the production (1) given in Table 7 on page 26. This production states that an assignments consists of an arbitrary identifier on its left-hand side.*
(4.2)	An assignment can only have an identifier on the left-hand side, whose target node is the node that contains the assignment. Thus, assignments can only define the new value of attributes that belong to the node to which the assignment itself belongs.	*This well-formedness rule ensures some kind of enclosure and membership of assignments.* *Thereby, an assignment is somehow bound to the DMM node on which it is defined.*

No	Well-formedness Rule	Comments
(4.3)	The data type of an expression that is situated on the right-hand side of an assignment must fit to the data type of the attribute to which the new value is assigned.	*Fit means that the data types have to be equal. Therefore, it is not possible to assign values to an attribute that would not fit to the data types of these attributes.*
(5)	The following group contains constraints that are checked per node.	*A node can normally have an amount of conditions and assignments. Nevertheless, some well-formedness rules for this construct result from the role of the node and have to be fulfilled such that the whole DMM rule is well-formed.*
(5.1)	It is not allowed to define two or more assignments within one node that have an identifier referring to the same attribute on their left-hand side. Thus only one assignment to each attribute is possible within one rule.	*This results from the TVC–see 3.1 on page 15. If it would be possible to define more than one assignment to the same attribute, there would be more than two values for each attribute within one application of a rule.*
(5.2)	It is not possible to define assignments within nodes that have the role `not exists`.	*Because the according objects are not allowed to exist in the host graph it does not make sense to define assignments on the nodes that are matched–respectively not matched–to these objects.*
(5.3)	It is not possible to define an assignment within a node that has the role `destroy`.	*The objects to which these nodes are matched are destroyed on application. As a conclusion, it does not make sense to assign new values to the attributes of these objects.*
(5.4)	The nodes with the role `create` cannot contain any conditions.	*Conditions are a part of the left-hand side and creations of objects are part of the right-hand side of two sided rules. Therefore nodes having the role create cannot have any conditions.*
(6)	This last group consists of well-formedness rules that are checked upon a whole DMM rule.	*The following well-formedness rules can only be checked successfully if nearly all the other well-formedness rules were checked successfully before. Especially in the case of the well-formedness rule (6.1), all the assignments and identifiers must have been checked successfully before.*
(6.1)	All assignments of a rule must be arrangeable in such a way that they can be evaluated in a sequential way. This means, if an assignment `a` uses the new value of an attribute that is defined by an assignment `b`, there must be a possibility to evaluate assignment `b` before the assignment `a`.	*This well-formedness rule means: Define the following graph G=(V,E). Define an assignment "a" to be a vertex in that graph. If the assignment "a" uses the new value of an attribute defined by the assignment "b" then create an edge from "b" to "a". If this graph is cycle free then there is a sequential order in which the assignments can be evaluated.* *Cycle freeness can be checked with a topological sorting of the vertexes of the created graph.*
(6.2)	It is not possible to define any	*This results from the reason that Premise rules are only used in order to check if a rule can be*

No	Well-formedness Rule	Comments
	assignments within Premise rules.	*applied. They must not modify the host graph in any way.*

Table 8: Table 8: The list of well-formedness rules

3.1.2 Extension of the DMM Metamodel

As already mentioned, the Dynamic Meta Modeling technique uses itself a metamodel to define all the possible DMM rulesets. A DMM ruleset is nothing else than an instance of this metamodel.

The DMM metamodel is implemented as an Eclipse Modeling Framework *(EMF)* Model. For now, such a model can be simply represented by UML class diagrams. Thus, let us assume for simplicity that a DMM metamodel is a simple UML class diagram. Nevertheless, subsection 5.1.1 on page 53 illuminates EMF and defines how the DMM metamodel relates to EMF.

A whole description of the current DMM metamodel can be found in [5]. Therefore, it is not repeated at this place. The metamodel of DMM was extended by two major independent parts. The first one is the introduction of conditions and assignments that are described in subsection 3.1.2.1. The second part, the addition of a possibility to abstract expressions, is depicted in subsection 3.1.2.2. Each of these subsections contains a small overall description of the changes and additions to the metamodel followed by a part for each important type that was added to the DMM metamodel.

3.1.2.1 Conditions and Assignments

There are two additions that were made in order to represent the new constructs that were attached to DMM, namely the classes `Condition` and `Assignment`. Their relationship to the other classes of the DMM metamodel can be seen in Figure 8. This figure shows only an extract of the whole DMM metamodel. Similarly to the fact that conditions and assignments belong to a node, the two new classes `Condition` and `Assignment` have an association to the class `Node`. The association with the diamond shape is called a composition in terms of UML, indicating that a condition or an assignments belong to at most one node. Because a node can have arbitrary many conditions and assignments, these associations have cardinality `0..*`.

Furthermore a whole new package can be seen in the given figure. This package, called `expression`, contains all the classes that are needed for the abstract representation of the expressions that belong to either a condition or an assignment. All the types that belong to this package are described in the following subsection 3.1.2.2.

Condition. An instance of this class represents a simple condition that is checked for fulfillment while matching DMM rules into the host graph. The `Condition` class has a composition with cardinality `1..1` to class `Expression`, representing the fact that a condition consist of exactly one expression. The prefix `expression::` of type `Expression` indicates that this class is not defined within the same package as the type `Condition`. It is contained by the `expression` package and therefore described in the following subsection.

Assignment. Instances of this class represent assignments. Similarly to an assignment that consists of two parts–an identifier referring to an attribute and an expression defining the new value of this attribute–the class `Assignment` has an compositions, one to the class `AttributeExpression` and the other one to the class `Expression`. An `AttributeExpression` represents an identifier and will be depicted in detail in subsection 3.1.2.2. The composition to the `Ex-`

pression class states how the new value will be computed on application of the rule. Both compositions have the cardinality 1..1, as can be seen in Figure 8.

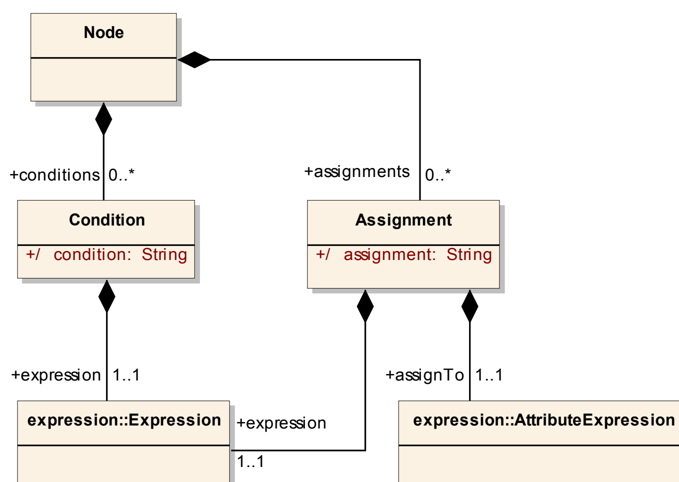

Figure 8: Overview of the modifications of the DMM metamodel

Both types, the type Condition and the type Assignment, have an additional attribute of the data type string. In the case of the class Condition it is called condition. Similarly, in the case of the class Assignment it is accordingly named assignment. The reason for the introduction of these two different attributes is described in detail subsection 5.2.5 on page 65. The values of these attributes are string representations of the according instances of the type Condition or respectively Assignment.

3.1.2.2 Expression Package

Instances of classes that belong to this package together abstract expressions that are entered as a simple string. By this abstraction, it is possible to operate on the instances that represent the expression without having to work with simple strings. All the classes concerning the newly created expression language were separated in an own package in order to increase the encapsulation and guarantee the separation of concerns. This package can itself be reused for the definition of other expression languages within other metamodels if necessary.

The structure of the package can be seen in Figure 9. It can be divided into three subparts. The first one concerns the Expression class and all its subclasses, the second one concerns the class Literal and the according subtypes and finally the Operator class, generalizing the classes UnaryOperator and BinaryOperator.

Expression. The Expression class is the superclass for all the types of expressions that can be used recursively to define complex expressions. These subtypes are AttributeExpression, LiteralExpression and OperationExpression.

OperationExpression. This type represents a simple operation. The recursive manner of nested operations results from the composition between the OperationExpression to the type Expression having cardinality 1..2. This composition is called subexpressions. It states that every instance of the type OperationExpression can have either one or two subexpressions– which are themselves nothing else than instances of the type Expression.

Every instance of the type OperationExpression links to exactly one instance of the class Operator that contains the attribute symbol. This value of this attribute represents the operator of the operation. According to the number of subexpressions of the operation, this operator is an

instance of the class UnaryOperator or BinaryOperator. Besides the attribute symbol of the class Operator, every instance of the type Operator carries another piece of information, given by the attribute priority. This attribute contains the priority of the operator. Although the priority does not change for a given operator, it is automatically deduced from the given grammar and set to the value given in Table 6 on page 24. This ensures the possibility to unparse a given instance of the type Assignment or the type Condition and thus create a string representation of these instances again. How exactly this is achieved and in which way the attribute priority is used for the unparsing is described in subsection 5.2.2 on page 60.

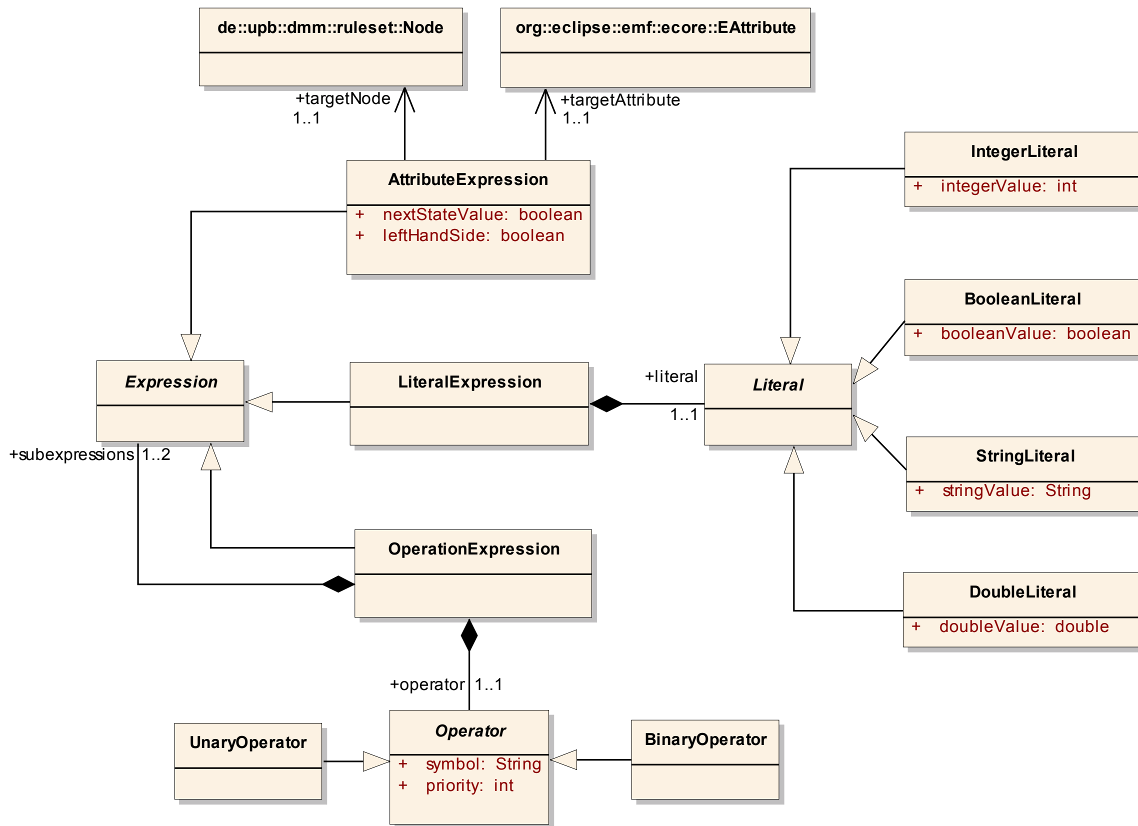

Figure 9: The metamodel for the expression language

LiteralExpression. If a literal is used within an expression, it is represented by an instance of the class LiteralExpression. Since this is a subtype of Expression, it can be used in every place where an instance of the type Expression is expected. Therefore, it can be used as an operand within an operation.

An instance of the type LiteralExpression links exactly one instance of the Literal type. The class Literal itself is abstract and therefore only instances of the types IntegerLiteral, BooleanLiteral, StringLiteral or DoubleLiteral can be referenced by instances of the type LiteralExpression. The subtypes of the class Literal hold the according value of the literal.

AttributeExpression. Finally, instances of this class represent identifiers, i.e., references to attributes. Similarly to instances of the class LiteralExpression, objects of this class can be used in every place where instances of the type Expression are expected. Therefore, an operation can consist of instances of AttributeExpression as well.

This class has attributes in order to store different pieces of information. Furthermore, it has two associations to other objects of types from the DMM metamodel.

The nextStateValue attribute, having the data type boolean, results from the TVC–described in subsection 3.1 on page 15–that defines if the identifier refers to the new or to the old value of an attribute. The attribute leftHandSide, also having the data type boolean, denotes if the AttributeExpression belongs to the left-hand side of an assignment or if it is used within an expression, belonging to either the right-hand side of an assignment or a condition. There is a slightly different meaning of these two kinds of instances of the type AttributeExpression. If this attribute has the value true, it determines that the new value calculated by the expression is assigned to the attribute referenced by the AttributeExpression. If the leftHandSide attribute has the value false the AttributeExpression is used within an expression referring to an attribute whose value will be used for the calculation.

Finally, this class has two associations to the types Node and EAttribute[10]. One instance of each type together define the attribute that is identified by the instance of the type Attribute-Expression. Similar to an identifier, having a target attribute and a target node, these two associations of the type AttributeExpression are called targetNode and targetAttribute. They are tried to be set within the process of parsing the expression string and converting it to an instance of the type Condition or Assignment. If the identified attribute is present within the rule, a link to the instance of the type Node to which the referenced attribute belongs and a link into the runtime metamodel to an instance of the type EAttribute is set.

If the entered identifier does not refer to a particular attribute within the given rule then the according instance of the type AttributeExpression is visually represented by the string null, having the targetNode and the targetAttribute set to the value null.

3.2 Visual Extension of DMM Rules

Some hints on the visual representation of the new constructs within DMM rules were already given in the examples of subsection 3.1 starting on page 13. These hints shall now be combined and summarized in order to present the visual appearance of the different new constructs.

There is one main requirement affecting the extension of the visual representation of DMM rulesets and DMM rules. Due to the fact that DMM rules are expected to be used by users with advanced knowledge in object oriented programming, like knowledge about objects, methods, classes, class diagrams and object diagrams, the syntax for the newly introduced constructs must be intuitively usable for such users. This means that properties and visual representations of object diagrams or class diagrams have to be reused in a similar fashion or have to be changed to a particular extent in order to guarantee the usage in a familiar way.

Therefore, the current visual representation of DMM rules that resembles the UML object diagrams is extended with notations that were taken from UML class diagrams. Because conditions and assignments belong to nodes, the graphical representation of nodes is extended by two container. These container look like the container of classes comprising attributes and methods.

The first container of a node comprises the set of conditions. The second one accordingly is the container for the set of assignments. As a result, each node that contains either conditions or assignments would look pretty similar to the following rule shown in Figure 10 that was created by the extended GMF-based graphical editor. Attributes are not presented in a special container within the DMM rules as they are known from the underlying runtime metamodel. Thus, no

10 A runtime metamodel is defined by an instance of the Ecore metamodel–see section 5.1.1 on page 53 for a detailed description. One particular attribute that belongs to a type within this runtime metamodel is an instance of the type EAttribute. Therefore, a link to an object of this type is used to refer to a particular attribute in the runtime metamodel. The same technique is used for the relationship between a node and its type–represented by an instance of EClass–or an edge and its reference–represented by an instance of the type EReference.

space is wasted for redundant pieces of information.

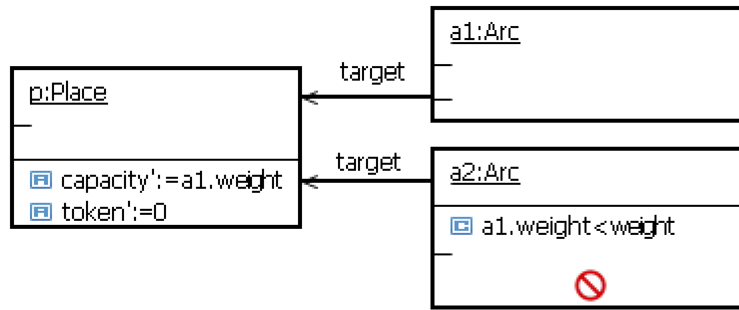

Figure 10: The new visual representation of nodes, conditions and assignments

The given example rule visualized in Figure 10 is based on on the extended runtime metamodel from the running example. It states that the capacity of a place is set to the maximum of the weights of the incoming arcs. This figure demonstrates different visual aspects at the same time.

On the one hand, nearly no changes are made to nodes that do not contain any conditions or any assignments. Only two small lines on the left-hand side of the node with the label `a1:Arc` indicate the two additional containers that can be used for the assignments and conditions. If at least one condition or assignment is added to a node, the line above the according container is visualized by a line that separates the container from the other parts of the node. Thus, it can be easily distinguished whether the node contains a condition or an assignment.

Furthermore, the two new symbols, containing the capital letter `C` or respectively the capital letter `A`, indicate whether the node contains a condition or an assignment. Thus, the danger of confusion is reduced to a minimal level.

These two minor changes to the notation of DMM rules already contain all the modifications to the visual appearance of DMM nodes. Therefore, every user of DMM that is familiar with the old notation should not have any problems to use DMM rules with the new representation of nodes–visualizing conditions and assignments.

4 Implementation of the Expression Language

This chapter describes the efforts made in order to implement the expression language that was specified in subsection 3.1.1 starting on page 19. The implementation of the expression language was not completely created from scratch by writing a complete piece of software; instead, a framework was used that facilitated and accelerated the implementation.

This framework–called JavaCC [13]–is a code generation tool for generating Java code especially for the use within compilers and other programs that handle and analyze sequences of characters. Although there are different other frameworks that solve similar problems, this framework was taken because of its widespread usage and its technological maturity.

First of all, this chapter gives a brief introduction to the JavaCC framework in section 4.1, including some hints on the generated software components. Thereafter, the concrete implementation of the described expression language is given in section 4.2. This section is itself divided into three subsection, similarly to the three steps taken for developing an expression language. The first two subsections describe the implementation of the regular expressions and the grammar. In addition, they contain some pieces of information about the syntax that has to be used within the JavaCC framework. The last subsection depicts the implementation of the well-formedness rules given in Table 8 on page 32.

4.1 Introduction to JavaCC

JavaCC is a code generator for two different kinds of software components [14]. Both software components together are intended to create an abstract representation of an input sequence of characters. The way of abstracting a sequence of characters can be divided into two different steps, corresponding to the two different kinds of software components that are generated by JavaCC. These two software components are called lexical analyzer and parser [15].

The two different software components and their mode of operation can be seen in Figure 11. The big rectangles represent the software components and the small circles represent elements that are created while the generated code is executed. The different parts of the figure will be depicted in the following paragraphs.

Lexical Analyzer. The first step of abstracting the input sequence of characters is to create groups consisting of parts of the sequence of characters. This is done according to a special scheme. The scheme is described by regular expressions and defines how many characters together form such a group. These groups of characters, build upon regular expressions, are called tokens. For instance, a literal having the data type `integer` or an identifier of an attribute can be seen as such a token. The piece of software that is generated by JavaCC to accomplish this task–consume the input sequence of characters and produce according tokens–is called a lexical analyzer. Its functionality can be seen in Figure 11. It scans the sequence of characters from the beginning to the end and produces tokens, like for example the `23`, accordingly to defined regular expressions.

Parser. In the second step, these tokens are grouped according to another special scheme that states how the tokens relate to each other. This special scheme is defined by a grammar. The result of this step is an abstract representation of the sequence of tokens and thus an abstract representation of the input sequence of characters. An example for such an abstract representation can be seen in Figure 11. In the case of DMM and its newly elaborated expression language, this abstract representation is a part of an instance of the DMM metamodel. The description of this part of the DMM metamodel was given in section 3.1.2.2 on page 33. The software component that

achieves this task is called parser.

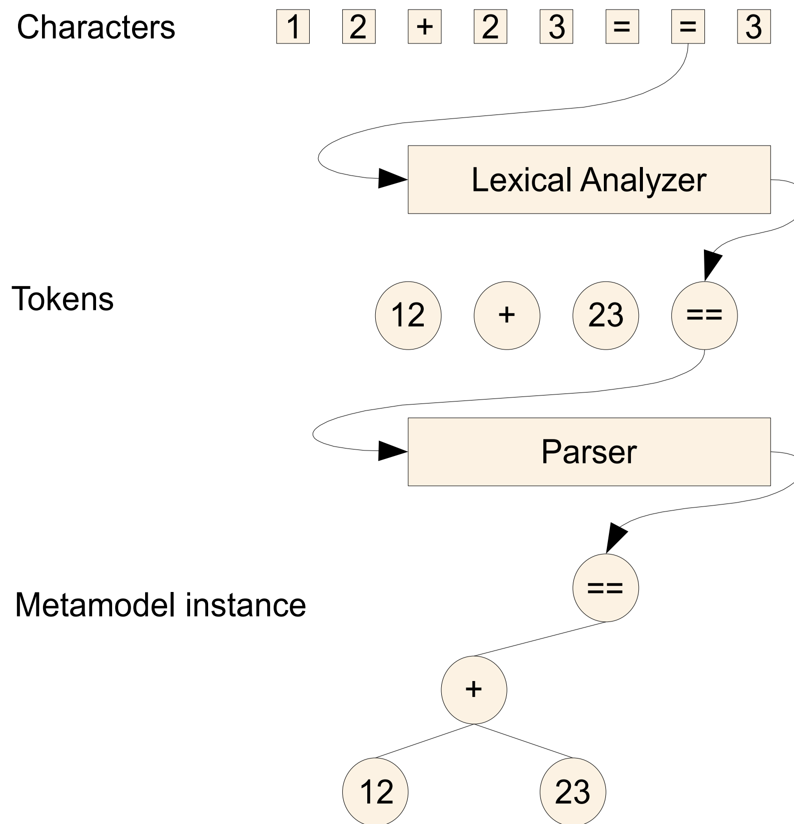

Figure 11: Visual representation of the collaboration between the generated software components

To be more precise, JavaCC creates an LL(k) parser [12]. This kind of parser is a top-down parser that scans the sequence of tokens from left to right. LL(k) is in this case an abbreviation for "scanning from left, using left productions, `k` symbols lookahead".

Using left production means that the left-most nonterminal of the right side of the productions is used in order to derive the sequence of tokens. The productions have to fulfill certain constraints. By this, they create a special kind of grammar, called LL(k) grammar. This grammar is illuminated below.

The term lookahead represents the fact that at any time, the decision which production has to be taken in order to derive the sequence of tokens is based on the next `k` tokens.

In most of the cases an LL(1) parser is sufficient for parsing sequences of characters. These parsers are much faster than other LL(k) parsers with `k` larger than `1`. The elaborated expression language can mostly be parsed by an LL(1) parser. Nevertheless, there are cases where a larger lookahead than `1` has to be used. Within JavaCC, there exists the possibility to enlarge the lookahead only for single productions, thus having the possibility of a large lookahead without loosing the high performance of the parser.

LL(k) grammars. LL(k) grammars are a special kind of grammars that are used by LL(k) parsers. A main characteristic of the productions of these grammars is that the next production is selected only according to the next `k` tokens.

Let us look at an example in order to understand which kind of productions are not allowed by these grammars. Think of a grammar that consists of the two productions `<Tally> ::= <Tally> '|'` and `<Tally> ::= 'tally:'` that are used in order to derive all the words that start

with the word `tally:` and end with an arbitrary long sequence of the symbol `|`[11]. Thus, the given example produces a tally sheet, with an arbitrary amount of elements. The regular expression `'tally:' ('|')*` specifies all the words that could be derived with the help of this small grammar.

But how should the parser derive such a word if the decision of the derivation must be based on the next `k`–a finite number–tokens? It has to scan the whole word in order to know when to use the second production of this example for the derivation of the prefix `tally:`.

Thus, there is no possibility to have a production like `<Tally> ::= <Tally> '|'`, since otherwise the parser could recursively derive the left-most terminal on the right-hand side of the production infinitively often, without having the possibility to decide when the derivation should stop, by only looking at the first `k` tokens delivered by the lexical analyzer.

Therefore, sequences like `'tally:' ('|')*` could not be derived by the parser. But fortunately, grammars that have productions having the same nonterminal on the left-hand side like the left-most terminal on the right-hand side can be rewritten in such a way that they derive the same amount of words without having this left recursive productions. How this is achieved is depicted in section 4.2.2.

4.2 Implementation

Inputs for JavaCC. But what does JavaCC need in order to create a correct lexical analyzer and the according parser for a specific expression language? The only ingredient to achieve a correct generation is a definition file. This file contains four different pieces of information that together specify the lexical analyzer and the parser completely.

The first part contains instructions for the generator that tweak the generated Java code. These instructions allow, for example, to adjust the compilation level of the generated code or the access kind to some important methods–`static`, not `static`–that are generated by JavaCC.

One characteristic of JavaCC is that nearly everywhere within the definition file Java code can be interleaved with the instructions for the JavaCC generator. The second part of the file is only destined for this purpose. Within this part, no JavaCC instructions are allowed. This part can be used to define the class that acts as the accessing point of the parser. Additionally, auxiliary methods and attributes can be defined here which could be used for the definition of the parser later on.

The third part holds instructions and regular expressions for the generation of the lexical analyzer. Here, characters–or even character sequences–are defined that are completely skipped by the lexical analyzer. This is done for example by nearly all expression languages for the white space characters. This means that regular expressions that match these characters or character sequences are not delivered to the parser, but ignored completely. Furthermore, a regular expression for each token is given within this part of the definition file. If such a regular expression matches the input sequence of characters, the according token is created and returned by the lexical analyzer. The syntax that has to be used in order to define such regular expression differs a little bit from the standard notation of regular expressions and is therefore briefly introduced in section 4.2.1.

Finally, the last part of the definition file contains pieces of information that are needed for the generation of the parser. The only possibility to specify how the parser is generated is to define a grammar consisting of the productions for the intended language–in the case of this bachelor thesis the grammar given in Table 7 on page 26. Each production will result in the generation of a

11 These productions are artificial. They are not taken from the elaborated grammar given in Table 7 on page 26.

method with the same name as the nonterminal of the production. A concrete example and the syntax for this part of the file is given in detail in section 4.2.2.

In addition to these descriptions of the implementation of the regular expressions and the grammar for the elaborated expression language, this section covers the implementation of the well-formedness rules given in Table 8 on page 32 in subsection 4.2.3.

4.2.1 Regular Expressions

Within this subsection, a brief exemplary implementation of a regular expression used for the definition of the elaborated expression language is given. Furthermore, some explanations towards the syntax of the definition file and the functionality of the generated lexical analyzer is provided.

On the whole, a regular expression is defined for each kind of token that was defined in Table 4 on page 21. Similar to the reuse of sub regular expressions like it was done in Table 4, JavaCC provides a possibility to create complex regular expressions by the definition of special regular expressions that could be reused in the definition of other regular expressions. Code extract 1 contains the implementation of the regular expression `integer_literal` that was given on page 21 that makes use of this possibility of combination.

```
1. TOKEN :
2. {
3.       < #DIGIT :          ["0" - "9"] >
4. |     < INTEGER_LITERAL : (<DIGIT>)+ >
5. }
```

Code extract 1: Exemplary token definition within JavaCC

Every section within the third part of the definition file for JavaCC has to start with the identifier `TOKEN`. Such a section can contain arbitrarily many token definitions. If more than one token definition is given in such a section, the single token definitions have to be separated by the symbol `|`. A token definition itself consists of the name of the token, followed by a colon and the regular expression for the token definition. Every token definition must be enclosed by angle brackets.

If the name used within the token definition starts with a `#` then this kind of token can be used as a part of a regular expression in order to define more complex regular expression of other tokens.

The syntax of regular expressions differs slightly from the syntax that was given in section 3.1.1.1 starting on page 20. In JavaCC, every terminal must be enclosed by quotation marks. This means that the particular string enclosed in quotation marks states exactly the sequence of characters that has to be read from the input sequence of characters.

The choice of a value from a set of possible values is defined differently as well. The regular expression `(0-9)` containing a choice is defined within JavaCC by the following construct: `["0" - "9"]`. Both notations state that exactly one element out of the given set of elements has to be chosen.

Besides this possibility to define a token, there exists another a far shorter one. It is possible to define tokens without names, only by writing their regular expression explicitly enclosed by quotation marks directly into the production where they are used as terminals. This possibility is used in Code extract 2 within the following section.

Token precedence. But what happens if more than one regular expression can match a prefix of the input sequence of characters? In this case, the generated lexical analyzer takes the regular expression that matches the longest prefix of the input sequence. But, by this, there might still be

the possibility of ambiguities. Therefore, the tokens have some kind of priority within JavaCC, according to their place within the definition file. If more than one regular expression matches the longest possible prefix, the token is created and returned by the lexical analyzer that is found foremost by passing the definition file from the beginning to the end.

Exactly this technique of resolving ambiguities was used in order to define the operators `max`, `min`, and `abs`. Because these three words could be matched by the regular expression `identifier` defined on page 22 they were defined directly at the beginning of the third part of the definition file. Therefore, they have a higher priority. Accordingly, if the longest prefix `max`, `min` or `abs` is found, it is not matched by the `identifier` regular expression[12].

Nevertheless, this approach leads to another problem that has to be solved. The resulting problem is that normal identifiers for attributes with the names `max`, `min` and `abs` are then not matched by the `identifier` regular expression any more. As a conclusion, these three kinds of tokens for operators must be explicitly accepted as names for attributes. This can be done with an additional regular expression that comprises the different possibilities. It would be noted as `(identifier|'max'|'min'|'abs')`.

4.2.2 Grammar

As already mentioned, the fourth part of the definition file for JavaCC includes the definition of the grammar. This grammar is used by JavaCC to generate the parser that is fed with tokens from the lexical analyzer and decides which production to use for the derivation. During the derivation and the usage of the different productions, it is possible to create an abstract representation of the input sequence of characters and tokens.

Production rewriting. As already mentioned in section 4.1, JavaCC does not support productions with left recursion. However, there exists a simple rewriting rule for these productions, such that on the one hand they do not make use of left recursion any more and on the other hand do not change the amount of words derived by the grammar. This rewriting rule is explained exemplarily using the productions given at (2) in Table 7 on page 26.

The productions for the derivation of the nonterminal `<orExp>` are defined as follows.

```
<orExp>
    ::= <orExp> '|' <andExp>
    ::= <andExp>
```

The left recursion results from the first nonterminal `<orExp>` on the right side of the first production. The recursion ends if the nonterminal `<orExp>` is derived to the nonterminal `<andExp>` by the use of the second production. Therefore, every derivation ends with the replacement of the nonterminal `<orExp>` by the nonterminal `<andExp>`. If `<orExp>` is derived to the expression `<orExp> '|' <andExp>` by the first production this results in a further derivation possibility to `<andExp>`. As a result from this analyses, it can be seen that the productions state that an `<orExp>` can be derived to an arbitrarily long sequence of the nonterminal `<andExp>` separated by the symbol `|`. Because of that, the production given above can be rewritten as follows.

```
<orExp>
    ::= <andExp> ('|' <andExp>)*
```

The parenthesis and the star state that the terminal `|` and the nonterminal `<andExp>` can be repeated as often as necessary in order to derive the input sequence of characters. This is exactly the same result as if the other two productions were used for the definition of the derivation of

12 This priority is not related to the priority of the operators defined for the elaborated expression language.

the nonterminal <orExp>. Apparently, these two different definition possibilities are equivalent, and the latter one can be used instead. The advantage of this production is its avoidance of left recursion.

Syntax for productions. Within JavaCC, every production that derives the same nonterminal is given in a notation which is related to the one of Java methods. This notation will be called method-like notation *(MLN)* in the sequel of this subsection. This abbreviation is also used to address a sub part of the fourth part of the JavaCC file, where the definition of all the derivation possibilities for one nonterminal are given. During the generation of the parser, every MLN is directly translated to a normal Java method that is called if the nonterminal has to be derived.

Besides the derivation task of an MLN, it is possible to define instructions that have to be executed on derivation. Thus, it might be for example possible to change the state of a Java object or create new Java objects that would represent the input sequence of characters in an abstract, more operational way. In the case of the elaborated expression language, this sequence of tokens is translated to instances of types defined in the DMM metamodel, thus leading to a possibility to process the input sequence of characters in a much more convenient and satisfying way.

An example of such an MLN is given in Code extract 2. This code extract shows the implementation of the productions given at (2) in Table 7 on page 26. This example does already include the rewriting of the productions such that the resulting grammar does not make use of left recursion.

```
1. Expression orExp() :
2. {
3.      Expression result = null;
4.      Expression subexpression1;
5.      Token t;
6. }
7. {
8.      result = andExp()
9.      (
10.          t = "|" subexpression1 = andExp()
11.          {
12.               OperationExpression opExpression =
13.                    ef.createOperationExpression();
14.               Operator operator = ef.createBinaryOperator();
15.               operator.setSymbol(t.image);
16.
17.               opExpression.setOperator(operator);
18.               opExpression.getSubexpressions().add(result);
19.               opExpression.getSubexpressions().add(subexpression1);
20.
21.               result = opExpression;
22.          }
23.      )*
24.      {
25.          return result;
26.      }
27. }
```

Code extract 2: An MLN that defines how the nonterminal <orExp> should be derived

Every MLN starts with the result type of the generated method, followed by the name of the nonterminal that is going to be derived. All the parts within the inner brackets and the part enclosed by the first brackets contain normal Java code that is injected into the generated code (line no. 2-6, 11-22). All the non Java code is emphasized in the given code extract. If all the intermediate Java code is left out, the rewritten production can be seen easily. Then the example could be reduced to the following lines. This is exactly the JavaCC equivalent of the production that was

given before.

```
Expression orExp() : {
   andExp() ( "|" andExp())*
}
```

Within this example, the second possibility to define tokens–nameless tokens–is used. The regular expression `"|"` represents a token that matches the input sequence of characters. This regular expression must match the input sequence of characters at this place such that the nonterminal `<orExp>` can be correctly derived.

The Java code itself is only present to create a correct instance of the type `OperationExpression`, to set all the subexpressions of the type `Expression` and to create the according instance of the type `Operator`. The attribute `image` of a token given in line 15 returns the string representation of that token. In this particular example, the `image` variable would have the value `|`.

4.2.3 Well-formedness Rules

This subsection covers the implementation of the well-formedness rules given in Table 8 on page 32. Every well-formedness rule results in a constraint that is validated against a given DMM construct. After the successful validation of these constraints it is guaranteed that only well-formed expressions were used across the rule and therefore the instance of the DMM metamodel is well-formed[13].

As already shown, the variety of well-formedness rules given in Table 8 on page 32 have to be validated in a particular order, since they have different dependencies to each other. For instance, the constraint (2.1) of Table 8 cannot be validated unless constraint (1.1) has been validated successfully. Because of these dependencies, some already developed software components for the use with the generated GMF-based graphical editor could not be used for the validation in this case. The EMF Validation Framework [16] for example does not support a specific ordering of the different constraints and can therefore not be utilized for the validation of the well-formedness rules.

As a result, a completely new validation framework was developed from scratch. One main driver during development of the validation framework was the ability to support the traversal of the instance of the DMM metamodel and the traversal of the constraints in a customizable way. In order to hide the newly developed validation framework from the user of the graphical editor, the EMF Validation Framework that is used by the automatically generated graphical editor from GMF delegates the validation of the constraints to this new piece of software. How this is achieved is the topic of subsection 5.2.1 on page 59.

Similarly to the development of the grammar, another main driver for the creation of the validation framework was extensibility enabling the addition of further constraints easily. Hence, a validation framework was developed according to these specifications. The result is a validation framework with a possibility to extend the set of constraints easily and to adjust the traversal of these constraints in a simple way.

The following subsections address the implementation of this framework. Subsection 4.2.3.1 gives an overview of the validation framework. The subsequent sections 4.2.3.2 to 4.2.3.4 sum up the tasks and the implementation of some packages of the validation framework. Within this bachelor thesis only a short overview of the validation framework can be given. A detailed description is given in the generated Javadoc that is delivered along this bachelor project.

Note that the actual implementation slightly differs from the given figures below. This results

13 Assuming that all the other constructs of the DMM ruleset are well-formed.

from the abstraction of details and the goal to give a brief overview of the functionality of the validation framework. For instance, all the packages that are described are sub packages of the `de.upb.dmm.parser` package, but this prefix is left out for the sake of easier comprehension. Furthermore, all the figures are only views on the given validation framework and therefore do not necessarily contain all the methods, attributes and associations owned by the given types.

4.2.3.1 Overview

The most important part of the validation framework is structured into three different packages that are visually presented in Figure 12. The first one, the `validator` package–described in subsection 4.2.3.2–, is the heart of the validation framework. One of its tasks is, for example, the traversal of the model and the constraints and the subsequent storage of the validation results in the `validator.result` package–that itself is described in subsection 4.2.3.4. The necessity to store the validation results–besides other reasons that would be given in the following subsections–evolves from the decoupling of the validation framework from the graphical editor. Thus, when the validation is started from the GMF-based graphical editor, the `validator` package can be queried for the results of the validation, which then delegates the queries to the `validator.result` package. How the delegation of the results to the GMF-based graphical editor is performed is described in section 5.2.1 on page 59.

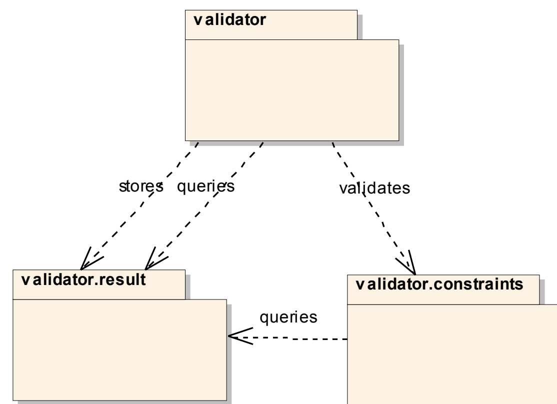

Figure 12: The overview of the validation framework

The different constraints that are validated are placed in the `validator.constraints` package and its sub packages, which are described in section 4.2.3.3. All the constraints have the ability to query the `validator.result` package for the results of the validation of other constraints that were validated before. If the validation framework should be extended by the addition of further constraints, these constraints should be placed within the `validator.constraints` package.

Besides these packages, the validation framework makes use of different other packages and libraries for the achievement of different tasks. For instance, the functionality to search the model for a particular instance or to arrange the constraints according to the defined dependencies is implicitly assumed but not described within this bachelor thesis.

4.2.3.2 Result Package

This subsection deals with the structure of the `validator.result` package and its ability to store, handle and lookup results of the validation process. All constraints that are validated on a particular model object–a model object is nothing else than an instance of a type of the DMM metamodel–produce a particular result if they are validated. The possible results of the validation

are described below. Let us define such a piece of result as the result of a validation step. Therefore, a validation step is the validation of one constraint upon one particular model object.

There are two different reasons why all these results have to be stored during validation of a DMM ruleset. First, as already mentioned, the constraints have complex dependencies, and can therefore not be evaluated independently from each other. The result of a previously validated constraint on a particular model object must be known for the validation of other constraints. These results must be stored.

Secondly, the validation is performed completely at the beginning of a validation process[14]. After the validation of a whole instance of the DMM metamodel has been performed, the GMF-based editor and the EMF Validation Framework have to retrieve the results of this validation. They can query the `validator.result` package through the `validator` package for this purpose.

Figure 13 shows the overall structure of the `validator.result` package. An instance of the `ValidationResultContainer` type is created for each DMM rule, i.e. for any instance of the class `Rule`. Instances of the type `Rule` represents complete DMM rules within the DMM metamodel. It stores a collection of objects of the type `ModelObjectValidationResult`.

ModelObjectValidationResult. A model object, as already mentioned, is nothing else than an instance of a type defined within the DMM metamodel. Because the DMM metamodel is itself an EMF Model–see section 3.1.2 on page 32 for the description of the DMM metamodel, and subsection 5.1.1 on page 53 for the description of EMF–, instances of the types defined within the DMM metamodel inherit the type `EObject`. This interface is shown in Figure 13 on the left side of the figure. Each well-formedness rule, as depicted in subsection 4.2.3.3, is represented by an instance of the type `Constraint`.

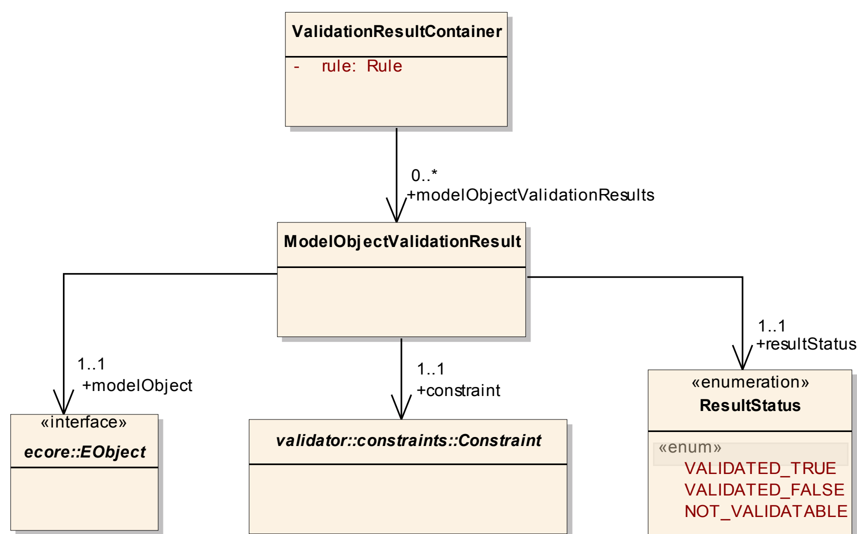

Figure 13: The structure of the validator.result package

The result of each such validation step is stored as an instance called `resultStatus` of the enum type `ResultStatus`. Thus, the overall validation result consists of the results of each validation step. As each validation step can be identified by a constraint and a model object, every object of the type `ModelObjectValidationResult` can be uniquely identified by its constraint and its model object.

ResultStatus. There are different validation results that can result from a validation step. If the

14 This results from the inability of the EMF Validation Framework to traverse the given constraints in a defined order. Therefore, the validation has to be completely accomplished such that afterwards the results can be polled by the EMF Validation Framework in an arbitrary order.

validation step could be performed successfully–i.e., a constraint is fulfilled for a particular model object–the `resultStatus` gets the value `VALIDATED_TRUE`. If a constraint is not fulfilled then the value `VALIDATED_FALSE` is assigned to the `resultStatus`.

Besides the cases that a constraint is successfully or unsuccessfully validated, there exists a third case that states that a particular constraint could not be validated on a model object at all. This third case results from the different complex dependencies between constraints. For instance, it is not possible to deduce the data type of an identifier represented by an instance of the type `AttributeExpression`–given as constraint (2.1) in the Table 8 on page 32–if the identifier does not refer to an existing attribute. This latter well-formedness rule is ensured by constraint (1.1). This results in an invalidatability of constraint (2.1) for the part of the expression that consists of this identifier. This third kind result of a validation step is called `NOT_VALIDATABLE`.

Thereby, it can be seen that every constraint itself has some kind of preconditions for its validation. If these preconditions are not fulfilled, the particular constraint is not validated on a particular model object, thus the validation of this constraint upon the model object can not be performed. The concept of preconditions and dependencies of constraints is depicted in detail the following subsection on page 50.

Due to the introduction of this third value as a `ResultStatus` instance, the meaning of `VALIDATED_TRUE` and `VALIDATED_FALSE` has to be adjusted slightly. They now both mean that all the preconditions given by a constraint are fulfilled and the validation of a constraint upon a model object could be performed–either successfully or unsuccessfully.

4.2.3.3 Constraints Package

This subsection depicts the package `validator.constraints`. Therefore, it illuminates how the dependencies between the different constraints are expressed and how they are resolved. Furthermore, a brief description of the functionality of a validation step is given.

An excerpt of the overall structure of this package is given in Figure 14 on the following page. It contains the classes `ConstraintListContainer` that comprises different instances of the type `ConstraintList` that itself contains arbitrary many instances of the type `Constraint`.

In this package nearly every non abstract class has only one instance of its type. Such an instance is called singleton. There are two reasons why this technique was used for the non abstract classes.

The first one results from the fact that all the objects of these classes are stateless but share a common subset of methods. Therefore, generalization could be used but it did not make sense to instantiate the classes several times. Secondly, by this, an easy possibility is created to find an instance of each class at any place in the program. Why this requirement–finding a particular instance that represents a constraint–is important will be presented soon.

Constraint. The class `Constraint`, or more precisely, every non abstract subtype of the class `Constraint` implements a well-formedness rule as given in Table 8 on page 32. The singletons of these subtypes are arranged in constraint lists represented by singletons of the subtypes of `ConstraintList`. This results in a kind of grouping of the different constraints, like it was done in Table 8. How exactly the constraints are arranged and to which constraint list they are added is described in the following paragraphs.

The singleton of the class `ConstraintListContainer` represents the collection of all the singletons of the subtypes of `ConstraintList`. The task of this class is to arrange the constraints and to put them in the correct constraint list according to particular properties of the constraint.

After the arrangement, this singleton of the type `ConstraintListContainer` can be used to retrieve all the constraint lists together with their constraints that have to be validated on a DMM metamodel instance.

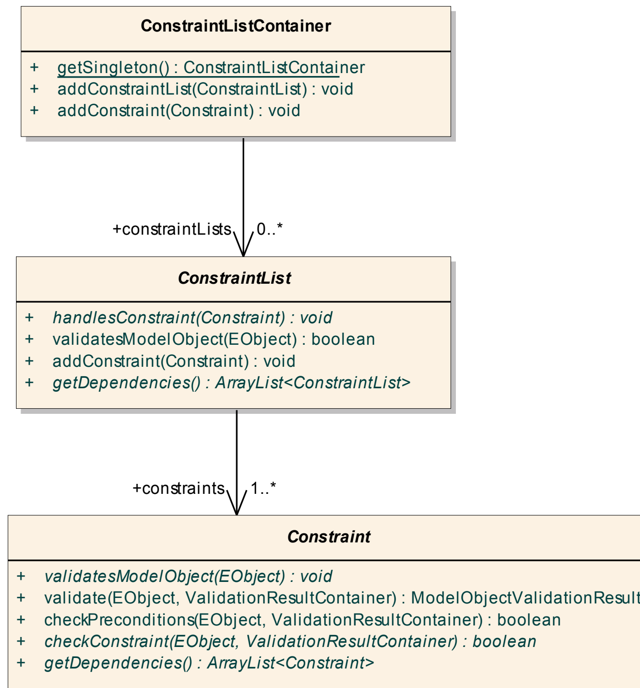

Figure 14: The overview of the validator.constraints package

Each direct subtype of the type `Constraint` determines which model objects their subtypes should validate. Similar to the grouping in Table 8 given on page 32, where each group of well-formedness rules is formulated for a particular part of a DMM rule[15], each direct subtype of `Constraint` represents the generalization of all the constraints of one group. Figure 15, for instance, shows on its right side the relationship between the subtypes of `Constraint` in the case of the well-formedness rule (1.1).

The abstract type `AttributeExpressionConstraint` determines which model objects are validated by the subtypes of this type. In this case, all model objects that are instances of the type `AttributeExpression` are validated by the subtypes of the type `AttributeExpression-Constraint`. Each such direct subtype of `Constraint` encapsulates different helpful methods that are reused by their subclasses. These are methods that are useful for the validation of the particular kind of model object. E.g., the given type holds methods for the comparison of instances of the type `AttributeExpression`.

An example for a concrete type that implements a well-formedness rule is the type `AECAttributeIdentifier` shown in Figure 15. The singleton of this class checks whether given identifiers–represented by an instance of the type `AttributeExpression`–refer to existing target attributes that belong to existing target nodes.

It can be seen that constraints that have to validate a particular well-formedness rule against a special type of model object can be easily added to the validation framework. They only have to subclass the according subtype of `Constraint` and implement the well-formedness rule to be validated.

ConstraintList. For each group of the well-formedness rules given in Table 8 on page 32, one

15 For example the first group of constraints contains constraints for identifiers.

subtype of the type `ConstraintList` was created. This fact can be seen in Figure 15. There, the type `AttributeExpressionConstraintList` as a subtype of `ConstraintList` is created in order to handle all the well-formedness rules formulated upon identifiers. Therefore, this constraint list, or more precisely instances of the type `Constraint` that belong to this constraint list, validate model objects of the type `AttributeExpression`. But, as all the constraints that validate model objects of this type have to extend the class `AttributeExpressionConstraint`, a special relationship between each subtype of `ConstraintList` and each direct subtype of `Constraint` evolves. As a conclusion, the type `AttributeExpressionConstraintList` only manages constraints of the type `AttributeExpressionConstraint`, thus guaranteeing that all the constraints within one constraint list validate the same type of model objects.

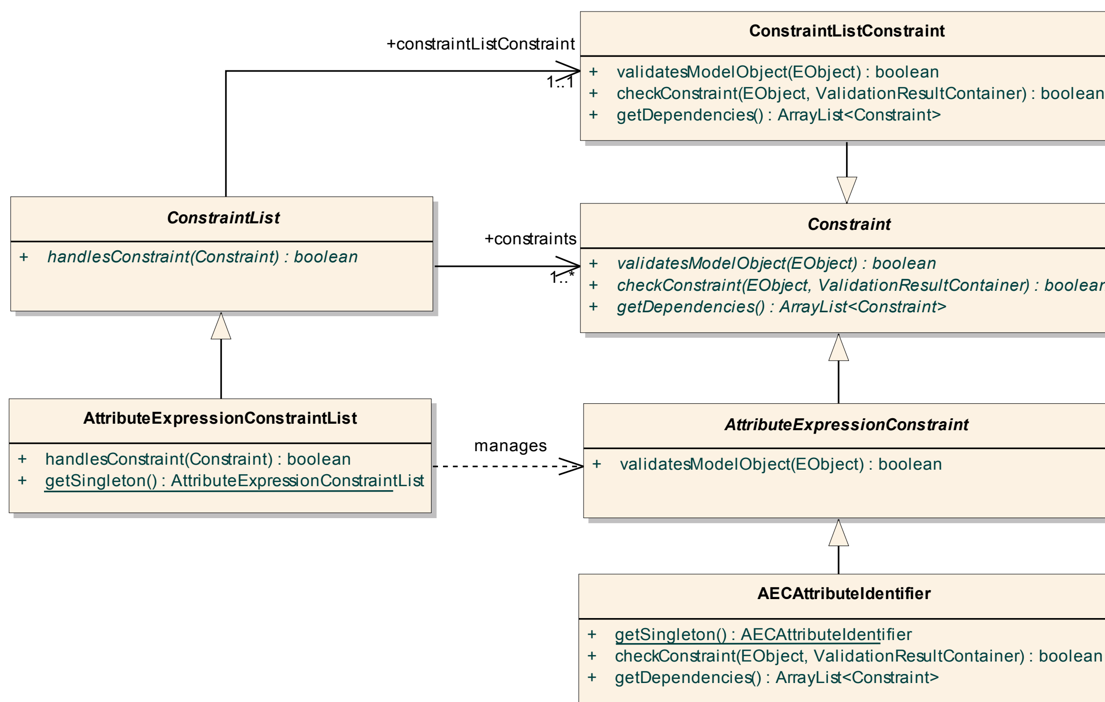

Figure 15: Relation between constraints and constraint lists

In order to arrange the singletons of the subtypes of each direct subtype of `Constraint` to the according singletons of the subtypes of `ConstraintList`, the method `handlesConstraint(Constraint)` belonging to the class `ConstraintList` is used. It decides if a particular constraint belongs to a given constraint list. E.g., the type `AttributeExpressionConstraintList` overrides the method `handlesConstraint(Constraint)` and states that it accepts all the objects of the type `AttributeExpressionConstraint`, resulting in an arrangement similar to the one that was given in Table 8.

ConstraintListConstraint. For each `ConstraintList` there is one special constraint of the type `ConstraintListConstraint`. This constraint does not directly validate a particular well-formedness rule, but checks whether all the constraints arranged into one constraint list were validated correctly upon a given model object. For instance, if a given identifier–an instance of the type `AttributeExpression`–is successfully validated by all the well-formedness rules formulated upon identifiers, the instance of the type `ConstraintListConstraint` is also validated successfully. If at least one constraint within this constraint list is validated unsuccessfully or could not be validated according to the unsuccessful validation of a dependent constraint, this special constraint can not be validated successfully either, since it depends on the

validation of all the other constraints. This results in the ResultStatus NOT_VALIDATABLE that is stored within the validator.result package for this special constraint and the particular model object.

By the introduction of this special constraint, the possibility arises to simply check if a model object fulfills all the well-formedness rules that are formulated upon the type of the model object. This constraint is for instance used in order to check if an identifier, represented by an instance of the type AttributeExpression, is validated successfully by all constraints that validate instances of the type AttributeExpression.

Dependency Concept. As already mentioned, the separate constraints and even the separate constraint lists have different dependencies to each other. Basically, these dependencies can be divided into two kinds of dependencies.

The first kind affects the dependencies between the constraint lists. For example, the group of well-formedness rules that are validated upon conditions can only be evaluated if the well-formedness rules for expressions were validated before.

All these dependencies between different constraint lists are summed up in the object diagram in Figure 16. The names that represent the different groups, i.e. the different constraint lists, are the type names of the according constraint lists. The dependency-links between the objects state that the source singleton of the particular subtype of ConstraintList depends on the target singleton of another subtype of the type ConstraintList.

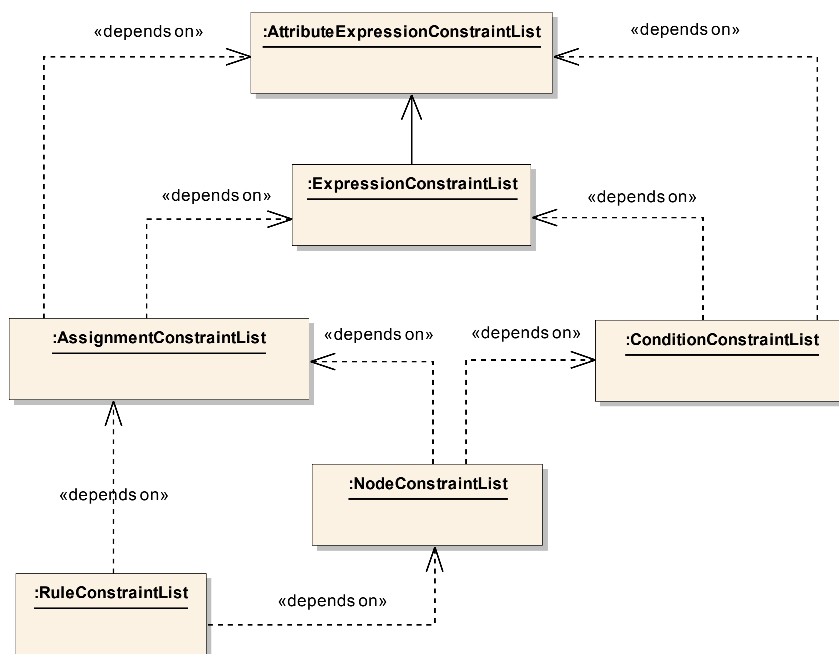

Figure 16: Dependencies between the different constraint lists

The second kind of dependencies results from dependencies between the separate constraints within one constraint list. E.g., all the well-formedness rules given for identifiers depend on the successful validation of the well-formedness rule (1.1) given in Table 8 on page 32. These dependencies can be seen within the Javadoc documentation that was created for the whole bachelor project. The special constraint for each constraint list–the instance of the type ConstraintListConstraint–is added to each constraint list automatically and thus automatically depends on the correct validation of each constraint within one constraint list.

All the dependencies are resolved automatically by the singleton of the type ConstraintList-

Container or respectively the singletons of the different subtypes of ConstraintList. This is done by a topological sorting of the constraint lists within the container, or respectively the sorting of the constraints within one constraint list. After this sorting, the different constraint lists and the different constraints are arranged in a particular order that regards all the dependencies and can therefore be validated according to this order. If such a sorting cannot be performed due to cycles within the dependencies of the different constraints, the singleton of the type ConstraintListConstraint is not created.

Now, the advantage of the singleton concept becomes clear. As every constraint and every constraint list has to decide on which other constraints or respectively constraint lists it depends, it can simply obtain the singletons of the particular types and announce its dependencies.

Validation Step. Some aspects have to be mentioned about one validation step. As already defined, a validation step is the validation of one single constraint upon one single model object. Because of the dependency concept, some constraints cannot be validated unless others were validated successfully. Thus the ResultStatus NOT_VALIDATABLE can be set automatically if some of the constraints on which the constraint under consideration depends are validated unsuccessfully or cannot be validated due to another unsuccessful validation step.

This validation is done by the cooperation of the three different methods validate, checkPreconditions and checkConstraint. If the validate method of a constraint is invoked for a given model object, it checks if all the constraints on which the constraint under consideration depends on were successfully validated. This is done within the method checkPreconditions. If this is the case, the checkConstraint method is invoked and intended to perform the actual validation of the constraint upon the model object.

Sometimes different other preconditions for the validation of a particular constraint exist that differ from the common preconditions resulting from the dependency concept[16]. In these cases the method checkPreconditions is intended to be overridden and the preconditions for the validation of the constraint to be formulated in this overridden method.

4.2.3.4 Validator Package

This package is the heart of the whole validation framework. It is responsible for the traversal of the model and the traversal of the constraint lists and the according constraints in a particular order. The result of each validation step–an instance of the type ModelObjectValidationResult–is retrieved and stored in the according instance of the type ValidationResultContainer.

The structure of this package can be seen in Figure 17. Basically, this package comprises only two different classes. The two other types on the right side of the figure are taken from other packages in order to give an overview of the relationships between the classes of the validator package and classes from other packages.

Validator. One instance of the type Validator is created for each DMM rule that is validated. This instance itself creates an instance of the type ValidationResultContainer in order to store the results of the validation of one DMM rule within this object. Thus, there is exactly one instance of the class ValidationResultContainer for each DMM rule.

One main task of the type Validator is the correct traversal of the whole rule–an instance of the type Rule defined within the DMM metamodel. Because of the various dependencies be-

16 For example, the precondition for the validation of a particular expression is that all the identifiers used within this expression were successfully validated by all the appropriate constraints. This is a dependency that crosses different constraint lists.

tween the different constraints, it is not possible to traverse the whole rule only once. The whole rule is traversed once for each constraint list, obeying the dependencies between the different constraint lists. If a model object is found in the rule that is validated by the constraint list–or to be more precise by the constraints within this constraint list–, all the constraints of the constraint list are validated against this given model object. The separate constraints are validated accordingly to a sorting that obeys all the dependencies between the constraints.

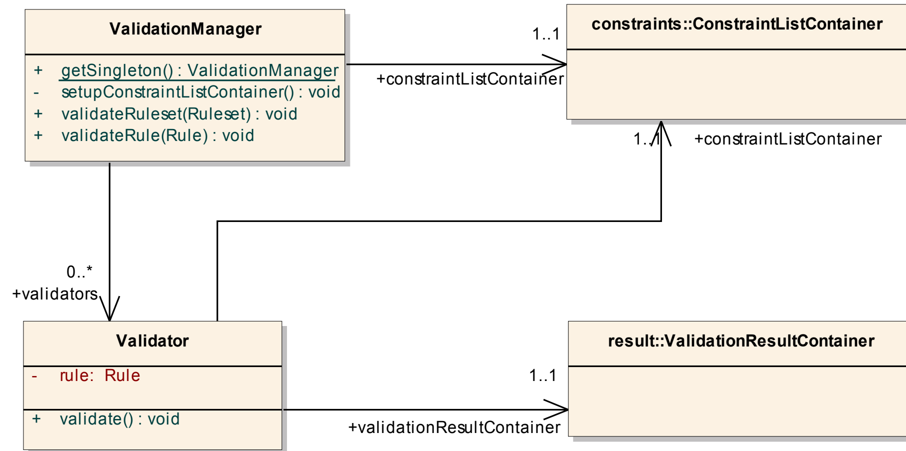

Figure 17: Overview of the validator package

One special traversal of the DMM rule that is being validated has to be performed for the constraint list `ExpressionConstraintList`. Because the well-formedness rule (2.1) from Table 8 from page 32 belongs to this constraint list, it must be ensured that subexpressions are validated by each constraint of the `ExpressionConstraintList` before the composed expression is validated. Therefore, a postfix run over the whole model is performed. This guarantees that the subexpressions of each expression are validated before the composed expression itself.

Validation Manager. The singleton of the class `ValidationManager` tracks the validation process for each DMM rule, or even DMM ruleset. It creates an instance of the type `Validator` for each DMM rule and starts the validation process.

Besides this task, this singleton of the `ValidationManager` handles the creation of the singleton of the type `ConstraintListContainer`. The method `setupConstraintListContainer()` is the place to register additional constraints of the validation framework.

5 Enhancement of the Graphical Editor

This chapter gives an overview of the extensions that were made to the graphical editor. As described in section 1.2 on page 2, this is the second great part of the whole bachelor project. The changes that were made to DMM due to the addition of attributes and familiar concepts were incorporated into the already existing graphical editor.

Since the current graphical editor for the composition of DMM rulesets and DMM rules is composed by the usage of different Eclipse-based frameworks, this chapter starts with an introduction to Eclipse [17] in section 5.1. There, the two most important Eclipse-based frameworks for the achievement of the goals of this bachelor project are illuminated.

Thereafter, section 5.2 covers all the changes that were made in order to create a graphical editor with attribute support for DMM. The solutions for the various tasks and challenges to achieve the goals described in section 1.2 and the visual appearance given in section 3.2 on page 35 are described in this section.

5.1 Introduction to Eclipse

Eclipse is an open source framework and a highly integrated tool platform [18]. This framework itself is intended to be extended by many different other frameworks that enrich Eclipse's functionality. Eclipse and many of these frameworks are hosted by the Eclipse Foundation. The Eclipse Foundation is a non-profit organization in which nearly one hundred companies and many different individual software developers take part.

One of the main reasons for the success of Eclipse–besides the fact that Eclipse is an open source software–is its great opportunity for extensibility. That is why Eclipse can be enriched by different frameworks and accordingly turned into different applications that are appropriate for the achievement of tasks in distinct contexts.

At the beginning of the development of Eclipse it was intended to be an Integrated Development Environment *(IDE)* for Java. But in the course of time it evolved to an extendable framework, where now the characteristic of an IDE is only one side of Eclipse.

But how does Eclipse achieve its goal of flexibility and extensibility? To answer this question, let us focus on the architecture of Eclipse in the following paragraphs. The knowledge about the architecture and its possibility to extend Eclipse's functionality will be reused in subsection 5.1.1 and 5.1.2, where two of the frameworks based on Eclipse are illuminated.

Eclipse's Architecture. Currently Eclipse consists of two parts. The first part builds up the foundation of Eclipse and the second one comprises a variety of different plug-ins[17] that contribute functionality to this foundation [19]. The foundation itself is sometimes called to be Eclipse–as otherwise the term Eclipse covers too many topics and frameworks at the same time. In this bachelor thesis, the foundation will be called Eclipse Core. The upcoming paragraphs are based on and refer to the architecture given in Figure 18.

Eclipse Core consists of two subparts, Equinox and the Eclipse Core Runtime. Equinox is an OSGi standard compliant implementation. The OSGi specification is a layer for the management of bundles on top of the Java Runtime Environment. Bundles are units of files and resources, combined with a description of the bundle. OSGi specifies a possibility to load and unload such bundles at runtime without restarting the managing application. In other words, it specifies the life cycle of bundles and the adjacent management of this life cycle. These bundles are called

17 This term will be described in the following paragraphs.

plug-ins in the context of Eclipse. Every bundle corresponds to exactly one plug-in and vice versa. Therefore, within this bachelor thesis only the term plug-in is used.

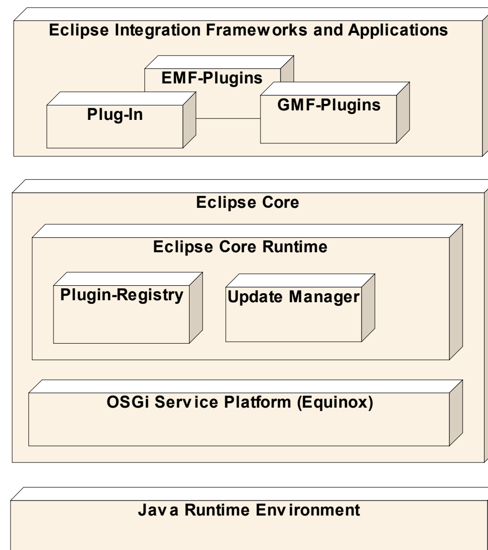

Figure 18: Eclipse's Architecture

The other subpart of Eclipse Core, the Eclipse Core Runtime, contributes functionality for managing the relationships between the plug-ins and a possibility to update these plug-ins. The relationship between the plug-ins is created in terms of extensions and extension points. A plug-in can define extension points that can modify or extend its behavior. These extension points can then be used by other plug-ins. Therefore, they define extensions to these extension points which implement the modified or extended behavior.

All the different plug-ins form the second huge part of Eclipse. Accordingly to the collection of plug-ins, the functionality provided to the user is defined. Thus, Eclipse enables the possibility to be a Rich Client Application *(RCP)* that offers particular functionality for a given domain.

There is another possibility of usage for plug-ins that is used by the graphical editor. Extension points can be used to customize the behavior of generated applications–like for example graphical editors–without touching and rewriting the generated code. How this can be achieved is described in subsection 5.1.2.

Two frameworks that extend the functionality of Eclipse by the use of the plug-in concept–that was described above–are introduced in the following two subsections.

5.1.1 Eclipse Modeling Framework

In order to understand the main intentions of the Eclipse Modeling Framework *(EMF)* another much more abstract concept has to be introduced first. This is the concept of the Meta Object Facility *(MOF)* [20].

Meta Object Facility. MOF is an Object Management Group *(OMG)* compliant standard with the intention to define a common way for modeling metamodels. At first glance, it seems to be confusing that metamodels that itself describe models needed to be modeled by the use of a special modeling language. But this necessity for a common modeling language in order to define metamodels arose from different problems concerning the interoperability between different tools, databases and middlewares that process such metamodels.

MOF can be compared to the grammar definition language BNF for the definition of programming languages. Just as BNF defines a grammar for a particular language, it is possible to define

a metamodel with the help of MOF that specifies a particular modeling language. In order to understand the different kinds of models that are used by MOF, the standard defines a four layer concept that is visualized in Figure 19.

The layer at the top is called `M3` layer. This layer contains the model that defines all the possible metamodels. Thus, it can be defined as meta-metamodel. It defines the concepts of classifiers and references. An instance of this meta-metamodel is a metamodel, like for example the DMM metamodel, or runtime metamodels used by DMM.

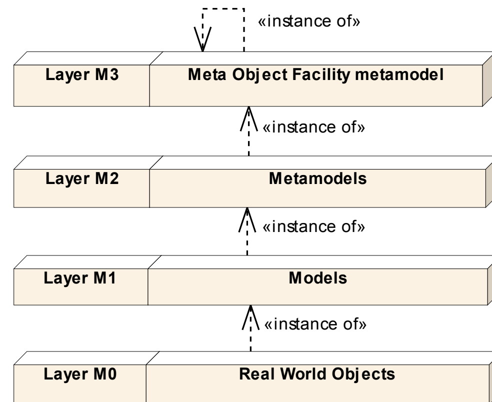

Figure 19: Layers that are defined by the Meta Object Facility

Metamodels itself are situated in the layer `M2`. Furthermore, this layer contains, for example, the UML metamodel that defines all the possible models that can be composed by the use of this modeling language–which are, besides others, class diagrams and object diagrams. Instances of these metamodels–this means the class diagram or the object diagram itself–are situated on the next layer.

The next layer is accordingly called `M1`. It comprises models that are instances of the metamodels of layer `M2`. A particular DMM ruleset, or an instance of the runtime metamodel belongs to this layer.

Finally, layer `M0` is rather technically and contains all the objects of the real world. If, for instance, the model situated in layer `M1` specifies the structure of a real world object, this real world object would belong to this layer at the bottom. For this, think of a real car that is modeled by a type called `Car` within a UML class diagram.

A standard that was specified in the course of MOF is the XML Metadata Interchange *(XMI)* that bases on the Extensible Markup Language *(XML)*. XMI specifies a standard for the exchange of metamodels that are based on the MOF standard between different applications.

Besides the specification of MOF, another specification for the modeling of metamodels was defined by the OMG. This standard is called Essential Meta Object Facility *(EMOF)*, and consists only of a subset of the constructs that can be used by MOF in order to define metamodels. The relation between EMOF and the Eclipse Modeling Framework will be depicted in the following paragraphs.

Eclipse Modeling Framework. The Eclipse Modeling Framework *(EMF)* is a modeling framework and a code generation facility for generating code based on structural models [21]. Such a structural model is called EMF Model. It is usually an abstraction of a situation within a given application domain. In this bachelor project, the DMM metamodel and all the runtime metamodels are such EMF Models. Thus, after the code generation based on the EMF Model, an instance of these generated classes can be seen as an instance of the DMM metamodel or an instance–represented by the host graph–of the runtime metamodel. For simplicity, these instances are called

to be instances of the EMF Model.

An EMF Model itself is nothing else than an instance of the ECore metamodel[18]. This is a metamodel that states all the possible EMF Models. Therefore, it uses types like `EClass`, `EReference` and `EAttribute` in order to create a particular EMF Model. By this, an instance of the type `EClass` can be regarded as a unit of information for the generation of a class.

This concept was already depicted within the description of MOF/EMOF. There, it was said that the MOF metamodel defines all the possible metamodels. The ECore metamodel is itself an implementation of the EMOF metamodel situated on layer `M3`. As a result, EMF is possible to express all the aspects that are claimed by EMOF. Some of these aspects and advantages are depicted in the following paragraphs.

Creation of an EMF Model. The whole process from the creation of an EMF Model to the final Java Code is shown in Figure 20. The different colors that are shown in this figure are reused from now on within the following figures in order to indicate that particular elements belong to particular models. The EMF Model and its components are always painted with this background. The Generator Model and its components are painted with this background.

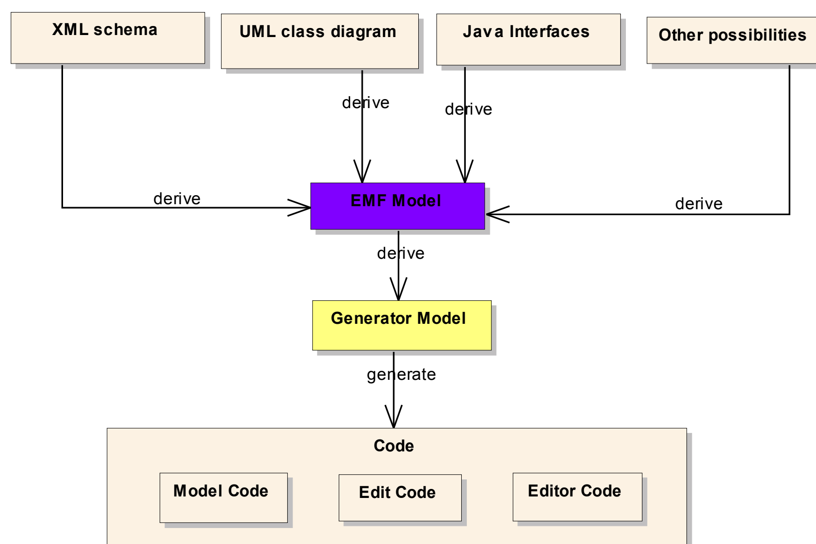

Figure 20: Lifecycle used within Eclipse Modeling Framework

The EMF Model can be directly modeled with an editor that belongs to the Eclipse Modeling Framework. Additionally, it can be derived from different other kinds of descriptions of data structures. These are for example XML Schema Definitions *(XSD)*, Java Interfaces, UML class diagrams or even Files generated by the tool Rational Rose from IBM [22]. All these sources can be used as a foundation in order to derive an EMF Model, like shown in Figure 20.

After the creation of the EMF Model, another model is derived from the EMF Model. As the EMF Model only holds pieces of information about the domain model and since different properties concerning the code generation are not defined by an EMF model, another model has to be build on top of the EMF Model. This model is called the Generator Model. It stores additional pieces of information like for example the package names and instructions concerning the creation of instances of the EMF Model.

These two files can be synchronized with the help of the Eclipse Modeling Framework. Besides the Model Code that can be generated from this file, EMF enables the possibility to create two

18 To be more precise, an EMF Model–a structural model–must therefore be called an ECore Model, since it is an instance of the ECore metamodel. But, as it is often referred to as an EMF Model, this term is used within this bachelor thesis as well.

other code parts, the Edit Code and the Editor Code. The Edit Code is generated for the connection of the Model Code to some kinds of UI components. The Editor Code forms a tree editor in order to modify instances of the Model Code.

EMF advantages. Besides the code generation facility that is provided by EMF, there are two other great advantages that are offered by this framework. The first one is the possibility to make every instance of the generated code and thus every instance of the EMF Model persistent. This results from the fact that EMF is an EMOF conform implementation. As already mentioned, MOF specifies the possibility to exchange metamodels with the help of XMI. As EMF Models are instances of the ECore metamodel, they can easily be exchanged between different tools. But, as the ECore metamodel itself is nothing else than an EMF Model[19], every instance of an EMF Model can be stored persistently.

EMF Model instances can even be split and stored across different files. The transformation of instances to XML files and back again and the resolution of references across different files is handled transparently by the Eclipse Modeling Framework.

Another advantage that is extensively used by the plug-ins that were developed during this bachelor project is the notification concept. Every instance of every generated type can be observed for changes and notifies the observer according to theses changes. The observers are instances of the type `Adapter` within EMF. The objects that notify these instances must implement the type `Notifier`. By this concept it is possible to adapt the behavior of instances of the type `Notifier` by the definition of behavior within the according instances of the type `Adapter`.

More details about this notification concept and how it is used to implement some elements of the intended behavior of the graphical editor is described in subsection 5.2.5 on page 65.

5.1.2 Graphical Modeling Framework

The Graphical Modeling Framework *(GMF)* is, just like EMF, a modeling framework and a code generation facility. But, unlike EMF, it focuses on the generation of graphical editors. This framework is used because the editor that was extended has been developed with the help of GMF.

In order to understand the tasks of GMF, a brief introduction to an underlying framework called Graphical Editing Framework *(GEF)* has to be given [23].

Graphical Editing Framework. On the whole, GEF is a framework in order to create a graphical editor for an arbitrary domain model, and to interact with this editor by the use of a mouse, a keyboard or the Eclipse workbench[20]. The domain model can be an arbitrary collection of classes, and does not necessarily have to be an EMF Model.

In order to enable the developer to create graphical editors effectively, GEF uses the Model View Controller *(MVC)* architectural style. Many auxiliary classes, methods and reusable components are provided in order to reduce the efforts that have to be spent on the creation of a graphical editor.

GEF can be seen as a foundation for GMF. Besides GEF, another foundation for GMF is the modeling framework EMF that was described in the previous subsection. GMF acts as a bridge between EMF and GEF. It combines GEF and uses EMF Models as domain models. Although GMF limits the ability of GEF to work with any arbitrary domain model, it provides the great advantage of code generation for the creation of graphical editors.

19 This shows that the ECore metamodel is bootstrapped. It is expressed by its own elements.

20 The graphical interface of Eclipse is called workbench. Therefore, the interaction through this workbench by the use of buttons and other widgets situated within the workbench is meant.

Besides these two frameworks, GMF takes a lot of other frameworks into account and combines them to achieve its tasks. But, as these frameworks are not used within this bachelor project, their documentation is left out of this document.

On the whole, GMF consists of two big parts, the Generation Framework and the Runtime Environment. The first one is responsible for the generation of the code of the graphical editor. The latter part offers different reusable types, figures and other elements that can be used by different GMF-based editors. The Runtime Environment can also be used without the Generation Framework, if for example the code that uses these types, figures and elements is hand coded.

Generation Framework. Similar to EMF, GMF uses different models in order to specify the code that has to be generated. These models and the process that is needed to create a fully functional graphical editor for a given EMF Models is depicted in Figure 21.

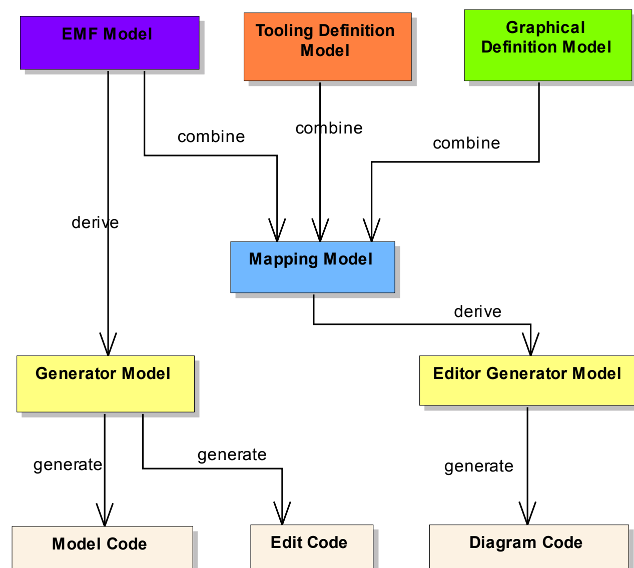

Figure 21: Models used within the Graphical Modeling Framework

The graphical editor is defined basically by four different models. The first one, the EMF Model, was already described in subsection 5.1.1 starting on page 53. The second and the third one describe the graphical figures respectively the tools that are used within the generated graphical editor.

Graphical figures are the visual representation of specific elements of the domain model. For example, think of the visual representation of a DMM node for an instance of a type `Node` defined within the DMM metamodel. Graphical figures are described in the Graphical Definition Model. Components of this model are from now on painted with this background as shown in Figure 21.

Tools can be seen as a possibility to create a graphical figure or an instance of a type defined within the EMF Model, or both at the same time, of course. A particular tool, for instance, is the tool for the creation of DMM nodes and the according graphical figures that visualize the instances of the type `Node`. The model is called Tooling Definition Model. Elements of this model are painted with this background.

These three different models are combined in a Mapping Model that references the elements of the other models. This Mapping Model defines, besides other aspects, which type within the EMF Model is created and which graphical figure should be created upon the usage of a particular tool. The model itself and elements from this model are from now on painted with this background.

The Mapping Model maps these three different pieces of information to each other and thereby

defines the graphical editor. Besides this, the Mapping Model can contain a lot of pieces of information for the specification of the graphical editor, like for example the nesting of graphical elements or audit rules–this are nothing else than well-formedness rules–that have to be fulfilled by the model created with the graphical editor.

The latter functionality will be used to access the well-formedness rules that were defined in Table 8 on page 32. Their implementation was described in subsection 4.2.3 on page 43. How the validation of the audit rules defined within the Mapping Model is forwarded to the newly created validation framework is described in subsection 5.2.1 on page 59.

Finally, the Editor Generator Model can be derived from the Mapping Model. Similar to the Generator Model that is derived from the EMF Model, this model holds additional pieces of information for the generation of the code. This model, for example, determines some of the names of the generated types or functionalities of the Runtime Environment that should be available during the usage of the generated editor. These functionalities are for example the possibility to print the visualized diagram, or to export the visual representation of the model to different image formats.

Finally, based on the Editor Generator Model, the graphical editor is generated into an own Eclipse plug-in. It requires the correct generation of the Model and the Edit Code of the Eclipse Modeling Framework.

Runtime Environment. The Runtime Environment is a collection of reusable components that is used by the generated graphical editor to enrich its behavior. This means that the Diagram Code uses the classes and interfaces provided by the Runtime Environment.

This part of GMF comprises components enabling the possibility to change the colors of the separate graphical figures and their arrangement on the screen. Additionally, it provides the opportunity of printing and the possibility to export the created diagrams to different image formats.

Customizing Code. One main requirement of extensibility and modifiability of the graphical editor is supported by GMF in a very satisfying way: GMF uses the concept of plug-ins and providers extensively. The framework does not access its items, like for example attributes defined within the Model Code, directly, but uses special types, subtypes of the type `IProvider`. Instances of this type deliver the instances of other types that are intended to manage such items. Because GMF enables the possibility to register own providers, an elegant opportunity evolves to customize the behavior of the generated editor.

For the registration of providers GMF makes use of the concept of plug-ins and their extension points or respectively extensions. It allows to register own extensions to given extension point that are regarded on the creation of the graphical editor at runtime. If an extension to a particular extension point is given, it defines which kind of provider can be obtained from the hand coded plug-in that is meant to extend or modify the behavior of the graphical editor.

Let us look at a particular example that will be discussed in detail within subsection 5.2.4. Strings, for example, that are shown within the graphical editor are not directly extracted from attributes of the instance of the EMF Model, but from managing types that offer a parser functionality. Instances of these types can be obtained via a special provider type, called `IParser-Provider`. GMF enables the possibility to register a hand coded parser provider, situated in a separate plug-in. Then this provider can create instances of the special type that implement the `IParser` interface. As a conclusion, this instance will henceforth be used for viewing or editing of strings within the graphical editor.

With this concept, the generated graphical editor can be modified to a certain degree without modifying the generated code. Thus, different aspects of the graphical editor, like the generated

and the self written code, can be separated into different Eclipse plug-ins.

5.2 Implementation

The unextended version of the graphical editor for DMM consists of two parts, two separate graphical editors[21]. The first graphical editor manages the creation of DMM rulesets and the second one the creation of separate DMM rules within one DMM ruleset. This second graphical editor can only be accessed via the first one if a particular DMM rule of the DMM ruleset is selected.

This section gives a brief overview over the modifications performed to this graphical editor. It starts with the description of the access of the validation results that are produced by the newly created validation framework in subsection 5.2.1. Because instances of the types `Condition` and `Assignment` have to be displayed after their creation, a special piece of software, an unparser, was developed. This unparser is depicted in subsection 5.2.2. Subsection 5.2.3 illuminates the addition of conditions and assignments to the graphical editor. It contains the description of the greatest changes to the different models used by the Graphical Modeling Framework. Thereafter, the connection between the parser, the unparser and the graphical editor is illuminated in subsection 5.2.4. Finally, subsection 5.2.5 covers the efforts that were made to keep the visual representation of the conditions and assignments up to date, since several changes within the DMM metamodel instance might affect their string representation.

5.2.1 Access to the Validation Results

GMF uses the concept of audit rules to define constraints that are validated against the EMF Model instance at runtime of the graphical editor. Audit rules are nothing else than well-formedness rules. Every audit rule validates a certain property upon an instance of a type defined within the EMF Model. To achieve this, GMF uses the EMF Validation Framework, as already mentioned in subsection 4.2.3 on page 43. But, due to the inability of the EMF Validation Framework to traverse the audit rules in a particular customized way, a new validation framework that supports this functionality was developed.

Nevertheless, the user should not recognize that the validation is performed by another validation framework than the EMF Validation Framework. Therefore, GMF audit rules are used and their fulfillment is checked by querying the results stored within the newly created validation framework.

Audit rules are defined within the Mapping Model of GMF. An audit rule mainly consist of a description that is shown if the audit rule is not fulfilled, an `ID` and a specification to which elements of the EMF Model instance an audit rule applies. In the case of DMM, it can e.g. be specified whether the audit rules are validated against instances of the types `Condition`, `Assignment` or `Expression`.

Every audit rule maps to exactly one well-formedness rule that was given in Table 8 on page 32. In order to create a direct connection between the audit rule and a well-formedness rule–that is represented by an instance of the type `Constraint` within the newly created validation framework–the `ID` of the audit rule is used. This `ID` contains pieces of information on how to find the name of a particular subtype of the type `Constraint` that implements one well-formedness rule. Because each singleton of the type `Constraint`–or to be more precise of one of its subtypes–implements one well-formedness rule, each well-formedness rule can be uniquely identi-

21 Unextended means that this was the state of the graphical editor at the beginning of this bachelor project that did not offer support for the definition of attributes and the according concepts.

fied by the name of the subtype of the type Constraint.

In order to access the results of the newly created validation framework, the validation, performed by the newly created validation framework has to be done before these results can be retrieved by the EMF Validation Framework. This is achieved with a modification of the generated code for the graphical editor. Therefore, the type ValidateAction for both editors–the one for DMM rules and the other for the DMM ruleset–were modified in order to start the validation process of the newly created validation framework before the validation process of the EMF Validation Framework is launched.

If the validation result for a particular model object and a particular audit rule should be obtained, the according singleton of a subtype of the type Constraint has to be found. Therefore, the according type name of the constraint is deduced from the ID of the audit rule. After that Java's reflection API is used to obtain the singleton of the according Constraint subtype. How this is done exactly is shown in Code extract 3.

```
1. Constraint result = null;
2.
3. Class<?> constraintClass = Class.forName(constraintClassName);
4.
5. Method method = constraintClass.getMethod("getSingleton", null);
6. Object constraint = method.invoke(null, null);
7.
8. if (constraint instanceof Constraint) {
9.       result = (Constraint) constraint;
10. }
```

Code extract 3: Finding a constraint according to the ID of an audit rule

The constraintClassName is a string variable that contains the fully qualified name of the particular subtype of the type Constraint. It is deduced from the audit rule ID with the help of some constants that define the fully qualified name. The according singleton is retrieved by invoking the static method getSingleton, which has to be present within each indirect subtype of the type Constraint. This is the other reason why only one instance of each indirect subtype of the type Constraint is created and always obtained via the getSingleton method.

Thereafter, the singleton of the type ValidationManager can be queried for the result of one validation step that validates one well-formedness rule against a model object. Recall that the singleton of the type ValidationManager delegates this query to the according instance of the type ValidationResultContainer that contains the validation result of the according validation step.

With these modifications, the user does not recognize any differences in the behavior of the graphical editor during the process of validation.

5.2.2 Creation of an Unparser

The process of unparsing means the creation of a string representation for a given model object or a set of model objects. In the case of the DMM, instances of the type Condition and Assignment have to be visually represented by a string within the graphical editor. It is said that the instance of the type Condition or respectively Assignment is unparsed–a string representation is created. The class achieving the unparsing is called ExpressionUnparser.

The main problem in the process of unparsing is to determine where to insert parentheses and where to leave them out. But, it is of course necessary to regard and maintain the priorities of the different operators.

In order to achieve this goal, to solve the problem generically and to match the goal of extensibility of the grammar an additional attribute was added to a type of the DMM metamodel. This attribute is called `priority`. It belongs to the type `Operator` as can be seen in Figure 9 on page 34.

During the creation of an instance of the type `Condition` or `Assignment`–based on an entered input string–, this attribute is set for each instance of the type `Operator` to a value that is automatically calculated. The result of this automatic calculation of the operator's priorities equals the last column from the Table 6 on page 24. But, to accentuate this fact explicitly, the table itself is not used within the implementation, the operator's priorities are derived completely automatically based on the elaborated grammar.

The value of the priority of each opeartor is calculated with the help of a special variable called `runningPriority`. This variable is initialized with the value `1` if one of the productions given at (2) in Table 6 is used to derive the first nonterminal `<orExp>`. The variable `runningPriority` is increased by one whenever other productions–like (3), (4) and so on–are used to derive a nonterminal that was created by the first production.

This process of increasing the `runningPriority` stops when the last production is used. This means, whenever the production (10) `<primaryExp> ::= '(' <orExp> ')'` is used in order to derive the nonterminal `<primaryExp>`. The `runningPriority` variable is reset to `1`, if the nonterminal `<primaryExp>` is derived to the nonterminal `<orExp>`. This results exactly in the same value as the priorities for the operators given within Table 6.

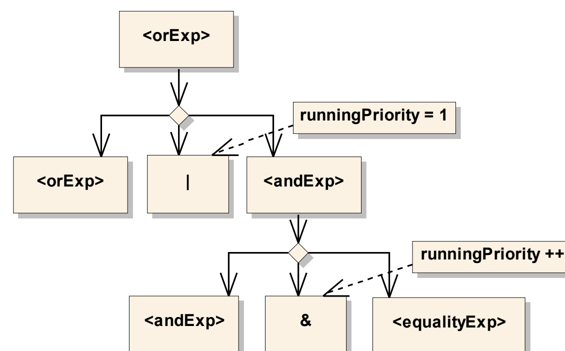

Figure 22: Visualization of the priority calculation

Let us look at an example in order to gain a deeper understanding of this procedure. The following example is visualized in Figure 22. The diamond shape, shown within the figure, represents the derivation of the according nonterminal by the production that can be deduced from the result of the derivation. If the string `a | b & c` is parsed then the first nonterminal `<orExp>` is derived with the production (2) `<orExp> ::= <orExp> | <andExp>`. At this place, the variable `runningPriority` has the value `1`, and this value is stored within the `priority` attribute that belongs to the newly created instance of the type `Operator`. The figure illuminates this behavior by using a dashed arrow that indicates that the current value of the variable `runningPriority` is assigned to the `priority` attribute of the operator. Now, as the `<andExp>` nonterminal is derived, the according production increases the value of `runningPriority` by one and assigns it to a newly created instance of the type `Operator` having the symbol `&` and now as a result a priority of `2`. This behavior is also visualized by a dashed arrow shown within Figure 22.

ExpressionUnparser. Now, as the priority values for each instance of the type `Operator` are given, there is a possibility to decide whether parentheses for subexpressions must be inserted or not. Every operation containing an operator and an arbitrary number of subexpressions can now determine whether any of the subexpressions needs to be parenthesized.

Let us look at a subexpression of a given operation. Let this subexpression itself be an operation and therefore contain an operator. To reduce ambiguity, the former operation is called outer operation and the latter one is called inner operation. If the priority of the operator of the outer operation is smaller–the `priority` attribute has a smaller value–than the priority of the operator of the inner operator, then no parentheses are needed for the latter operation. If, on the other hand, the priority of the operator of the outer operation is greater than the priority of the operator used within the inner operation, parentheses have to be inserted in order to present an expression with the same meaning as the entered expression. If no parentheses were added in this case, the operator of the outer operation would bind one operand of the inner operation more than the other operator. This would lead to a change of the meaning of the outer and the inner operation.

Although this technique seems a little bit complicated, it has one great advantage in comparison to hard coding the priorities of the different operators somewhere in the code. The great advantage is that the priorities of the different operators are not explicitly given, but implicitly obtained from the given grammar.

Thus, if the grammar needs to be extended, for example by addition of the power operator `^`, the priorities do not have to be changed somewhere explicitly. Only the changes made to the grammar would suffice in order to assign the new priorities to the power operator and all the other operators.

Besides this priority concept, another aspect of the unparsing procedure should be mentioned at this place. There are two different kinds of unparsing: One for viewing the condition or assignment and one for editing it. This distinction is made because escape characters for literals of the data type `string` should not be present within the view. But, if the string representation is edited, they have to be added automatically such that the newly entered string can still be parsed correctly.

5.2.3 Addition of Conditions and Assignments

To present conditions and assignments within the graphical editor as shown in Figure 10 on page 36, the distinct models used by GMF, like the EMF Model, the Tooling Definition Model, the Graphical Definition Model and the Mapping Model were extended in different ways.

One not directly apparent extension was performed to the EMF Model. Here an attribute was added to the two types `Condition` and `Assignment`. The additional attribute is called `condition`, respectively `assignment`. Both have the data type `string`. They are intended to carry the string representation of the according instance of the `Condition` or respectively `Assignment`. These attributes are both signed to be volatile, transient, derived and are not changeable. They are simply computed from the instance of the type `Condition` or `Assignment`. This extension of the DMM metamodel was made because of the facilitation of the modification of the other models and the generated graphical editor. GMF provides a lot easier possibilities to display elements that are based upon attributes defined in the EMF Model. Another reason for the introduction of these two attributes is the opportunity to notify that this particular attribute has changed–this means that the string representation of the according instance of the type `Condition` or `Assignment` has changed. How this is achieved is depicted in subsection 5.2.5.

Besides these changes to the EMF Model, the graphical representation of these two new attributes was specified. This was done within the Graphical Definition Model. For the definition of figures that are shown within the graphical editor, instances of the type `FigureDescriptor` are used. In this case, the `FigureDescriptor` instance for the type `Node` defined within the EMF Model had to be adjusted. Two sub figures of the type `Rectangle` were inserted that are intended to hold the set of conditions or respectively assignments–these are the two container

shown within Figure 10 on page 36. These sub figures are managed by instances of the type `Compartment`.

Every string representation of a condition or an assignment is displayed within a `Label` that is managed by an instance of the type `DiagramLabel`. This means that a `Compartment` contains different instances of the type `DiagramLabel`, managing an instance of the type `Rectangle` that displays the different instances of the type `Label`.

Two new tools, one for the creation of conditions and the other one for the creation of assignments, were added to the Tooling Definition Model.

Finally, the Mapping Model was adjusted in order to create a relationship between the different extensions and adjustments that were made to the other models. Figure 23 visualizes the relationships that were created within the Mapping Model in the case of conditions. Nearly the same structure is defined within the Mapping Model in the case of assignments. The colors that were used within Figure 23 show the model in which the particular painted object is defined. The different models were depicted in Figure 21 on page 57.

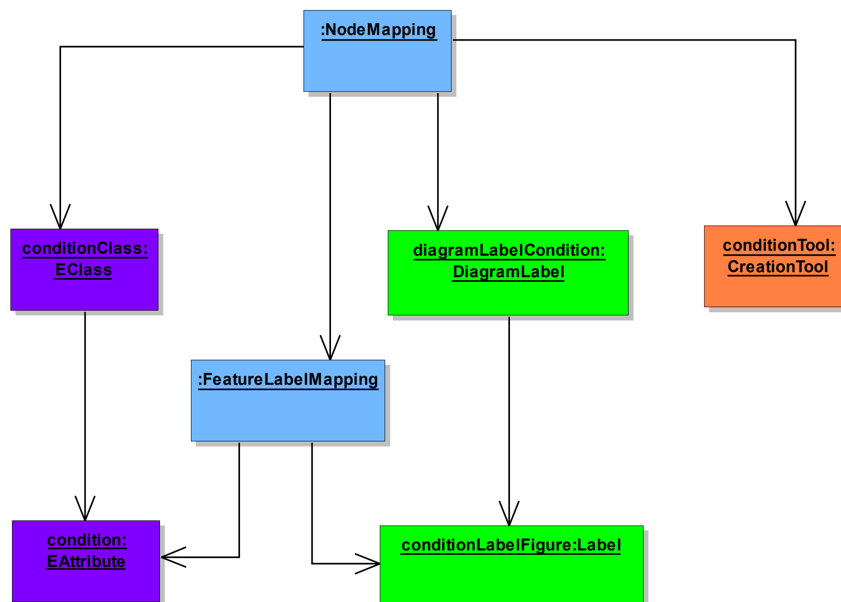

Figure 23: Mapping Model instance for the definition of conditions

The presented instance of the type `NodeMapping` describes what happens if the `CreationTool` for conditions is used in order to create a new condition. Then the `condition` attribute from the EMF Model is displayed in a `Label` that itself is displayed within a `Rectangle`. It can be seen that the instance of the type `NodeMapping` and the instance of the type `FeatureLabelMapping` acts as a kind of glue, bringing all the parts of the different models together. The type `FeatureLabelMapping` can only be used because the `condition` attribute was added to the type `Condition` defined within EMF Model. As instances of the type `FeatureLabelMapping` can only refer to instances of the type `EAttribute`, it is possible to use the `condition` attribute, represented by an instance of the type `EAttribute`, in this case.

Finally, after all modifications of the Mapping Model were performed, the Generator Editor Model was derived. But, as this process is automatically performed, no further changes to the models needed to be made in order to get a graphical editor for DMM with support for attributes, conditions, and assignments.

5.2.4 Access to the Expression Parser and Unparser

Now, as there is a possibility to display the string representation of a condition or an assignment, the parser and the unparser have to be inserted in between the diagram presented within the Eclipse workbench and the instances of the types `Condition` and `Assignment`. Without the following modifications, the labels intended to present the `condition` or the `assignment` attribute would only display the according attribute that itself would not contain any content.

This task can be achieved by an extension of the graphical editor via the concept of extensions and extension points that was described in section 5.1 on page 52. The extension point is called `org.eclipse.gmf.runtime.common.ui.services.parserProviders` and allows to add an instance of the type `IParserProvider`. This instance of the type `IParserProvider` creates an instance of the type `IParser` for a particular type model object–like for example the type `Condition`. Instances of the type `IParser` decide which string representation to choose for a particular part of the instance of the DMM metamodel or create a part of an instance of the DMM metamodel according to an entered string.

This is exactly the behavior that is needed to create a connection between the labels and the conditions and assignments that are going to be displayed. Because of the similarity between conditions and assignments, only the `IParser` implementation for conditions is depicted in the following paragraphs.

The particular implementation of the `IParser` interface for the type `Condition` and its attribute `condition` is called `ConditionParser`. It consists mainly of four different methods. Two of them are needed to obtain a string representation of the particular instance of the type `Condition`, and the other two for the creation of a correct instance of the type `Expression` for a condition based upon a given input string.

The first two methods are called `getPrintString` and `getEditString` and correspond to the two different kinds of unparsing described in the last subsection. The first one displays the string representation of the condition without the addition of escape characters for literals of the data type `string`. The second one adds the escape characters to these literals such that the edited expression can be directly parsed again. Both methods invoke the adequate method defined within the type `ExpressionUnparser`.

The other two methods are a little bit more complicated. They are responsible for the other direction, meaning to forward the entered string to the `ExpressionParser`–this is the class generated by the JavaCC framework–and thereafter set the references from the instance of the type `Condition` to the returned instance of the type `Expression`. This way is more complicated because at this place, the entered string might not necessarily be syntactically correctly formed, causing the type `ExpressionParser` to fail to parse the entered string. Therefore, one of the two methods, called `isValidEditString`, controls whether the entered string can be parsed based on the grammar given in Table 7 on page 26.

If the entered string can be parsed then the other method, called `getParseCommand`, can be invoked. This method is a result form the fact that GMF uses a command framework to modify instances of the generated Model Code. By this, the opportunity is given to undo and redo all the changes made to these instances. In order to use the command framework correctly, a special command has to be created that modifies the condition under consideration. The method `getParseCommand` defined in the type `ConditionParser` creates a command that sets the created instance of the type `Expression` to be the expression of a given instance of the type `Condition`.

Another feature provided by the IParser interface is to find out whether modifications that are performed on instances of the Model Code affect the visual representation of these instances. If they do affect the visual representation, the according parts of the diagram shown within the graphical editor are updated. This mechanism is used in order to keep the string representation of the conditions and assignments up to date. How exactly this is achieved is described in the following subsection.

5.2.5 Keeping Conditions and Assignments up to date

There are many cases that require an update of the string representation of the instance of the type Condition or respectively Assignment. E.g., if a condition uses identifiers that refer to attributes that belong to another node, and in the case of a name change of this target node, the string representation of the condition under consideration has to be updated.

One main task is to notice these changes. But, this task can easily be solved by an advantage of EMF that was already presented. EMF generates Model Code that can notify any instance of the type Adapter of changes that happen to any attributes or references used within instances of the generated Model Code. The concept of notifications was briefly described in section 5.1.1 on page 56 and has been used extensively for the achievement of this task.

First of all, let us collect all the possibilities that might require an update of the string representation of an instance of the type Condition or Assignment. Nearly all of them affect the target node of an identifier used within a condition or assignment. Recall that every identifier is represented by an instance of the type AttributeExpression and points to particular target attribute that belongs to a particular target node.

First and foremost, the target node might be deleted from the diagram. Furthermore, the name or the type of the target node might change. Furthermore, the target attribute might change, either the name, or the data type or it might even be deleted. Finally, the condition or assignment might be dragged to another node. This also affects the string representation, since identifiers that reference attributes that belong to the same node like the identifier itself are visualized without the name of the node.

Changes to the target attribute can be noticed only at startup of the graphical editor. This results from the fact that the runtime metamodel is only loaded once at start up and there is no possibility to change properties of the runtime metamodel while the graphical editor is still active.

More problematically are the changes that can be made within the graphical editor that effect the properties of the target node. Because these changes have to be recognized by the affected conditions or assignments, a special subtype of the type Adapter is created. This subtype is called ConditionUpdater or respectively AssignmentUpdater. These two types are basically equal, the only difference is the addition of the observation of the identifier on the left-hand side of the assignment.

An instance of either the type ConditionUpdater or AssignmentUpdater is added to the list of adapters directly on creation of an instance of the type Condition or Assignment. This instance of the type ConditionUpdater or AssignmentUpdater adds itself to every AttributeExpression, every target node represented by an instance of the type Node, and the according instance of the type Condition or Assignment. Any change of any property of these instances are observed and evaluated. In the case of a possible modification of the string representation of the condition or assignment, the attribute condition or the attribute assignment is announced to have changed. For this, a notification is fired that causes the graphical editor to update the string representation of the according instance of the type Condition or As-

`signment`.

In the case of an invalid modification, like for example the deletion of a target node, the according instance of the type `AttributeExpression` is represented by the predefined string `null`–like defined in subsection 3.1.2.2 on page 34–, indicating that the particular attribute, identified by the `AttributeExpression` could not be found within the given rule.

The technique described above ensures that there never exists an identifier–other than `null`–that refers to an attribute that does not exist within the given DMM rule.

6 Extension of the Transformation to Groove Rules

This chapter describes the third great part of this bachelor project. Like already mentioned in section 1.2 on page 2, this is the enhancement of the transformation of the DMM rulesets to equivalent set of graph transformation rules used by the graph transformation framework Groove [7][24].

This whole chapter depends heavily on the graph transformation concepts described in section 2.1 on page 4. Knowledge about concepts and ideas like graph transformation rules, host graphs and resulting transition systems are implicitly assumed within this chapter.

For this bachelor project, there was no room to decide which graph transformation framework to choose in order to perform the graph transformations described by DMM rulesets, since the given transformation bases on a particular framework. But earlier, Groove as this particular graph transformation framework was chosen because of the affinity between Groove's graph transformation rules and the ones used by DMM. Furthermore, as will be shown, Groove visualizes the graph transformation rules and the graphs that are modified in a very convenient way. Besides, a great advantage is the fact that the transition system created by the application of the different rules is visualized as well. The complete decision-making process with the result to use Groove as the processing graph transformation framework can be found within [4].

In order to convey a basic knowledge of Groove, this chapter starts with a brief introduction to the graph transformation framework Groove in subsection 6.1. It depicts some of the most important characteristics of Groove and gives a brief introduction of the constructs that can be used within Groove. This subsection contains different figures that explain the notation of these constructs in short. Thereafter, section 6.2 delineates the differences and the similarities between DMM and Groove. The following two subsections, section 6.3 and section 6.4, give a brief overview over the current state of the transformation and its augmentation resulting from the extension of DMM rules by attributes.

6.1 Introduction to Groove

The graph transformation framework Groove is developed at the University of Twente, Netherlands, under the control of project manager Arend Rensink. The name “Groove” is an abbreviation of the phrase “GRaphs for Object-Oriented Verification”. Thus, this graph transformation framework aims to prove characteristics of graphs that are abstractions of structures found within object oriented programs at runtime.

Tools within the framework. On the whole this frameworks bundles different separate tools that are highly integrated. First of all, it contains an Editor for the creation of Groove graphs and Groove graph transformation rules–called Groove rules. The Groove graphs that can be created by this editor correspond to the host graphs to which the graph transformations specified by Groove rules are applied.

Secondly, it contains a tool called Generator that is able to derive transition systems founding on a given start graph–this is nothing else than a particular Groove graph–and a set of Groove rules. The definition of a transition system was given in section 2.1 on page 4.

These two tools are combined within a third tool, called Simulator. With the help of this tool, the user is able to create Groove rules and Groove graphs, and apply the Groove rules to these graphs directly. In the case of different matching rules it is left up to the user which rule to apply to the Groove graph. The choices made by the user form a transition system–not necessarily complete–that is visualized by the Simulator as well.

Another tool, called Model Checker, can verify characteristics of the transition system that is generated by the Generator. It enables the user to explore the state space and to use Computation Tree Logic *(CTL)* in order to check characteristics of the Groove rules and the different Groove graphs.

Finally, the graph transformation framework contains a tool called Imager in order to export Groove graphs and Groove rules and convert them to different image formats.

This bachelor thesis only focuses on the Simulator, and consequently on the Editor and Generator tool, as the other tools were not used in order to achieve the given goals.

Graph Transformation with Groove. Groove, as already mentioned, is a graph transformation framework that comes along with a notation for host graphs–called Groove graphs–and graph transformation rules–called Groove rules–of its own. Therefore, Groove has somehow created an own modeling language for the description of graph transformations.

Groove graphs are directed graphs, consisting of edges and nodes. Different nodes and edges together are called Groove structures. Only edges are allowed to have labels. Node labels can be achieved by self edges–these are edges with the same source and target node. Therefore, from now on, the term node label will be used as well, although Groove does not support the definition of labels for nodes directly.

Edges have no identity and are completely determined by their source node, their target node and their label. Therefore, it is not possible to distinguish between two edges that are similar according to these three properties.

Groove rules are graph transformation rules. Similarly to DMM rules, Groove rules combine the left-hand side and the right-hand side of normal graph transformation rules, resulting in a graph transformation rule that consists only of one single side. Groove enables the possibility to assign priorities to Groove rules. Thus, if more than one Groove rule matches the Groove graph, only the one–or respectively the ones–with the highest priority are applicable.

With the help of the label of an edge, two characteristics of an edge can be determined. One is called the aspect of the edge and the other one is called the name. Aspects can only be used within Groove rules. An aspect of an edge can be seen as an additional piece of information that describes, e.g, what happens with the matched edge within the Groove graph on application of the particular rule. The other characteristic is the name of the edge. This limits the matching of a Groove rule, since the names of the edges defined in the Groove rule and in the Groove graph must fit to each other. This means they have to be absolutely equal such that they can be mapped to each other.

6.1.1 Some Groove Constructs

This subsection presents some of the constructs that are commonly used and are necessary to describe the transformation from DMM rules to Groove rules and its augmentation by the introduction of attributes, conditions and assignments. A profound introduction into the concepts and constructs that are used by Groove can be found in [24].

During this subsection, the attentive reader might already recognize some possibilities to transform constructs used within DMM rules to corresponding constructs that are used within Groove rules. Some hints are given how and what for the described Groove constructs can be used.

Visualization of Groove rules. As already mentioned, Groove graphs and rules are edge-labeled, where each edge consists of two parts: An aspect and a name.

The aspect can be regarded as an instruction that specifies the meaning of the edge. Therefore, an

edge can have, for example, a special aspect that indicates that this edge has to be created if the particular graph transformation rule is applied.

Aspect values are visualized using different colors within the simulator. They do not have a particular string representation. Instead, they determine the visual appearance of the according edges and nodes.

The name of an edge is shown as a string near the edge. If a node has a self edge, the edge's name is directly shown within the given node. Of course, it is possible to define more than one self edge for a node, giving a node different names.

Basic concepts. Figure 24 contains all the different aspects that can be used in order to define whether a particular structure must be present or absent within the Groove graph and what is going to happen with such a structure, if the particular Groove rule is applied to the Groove graph.

Everything that has to be present such that the Groove rule matches the Groove graph is painted black. Structures that have to be present, but are removed on application of the Groove rule are painted blue. Structures that are created on application of the rule, are painted green. Finally, negative application conditions are illustrated by red edges and nodes.

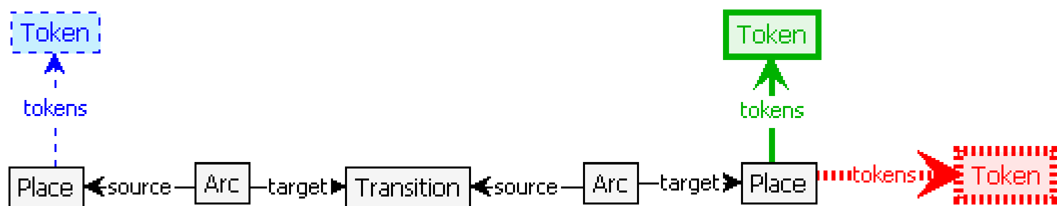

Figure 24: Groove rule demonstrating the usage of aspects

The Groove rule shown in Figure 24 describes that, if applied, a node with the name `Token` is deleted from a node with the name `Place` and recreated on another node with the same name. These two places have to be connected by the given structure that contains two nodes with the name `Arc` and one node with the name `Transition`. The negative application condition states that this rule is only applicable if the place on the right side does not have any tokens. The rule can be interpreted as follows: A token is handed over from an input place through a transition to an output place that itself does not have any tokens.

As the reader might have noticed, the names used for the nodes correspond to the types defined within the runtime metamodel of the running example. Furthermore, the names of the edges that connect the different nodes correspond to the references used within the runtime metamodel. Additionally, the given aspects resemble the different roles of nodes and edges that can be defined within DMM rules.

Quantifications. Besides these aspects, there are two aspects that are necessary in order to make statements about more than exactly one Groove structure of the Groove graph. These two aspects are assigned to Groove nodes that are then called Groove quantification nodes. These nodes themselves do not match any nodes within the Groove graph.

The following descriptions are visualized in Figure 25. Nodes having one of these two aspects are visualized by either the sign $\forall$ or the sign $\exists$. The $\exists$ node must be nested within the $\forall$ node. This is achieved with the help of an edge, originating in the $\exists$ node and ending in the $\forall$ node, with the name `in`. Normal groove nodes can be connected to these quantification nodes via an edge with the name `at`. Such a connection modifies the meaning of the connected nodes.

These quantification nodes can be compared to quantifiers known from predicate logic. Similarly to its meaning within predicate logic, the $\forall$ node selects all the subgraphs within the Groove graph that are matched by the structure connected to such a node via edges with the label `at`.

This can be seen in the given example Groove rule visualized in Figure 25. By the connection between the node with the name `Arc` and the ∀ node, the `Arc`-node is matched to all the nodes with the name `Arc` within the Groove graph that have an outgoing edge with the name `target` to the given transition. Thus, the ∀ node can be seen as a selector of Groove nodes or structures.

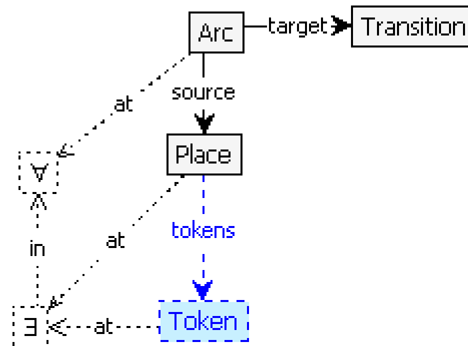

Figure 25: Groove rule that demonstrates the usage of quantification nodes

The nested ∃ node, again having a pretty similar meaning to the quantifier used within predicate logic, states which structure has to be present or absent for each structure matched by the nodes connected to the ∀ node such that the rule can be applied. This structure must be connected to the ∃ node via edges with the name `at`.

In the case of the example Groove rule given in Figure 25, the Groove structure connected to the ∃ node represents a place with a token. It states that for every node with the name `Arc` that is connected to the transition via an edge with the name `target`, there must be the nodes with the name `Place` and `Token`, connected as shown in Figure 25. Another aspect specifies that the node with the name `Token` is deleted on the application of the rule.

On the whole, the given example can be seen as a directive that deletes exactly one token of each input place, but only if every input place has at least one token.

These quantification nodes remind strongly of quantification cluster, nested cluster and uqs cluster that can be used within DMM–see subsection 2.3.2 on page 10 for a detailed description. Accordingly, this kind of Groove aspects and nodes will be used for the transformation of DMM quantification cluster.

Attributes. Finally, a third important Groove concept shall be mentioned in this place. This concept deals with attributes, their usage and calculations that can be defined to use these attributes' values. An attribute cannot be directly expressed by Groove. Only nodes can be marked within Groove rules and Groove graphs to represent either a particular value of a particular data type or–within Groove rules–to match an arbitrary value. Thus, it is possible to express an attribute via an edge, containing the name of the attribute, that is connected to such a value node.

By this, an object within the host graph, having different attributes and values for these attributes is represented within the Groove graph by an according Groove node having an edge for each attribute, with the attribute's name as its name, pointing to a node that defines the attribute's value.

Figure 26 visualizes a Groove rule that, if applied, deletes the edge with the name `token` to a node that maintains the number of tokens situated at the given place. Furthermore, it creates a new edge, likewise called `token`, to the result of a calculation. Both target nodes of these edges are visualized by circles, denoting that they refer to a value of some data type. The data type itself is not visualized in any way.

The diamond shape in the middle of Figure 26 represents a calculations. Such calculations are called production within Groove. In this case, it is an addition of the values referenced by the

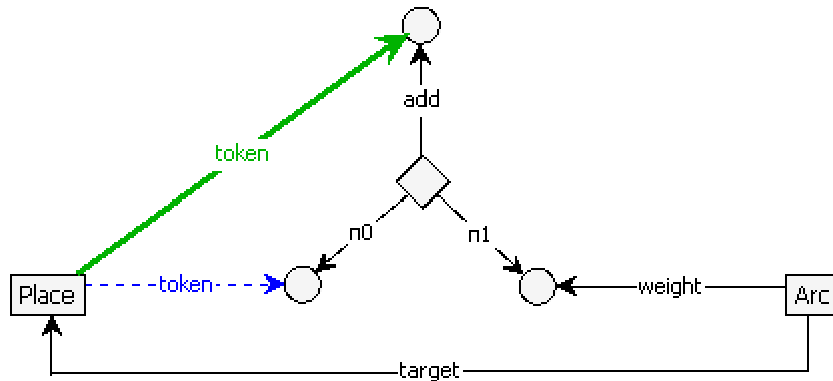

Figure 26: Groove rule containing a possibility to express attributes

edges with the name `n0` and `n1`. The kind of production is determined by the name of the outgoing edge that does not reference one of the operands–this is the edge with the name `add` in the given example. The result of this production is stored in the circle that is connected to the diamond shape via this edge. This node is called the production result node, or simply production result. Finally, as already mentioned, an edge with the label `token` is created to the production result, thus updating the number of tokens that belong to the place under consideration.

The whole Groove rule resembles strongly the example DMM rule given in Figure 6 on page 15. This is a first hint on how assignments can be realized within Groove.

6.2 Comparison between Groove and DMM

In the following paragraphs, let us focus on some similarities and some differences between Groove and DMM that have to be kept in mind for the transformation.

Groove, as already mentioned, combines a modeling language for the description of graph transformations and tooling support that enables the processing of these graph transformations.

DMM, on the other hand, is only a modeling language for the specification of dynamic semantics of behavioral modeling languages. Basically, the specification of DMM only explains the meaning of the different DMM constructs but does not provide any possibility to actually perform the defined graph transformations.

But, in order to really use DMM, it is necessary to perform these graph transformations. Therefore, Groove as a graph transformation framework is used. By the decision to take Groove, the framework's functionalities are automatically added indirectly to DMM. There must only be a possibility to transform DMM rules to Groove rules.

Besides, there is one great difference between DMM and Groove. As already mentioned, DMM types its host graphs and its rules over the runtime metamodel. Such a metamodel is not available within Groove. Because of this, Groove does not directly support the definition of instances of runtime metamodels. But, by a smart transformation of DMM rules and host graphs that are typed over the runtime metamodel to according Groove rules and Groove graphs, it is still possible to use the concept of the runtime metamodel within DMM. How this is achieved is described in the next section.

Furthermore, the concepts of conditions and assignments that were added to DMM are not directly supported by Groove and have to be transformed to Groove constructs that have the same meaning as conditions and assignments.

The terms that are used by DMM or Groove differ slightly and therefore are mapped in Table 9.

DMM	Groove
DMM ruleset	Graph grammar
DMM rule	Groove rule
host graph	Groove graph
Properties like roles and quantifications	Aspects of nodes and edges

Table 9: Relationship between DMM and Groove

6.3 Current State of the Transformation

This section delineates the current state of the transformation DMM rulesets to Groove grammars. This transformation transform DMM rules that contain attributes, conditions and assignments. At this place, the whole current state of the transformation is described as it builds a foundation for the augmentation of the transformation. Furthermore, it was partly modified such that the outcome of these modifications should be describe here, as well. But it should be mentioned that this transformation was not completely created in the course of actions of this bachelor project. Only minor modifications and the whole augmentation in order to support attributes within DMM was created during this bachelor project.

Additional data structures and algorithms used to perform the transformation are omitted at this place. These are, for example, data structures for the access of specific model objects quickly and algorithms for the traversal of the instance of the DMM metamodel[22].

At this place, it will only roughly be explained how the different DMM constructs and concepts are mapped to the according Groove constructs and concepts. Therefore, DMM rule excerpts and the according Groove rule excerpt are opposed to each other and a mapping of the different constructs is given by example.

For these mappings special markers are used in order to vividly express the relationship between DMM constructs and Groove constructs. Markers are numbers, like 𝟙, 𝟚 and so on, that are presented in a special outlined way. If, for example, the marker 𝟙 is assigned to one construct within the DMM rule and another construct within the according Groove rule, this marker states that these constructs correspond to each other, thus showing which construct is transformed into which other one. In the case of a missing according construct, for example because of extensions that had to be introduced on one side only, the marker is omitted on the other side.

Every transformation of a particular DMM construct ends with a transformation directive that sums up the transformation and specifies precisely which construct of a DMM ruleset is transformed to which construct or structure of the according Groove grammar. These two concepts–the marker concept and the transformation directives–are used within the this section and the following section 6.4.

All the example DMM rules that are used within this section in order to describe the transformation are typed over the unextended runtime metamodel shown in Figure 2 on page 7.

This section starts by describing some actions that have to be performed before the actual transformation. This is done in subsection 6.3.1. Thereafter, subsection 6.3.2 illuminates the transformation of DMM nodes, edges and roles. Subsection 6.3.3 continues with the description of the transformation of quantified nodes. Finally, subsection 6.3.4 depicts the transformation of invocations to according Groove structures.

22 Nevertheless, these data structures and algorithms are described completely within the Javadoc that was created along this bachelor project.

6.3.1 Preliminary Actions

Before the actual transformation of DMM rules to according Groove rules can be performed, some efforts have to be made in order to get rid of Premise rules and invocations of Premise rules–see section 2.3.2 on page 11 for a description of the different kinds of rules–since they are not supported by Groove in any way.

Premise rules are used in order to partition vast Bigstep or Premise rules and to reuse rule parts within other rules. They are only allowed to affect the matching of the Bigstep rule that invokes the Premise rules. Recall, that Premise rules are not allowed to use DMM constructs that would modify the host graph in any way.

Thus, in order to get rid of Premise rules, they can be merged into the invoking rules recursively. This means, if either a Bigstep rule or a Premise rule invokes another Premise rule, the Premise rule itself is merged into the invoking Premise or Bigstep rule. In the case of an invocation of a Premise rule that is performed by the invoked Premise rule, this process is repeated recursively.

Different aspects have to be considered in the process of merging. For this, think of an invocation of a method defined in an object oriented programming language. The invoking method binds the actual parameters and the object on which the invocation is performed to the formal parameters and executes the method in the context of the given object. Similarly, an invoked rule is bound to the invoking rule. Therefore the target node of the invocation is bound to the context node of the rule, and the actual parameter are mapped to the formal parameters.

By this knowledge, the merging of one Premise rule into one invoking rule can be performed. To get a binding between the invoking rule and the invoked Premise rule, the context node and the formal parameters are mapped to the target node and the actual parameters. All the edge adjacent to the context node in the Premise rule are copied to the target node situated within the invoking rule. Likewise, the edges adjacent to the formal parameter are added to the actual parameters. After this binding was created, all the other nodes and edges that were not copied yet are copied to the invoking rule. After all these actions, the invocation of the Premise rule within the invoking rule is deleted.

Different constraints, regarding the role and the quantification of the target node and the context node, or respectively the actual parameters and the formal parameters, have to be obeyed. These constraints are automatically ensured by the given graphical editor.

As a result of these actions, all the invocations of Premise rules are left out, leading to the possibility to delete all the Premise rules within a DMM ruleset.

Besides these actions, an additional properties file is added to each Groove grammar. This file contains some properties that affect the matching algorithms used by Groove. The main property defined within this file specifies that the resulting Groove rules should match the Groove graph injectively. This means in general that two different Groove nodes used within the Groove rule cannot match the same node of the Groove graph.

> The Premise rules have to be merged recursively into the invoking rules as follows: All the edges adjacent to the context node are copied to the target node. Likewise, the edges adjacent to the formal parameter are added to the actual parameters. Thereafter all the other elements–nodes and edges–are merged with the invoking rule. Finally the invocation of the Premise rule is deleted. After all the Premise rules are merged into their invoking rules, they are removed from the DMM ruleset.
>
> Additionally, a special properties file, ending with the extension `.properties` is added to each Groove grammar. There the property `matchInjective` is set to the `boolean` value `true`.

Transformation directive 1: Preliminary actions before the actual transformation

6.3.2 Transformation of basic constructs

On the whole, the mapping between DMM rulesets and Groove grammars is quite straight forward.

Every DMM rule leads to the creation of a corresponding Groove rule. The priority of this newly created Groove rule is set to the value 1. This is important since other helper Groove rules will be created in the following section that have another priority than 1.

The transformation of DMM nodes and edges to according Groove nodes and edges shall be visually explained with the help of the DMM rule visualized in Figure 27 and the according Groove rule shown in Figure 28. The latter rule was already shown in Figure 24 on page 69.

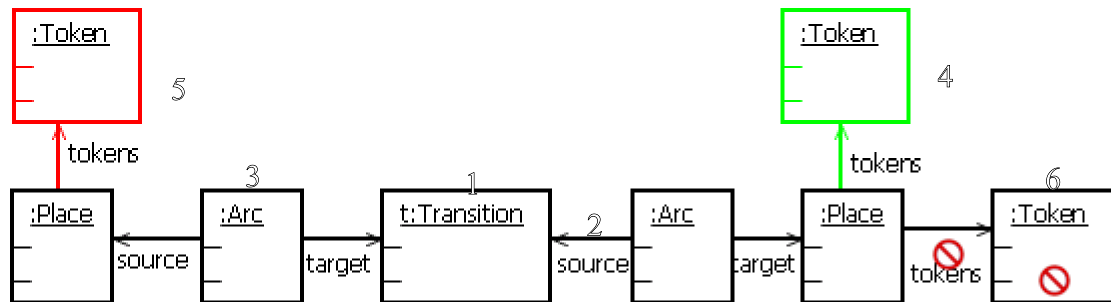

Figure 27: A DMM rule containing nodes and edges with different roles

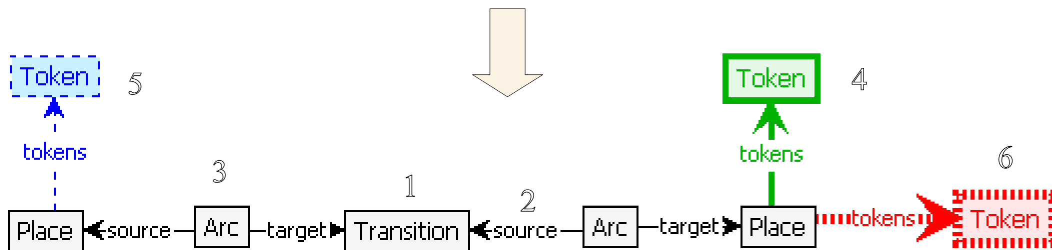

Figure 28: The resulting Groove rule, demonstrating the transformation of DMM nodes and edges

Every DMM node is mapped to an according Groove node, and every DMM edge is mapped to an according Groove edge. Therefore, the resulting Groove rule looks pretty similar to the given DMM rule. The type of a DMM node is used as the name of the according Groove node. Names that are used within DMM for the addressing of nodes are not transformed to according Groove names. An example for this is shown at marker 1. The name of the edge's reference defined within DMM rules is used as the edge name within the corresponding Groove rule. Marker 2 shows exemplary the described fact. Recall that the references name is defined in the runtime metamodel and not by the user within the graphical editor.

The roles are mapped accordingly. The `exists` role within DMM is mapped to the Groove aspect that is visualized by the color black–see marker 3 for an example. The nodes and the edges with the role `create` are accordingly mapped to Groove nodes and edges with the Groove aspect that states the creation of nodes. This is done by the color green shown at marker 4. Similarly, the `destroy` role results in the Groove aspect that is visualized by the blue color shown at marker 5. And finally, nodes and edges with the role `not exists` are transformed to Groove nodes and edges that are visualized by the aspect that is painted red. This is shown at marker 6[23].

One additional structure has to be created for each DMM node with the role `create`. Beside the name of the type of such a DMM node, the names of all the supertypes of this type are added to

23 Note that the colors used within Groove rules differ a slightly from the colors used within DMM rules. Differences are the color for the deletion of nodes and edges and the color used to indicate negative application conditions.

the corresponding Groove node. This enables the possibility of matching this Groove node when a DMM specifies the matching only of a supertype of the corresponding object within the host graph. This concept was described in subsection 2.3.2 on page 9. A much more detailed explanation of this concept can be found in [25].

Apparently, as both rules can be directly mapped to each other, the meaning of the rules seem to be equal and were not changed by the transformation.

Create a Groove node or respectively a Groove edge for each DMM node or respectively DMM edge. Use the type of the DMM node for the name of the Groove node, and the reference name of the DMM edge for the name of the corresponding Groove edge. Set the aspect value according to the element's role.

Finally for each DMM node with the role `create`, add the names of all the supertypes of this node as self edges to the corresponding Groove node.

Transformation directive 2: Transformation of the basic constructs

6.3.3 Transformation of quantified nodes

This subsection addresses the special treatments that are made for quantified nodes. Similar to DMM where the meaning of the node is changed by the value of its `quantification` property, the meaning of the corresponding Groove nodes has to be changed as well.

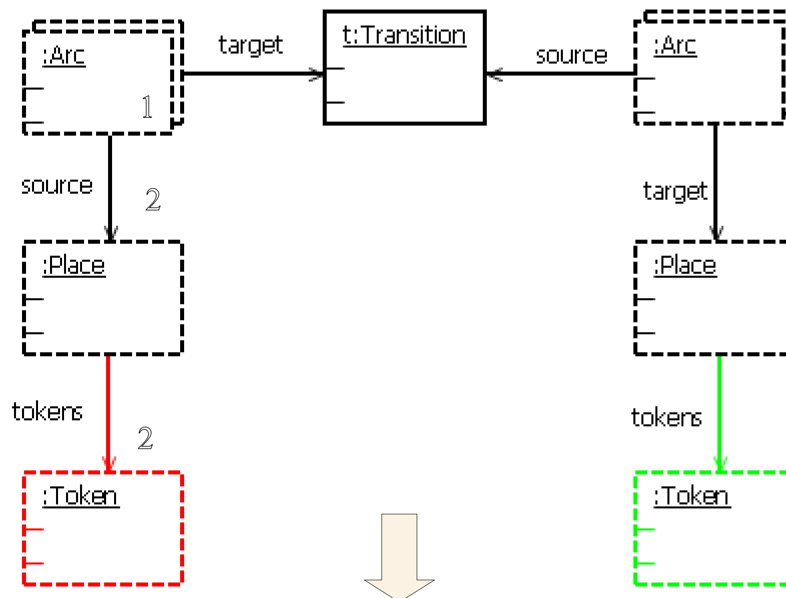

Figure 29: Firing of a transition without capacities and weights

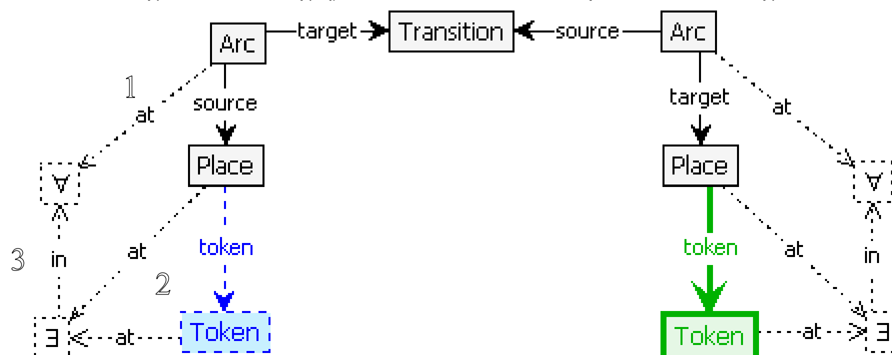

Figure 30: The according Groove rule using Groove quantification nodes

For a vivid explanation, the DMM rule given in Figure 29 is transformed to the Groove rule giv-

en in Figure 30. The DMM rule shown in Figure 29 is exactly the same like the one shown in Figure 3 on page 8. The DMM rule and the according Groove rule describe formally the possibility of a transition to fire and the changes that are made to the marking of the petri net if that transition fires.

As already mentioned in subsection 2.3.2 on page 10, each quantification cluster within DMM consists of a uqs cluster and a nested cluster. During the transformation of such a quantification cluster, two special Groove nodes, the ∀ node and the ∃ node are created. This pair of Groove nodes is created for each quantification cluster. All the universally quantified nodes of the quantification cluster are then connected via an edge with the name `at` to the ∀ node–an example for this can be seen at marker 1. There the node with the name `Arc`–representing the nested node with the type `Arc`–is connected to the ∀ node via an edge with the name `at`.

Similarly, all the nested nodes of the quantification cluster are connected to the ∃ node via an edge with the same name `at`. An example of this procedure for the given rules is visualized at marker 2. The nodes with the name `Place` and `Token` are connected to the ∃ node via an edge with the name `at`.

Finally, the ∃ node is nested within the ∀ node for each quantification cluster. This is done by the creation of a Groove edge from the ∃ node to the ∀ node with the name `in`. Marker 3 shows that exactly this was done for the quantification cluster on the left side of the figure.

By this transformation, the meaning of the quantified nodes within DMM is directly mapped to according Groove structures. Similarly to the meaning within DMM, where a uqs cluster states which subgraphs of the host graph to match and the nested cluster defines which elements must be present or absent for such a subgraph, the resulting Groove structure claims the same characteristics for the according Groove graph.

Create a ∀ node for each uqs cluster, and a nested ∃ node for each nested cluster, connected via an edge with the name `in` if the nested cluster belongs to the uqs cluster. Connect each universally quantified node to the corresponding ∀ node of the uqs cluster, and each nested node to the corresponding ∃ node of the nested cluster via an edge with the name `at`.

Transformation directive 3: Transformation of quantified nodes

6.3.4 Transformation of Invocations

This subsection illuminates the transformation of the invocation concept that can be used within DMM. This part of the transformation needs some more work, since Groove itself does not support invocations in any way. Therefore, additional Groove structures have to be created and maintained in order to achieve the desired functionality within the given graph transformation framework.

To gain a deep understanding on how and why the following Groove structures are created and how they are maintained, it is necessary to completely understand the invocation concept of DMM described in subsection 2.3.2 on page 11.

This subsection starts with the description of an additional Groove structure that is inserted into each Groove start graph in subsection 6.3.4.1. Thereafter, subsection 6.3.4.2 depicts shortly changes to invoking and invoked rules, but only regards invocations on non-quantified nodes. Finally, subsection 6.3.4.3 depicts the adjustments of the invoking rule that are made for invocations on quantified nodes.

6.3.4.1 DMM Invocation Stack

As already mentioned in 2.3.2 on page 11, the invocation concept in the case of Smallstep rules states that after the changes defined by the invoking rule are carried out to the host graph, all the invocations specified within the invoking rule are performed. This means that the invoked Smallstep rule is matched to the host graph and the host graph is changed as specified by the invoked rule.

This concept resembles strongly to object oriented programming where method calls are executed on particular target objects, with given actual parameters. Within these programming languages a special data structure, named call stack, is used to handle the invocations of methods.

The call stack maintains the invocations of methods and the transferring of the target object and actual parameters from the invoking method to the invoked method. For each invocation of a method, the target object and all the actual parameters are stored within this data structure. Thereafter, the method that has to be executed is searched and executed. The call stack ensures that the invoked method is executed on the right target object and retrieves the right actual parameters.

Similarly, DMM uses an additional data structure within Groove graphs in order to enable the transformation of invocations. This data structure is called the DMM Invocation Stack *(DIS)*. The DIS handles the invocations of Smallstep rules. But, in contrast to the call stack used by object oriented programming languages, it stores, besides a reference to the target object and the actual parameters, the name of the invoked Smallstep rule.

For instance, Figure 31 shows an exemplary DIS containing the invocation of one Smallstep rule called `deleteToken`. The target node of this invocation is referenced by an edge with the name `self`. In case of actual parameters, all the according objects–or respectively Groove nodes–would be referenced by edges with the name `parameter`. Every element within the stack has the additional name `Invocation`. The top element is always called `DMMSystem`. It indicates the top of the stack. The node with the name `_bottom` marks the bottom of the stack.

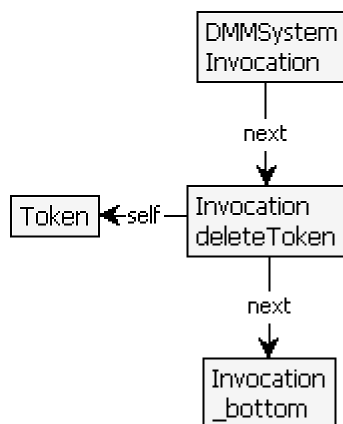

Figure 31: Exemplary DIS with the invoked rule deleteToken

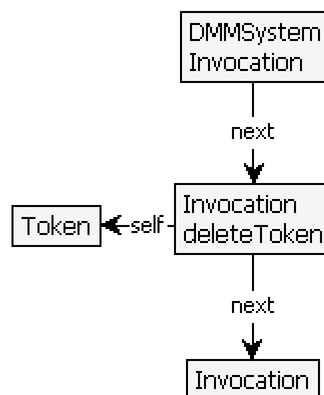

Figure 32: Necessary Groove structure to match invoked Groove rule

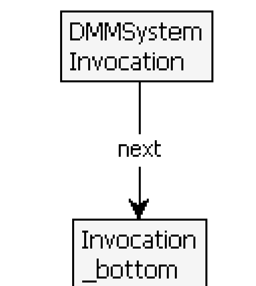

Figure 33: Empty DIS

With the help of this stack–and some adjustments of the invoking and invoked rule that will be given in the following sections–it is guaranteed that the invoked Smallstep rule or respectively the corresponding Groove rule is the only rule that can match the Groove graph.

Let us look at the structure that must be present such that the Groove rule corresponding to the invoked Smallstep rule with the name `deleteToken` can match the Groove graph. This structure is given in Figure 32. It can be seen in comparison to the structure given in Figure 31 that the exemplary DIS has nearly the similar structure that must be present such that the invoked

rule can be applied. The only difference is that the invoked rule does not state that its invocation must be the only invocation on the DIS and therefore be situated at the bottom.

In order to ensure that no Groove rule that corresponds to a Bigstep rule matches the Groove graph, although a Smallstep rule was invoked, an additional Groove structure is added to each Groove rule, created for a Bigstep rule. This Groove structure is given in Figure 33. It simply claims for an empty DMM Invocation Stack. Thus, another Groove rule–corresponding to a Bigstep rule–can as recently match the Groove graph as all the invocations are performed and the Groove rules corresponding to the invoked Smallstep rules are applied.

In this case, it can be obtained by the deletion of the middle part of the Groove graph given in Figure 31. But, exactly this–as we will see in the next subsection–is done if the Groove rule that corresponds to the invoked exemplary Smallstep rule with the name `deleteToken` is applied to the Groove graph.

Though, one problem still remains: What happens if the invoked Smallstep rule does not match the host graph? This normally indicates an error that was made in the specification of either the host graph or the DMM ruleset. This case can be identified by one generic additional Groove rule that is added automatically to each Groove grammar. It matches the Groove graph, if no Groove rule corresponding to a Bigstep rule and no Groove rule corresponding to an invoked Smallstep rule can match the graph. In order to guarantee that this special Groove rule does not match the Groove graph in any other situation, the smallest priority of all rules is assigned to this rule. This is the priority `0`.

> Add a Groove structure that represents an empty DMM Invocation Stack to each Groove rule that corresponds to a Bigstep rule. This Groove structure consists of two nodes that are connected by an edge with the name `next`. The source node has the names `DMMSystem` and `Invocation`, and the target node has the names `Invocation` and `_bottom`.
>
> Furthermore, augment the Groove grammar by one generic additional Groove rule with the priority `0` that matches the Groove graph if the Groove rule that corresponds to an invoked Smallstep rule does not match the Groove graph.

Transformation directive 4: Creation of the DMM Invocation Stack

6.3.4.2 Invocations performed on non-quantified nodes

Two additional adjustments of the transformation described so far have to be made in order to enable the invocation on non-quantified nodes. One of the adjustments affects the transformation of the invoking rule and the other one accordingly the transformation of the invoked rule.

Similarly to the compilation of object oriented programming languages, where the invoking method determines where to jump for the execution of the invoked method and the latter one states where to jump back, the same behavior has to be established by the transformation of the DMM ruleset.

The invoking rule. The adjustments regarding the invoking rule affect the modification of the DMM Invocation Stack at runtime–this means when the actual graph transformation rules are applied to the Groove graph. Invoking rules can be either Bigstep rules or Smallstep rules. Although there is a slight difference regarding the adjustment of the transformation, this difference is left out due to simplicity[24].

Figure 34 and Figure 35 are used to explain the adjustments vividly. The Bigstep rule given in Figure 34 makes use of an invocation of a Smallstep rule called `deleteToken` invoked on the target node with the label `t:Token`. The Bigstep rule itself does not modify the host graph.

24 Nevertheless it is documented in the corresponding Javadoc.

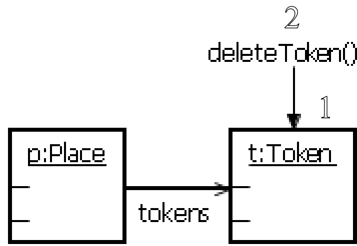

Figure 34: Demonstration of invocations

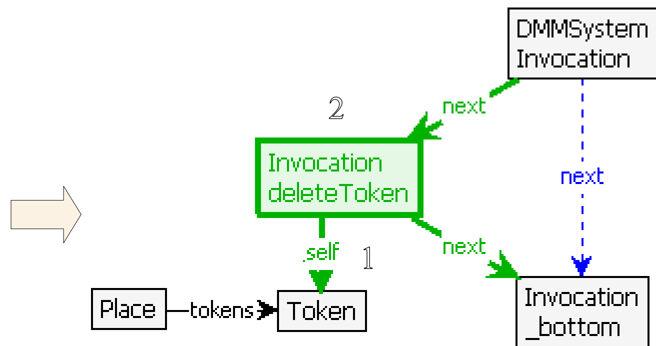

Figure 35: According Groove structure, representing an invocation

Marker 1 demonstrates how the target node is transformed to an according Groove construct. The edge with the name `self` points to the Groove node that corresponds to the target node.

The invocation itself is represented by a Groove node having the name `Invocation` and the name that determines the invoked rule. This Groove node is put on top of the stack[25]. This is described by the Groove structure given at marker 2. The resulting Groove structure after the application of the Groove rule that corresponds to the Bigstep rule visualized in Figure 34 regarding the DMM Invocation Stack is exactly the one given in Figure 31. Thus, the DIS is correctly modified by the Groove rule that corresponds to the invoking DMM rule.

If more than one invocation is used within a DMM rule, the resulting Groove structure pushes a list of Groove nodes that represent invocations on the DIS, according to the sequence numbers assigned to the different invocations.

The invoked rule. Besides these adjustments to the invoking rule, the invoked rule itself has to be adjusted in some ways. Similarly to the decision where to jump back after the execution of the invoked method within object oriented programming languages, the Groove rule that corresponds to the invoked rule has to change the DIS in a particular way, after performing its changes to the Groove graph. The adjustments have to guarantee that the DMM Invocation Stack is modified in such a way that the Groove node representing the invocation is popped from the stack. How this is done can be seen in Figure 36 or respectively Figure 37.

The invoked DMM rule states that the context node has to be deleted. Similarly the according Groove node has the aspect represented by the color blue–this can be seen at marker 1. The rest of the Groove rule, shown at marker 2, manages the DMM Invocation Stack. It deletes the Groove node that represents the invocation and removes it from the DIS.

Figure 36: Invoked Smallstep rule

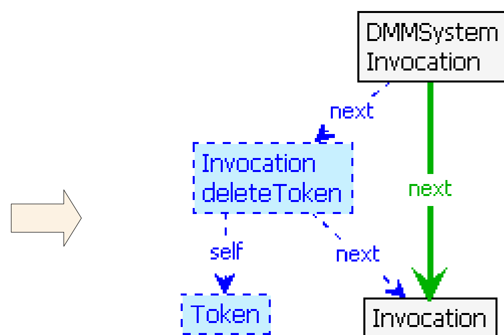

Figure 37: According Groove rule

If the aspects concerning the DIS are left out, the Groove rule visualized in Figure 37 requires

25 As this given DMM rule is a Bigstep rule, and because Bigstep rules only match the host graph if no Smallstep rule is invoked, the DMM Invocation Stack is accordingly empty.

exactly the Groove structure given in Figure 32 on page 77 to match the Groove graph. By this, the DIS and the transformation of the invoked DMM rules together ensure that the invoked DMM rule, or respectively the according Groove rule, is the only rule that can match the Groove graph, thus guaranteeing that only this–invoked–Groove rule can by applied.

By the deletion of the Groove node that represents the invocation the DIS is modified in such a way that it results in the same DIS like the one before the invocation. By this, the invoking rule some how cleans up the DMM Invocation Stack when it is applied to the Groove graph. In this case, the resulting DIS is shown in Figure 33 on page 77.

The Groove structure concerning the DIS given in Figure 37 has to be added to each Smallstep rule. Bigstep rules are not affected by this adjustment of the transformation, since they cannot be invoked, but can only match the host graph directly.

An additional Groove structure is added to the corresponding Groove rule for each DMM rule that contains invocations. For this, a list of Groove nodes with the name `Invocation` and the name that corresponds to the invoked Smallstep rule regarding the defined ordering is created. These nodes are connected by edges with the name `next`. This list is inserted into the DMM Invocation Stack by the correct insertion of edges with the name `next`. The target nodes and the actual parameters are bound by edges with the name `self` or respectively `parameter`.

Every Groove rule that corresponds to a Smallstep rule is adjusted by the addition of a Groove structure for the correct modification of the DMM Invocation Stack. This structure mainly consists of a Groove node with the name `Invocation` and the name of the Smallstep rule that has the aspect which deletes the this node from the DIS on application. Furthermore, the according Groove node–from the Groove graph–is removed from the DIS. This is expressed by the correct creation and deletion of edges with the name `next`.

Transformation directive 5: Transformation of invoking and invoked rules

6.3.4.3 Invocations performed on quantified nodes

There is one final adjustment that has to be made in the course of the transformation of invocations on quantified nodes that affects only the invoking rule. Invoked rules, since they are not aware whether they are invoked on quantified nodes or not, do not have to be modified in any way.

But what is the difference between invocation performed on non-quantified nodes and the ones performed on quantified nodes? In order to answer this question, recall that a quantified node matches arbitrarily many objects within the host graph. Consequently, an invocation on a quantified node, either with the quantification property set `0..*`, `1..*` or `nested`, can be regarded as many different invocations–or to be more precise a list of invocations–one invocation for each object of the host graph that is matched by the target node. Thus, the according DMM Invocation Stack has to be modified several times, one time for each such object. This fact again needs some efforts–some additional Groove structures and even some additional Groove rules–such that the invocations are performed as specified by DMM.

Let us look at an the example Bigstep rule from Figure 34, but let us now invoke the Smallstep rule `deleteToken` on a uqs node. This DMM rule and the corresponding Groove rule are shown in Figure 38 and Figure 39.

Because the node with the label `t:Token` is a uqs node, the corresponding Groove node is connected to a ∀ node. This can be seen at marker 1. Since the invocation is not performed on a non-quantified target node, a Groove node having the name `Invocation` and the name that corresponds to the invoked rule is created that is connected to the ∃ node which was created for the

quantification cluster. This ∃ node is connected to the ∀ node via an edge with the name `in`. By this, a Groove node having the stated names, is created within the Groove graph for each Groove node with the name `Token` that is connected to the place via an edge with the name `tokens`. All these nodes are connected to a node with the names `Invocation` and `UQSInvocation` via an edge with the name `DMM_handleInvocation`–shown by marker 2. This results in a star shaped structure in the Groove graph, where the `UQSInvocation` node builds the center point of this star.

Figure 38: Invocation on quantified node

Figure 39: According Groove rule, with invocation on quantified node

But only the node with the name `UQSInvocation` is inserted into the DMM Invocation Stack like shown at marker 3. Thus, there are still some graph transformations left that have to be performed in order to put all the Groove nodes that represent the invocation into the DMM Invocation Stack and delete the Groove node with the names `UQSInvocation` and invocation.

The unfolding of the star shaped structure is achieved by three different auxiliary Groove rules. Whenever the `UQSInvocation` node is present within the Groove graph, only one of these three auxiliary Groove rules can match the graph. This is achieved by giving these three rules a priority greater than `1`.

Every invocation on a quantified node is transformed to a Groove structure as follows:

A new node with the names `UQSInvocation` and `Invocation` is pushed on the DMM Invocation Stack. Furthermore, a Groove node that represents the invocation of the Smallstep rule, is created and bound to the ∃ node of the quantification cluster. Thereafter, an edge is created with the name `DMM_handleInvocation` from the `UQSInvocation` node to the quantified Groove node that represents the invocation. Additionally, the target node and all the parameters are referenced by edges with the name `self` and `parameter`.

Finally, three auxiliary Groove rules are added to the Groove grammar that unfold the star shaped structure created for each invocation on a quantified node.

Transformation directive 6: Transformation of invocations with a quantified target node

6.4 Implementation of the Augmentation

On the whole, many different adjustments and extensions were made to the transformation of DMM rulesets to Groove grammars for the introduction of attributes, conditions and assignments. The following subsections depict these extensions.

Similarly to the description given in the preceding section, data structures and algorithms used for the transformation are omitted at this place. Especially the expressions, conditions and assignments had to be transformed in a specific order in order to be able to reuse results of the transformation of other constructs. A precise description can be found within the according Javadoc. All additional data structures and algorithms are described there in detail.

The first subsection of this section starts with the description of the transformation of expres-

sions to according Groove constructs and structures. Subsections 6.4.2 and 6.4.3 state the transformation of conditions and assignments based on the transformation of expressions. Finally, subsection 6.4.4 depicts some additional treatments and changes that had to be performed to the transformation concerning certain roles and concepts of DMM and their interdependences with attributes.

6.4.1 Transformation of Expressions

In order to explain the transformation of expressions comprehensibly and reuse already familiar examples, a pretty similar example DMM rule like the one given in Figure 5 on page 15 shall be used at this place. Only a minor change–the replacement of the `capacity` identifier by the literal `100`–has been made in order to show all the cases of the transformation of expressions. The resulting DMM rule can be seen in Figure 41. The according Groove rule is shown in Figure 42. These two figures are used within this and the following subsection.

Let us first of all start to explain how attributes used within DMM host graphs are represented within Groove graphs. Every object of the host graph knows the current values of each attribute defined in the type and any of the supertypes of the object. For example, every instance of the type `Place` from the extended runtime metamodel–shown in Figure 4 on page 13–of the running example contains values for all of the different attributes defined in the type `Place`: This are the attributes with the names `token` and `capacity`.

Figure 40: Attributes within the Groove graphs

The attributes and their values are represented within Groove graphs by an edge having the name of the attribute, and a target node carrying the value of the attribute, as shown in Figure 40. This Groove graph contains a Groove node that represents an object of the host graph of the type `Place`. The attributes `token` and `capacity` have the values `3` and `10`. The type of the attribute's value is given by an appropriate Groove aspect. It is not visualized in any way in the given figure.

Similarly, Groove rules contain the same structure in order to access the values of attributes. For example, the current value of the `token` attribute, referenced by an instance of the type `AttributeExpression`, can be accessed via the structure indicated with the marker 1 in Figure 42. The current value is thus stored in the circular figure at the end of the edge with the name `token`. Such a structure is created within the Groove rule for every old value of an attribute that is referenced by an identifier within any expression of the DMM rule. Similarly, the value of the attribute `weight` of the object having the type `Arc` is accessed.

If literals are used within expressions–they are represented within DMM by an instance of the type `LiteralExpression`–, an additional Groove node is created that contains the value of the literal. These values are represented within Groove by rounded nodes with the value as the name of the node. The data type of the literal is determined by an according aspect, but not visualized within the given figure. This can be seen in Figure 42 at marker 2.

Every operation–represented by an instance of the type `OperationExpression`–is transformed to a production within Groove. The expression of the condition given in Figure 41 is comprised of two operations and is therefore transformed to two productions, represented by the two diamond shapes on the right side of the Figure 42. The result of the first operation–the addition–is used within the second operation–the comparison. This can be seen within Figure 42, where the result of the first production is used as input operand for the second production. The marker 3

shows the direct mapping between the addition and the corresponding Groove production.

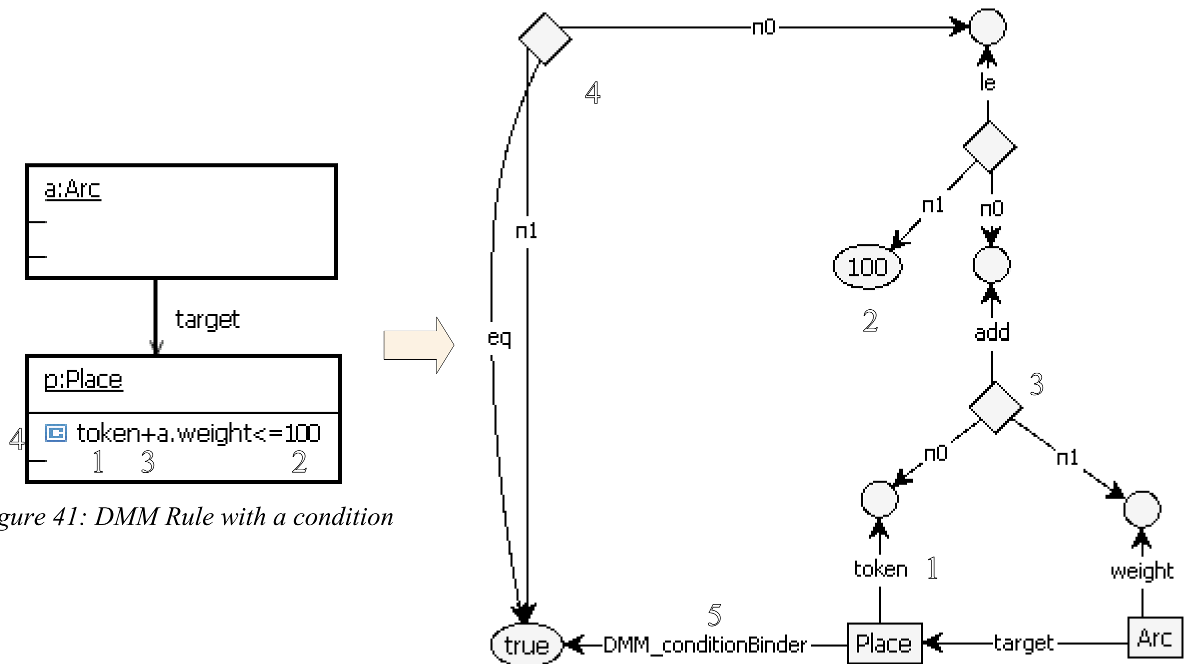

Figure 41: DMM Rule with a condition

Figure 42: According Groove rule, representing a condition

As Groove supports all the operations that were defined in Table 6 on page 24, all the DMM operations can be directly transformed to Groove productions. But, since Groove uses by far other representations for the different operators, a mapping between the DMM operations and Groove productions was used in order to find the Groove name of a production for a given operation.

> For every referenced old value of an attribute a possibility to access this value that is defined in the Groove graph is created.
>
> For every instance of the type `LiteralExpression` an according Groove node is created that carries the value defined by the instance of the type `Literal` referenced by the instance of the type `LiteralExpression`.
>
> An instance of the type `OperationExpression` is transformed to a Groove production node–that is connected to the operands of the operation–,a production result node and an edge that connects these two Groove nodes. The name of the edge is chosen accordingly to the instance of the type `Operator` the belongs to the instance of the type `OperationExpression`.

Transformation directive 7: The transformation of expressions

6.4.2 Transformation of Conditions

Within DMM, conditions limit the matching of a nodes to objects. Thus, the object within the host graph must have the type defined by the node, must obey the role[26] and beside other constraints must additionally fulfill all the conditions that are defined within the node. Thus, conditions are checked per object if they are fulfilled or not. Accordingly, a condition must have a tight relationship to the object, respectively to the node on which it is defined.

A condition itself is nothing else than an expression that must evaluate to a value of the data type `boolean`. If a condition evaluates to the `boolean` value `true`, the condition is fulfilled–for a given object–, and if this holds for all conditions, the rule matches the host graph and can be applied. How this meaning of conditions within DMM is mapped to according Groove constructs

26 Must either exist or not exist within the host graph.

and structures is depicted in this subsection.

Because Groove does not support conditions directly, some additional Groove structures are created in order to achieve the same meaning. First of all, the result of the expression used within the condition must be equal to the `boolean` value `true`. This can be ensured by an additional Groove production that takes the result of the expression and the `boolean` value `true` as operands and checks the equality between these two values. The result is checked to be the `boolean` value `true`. This can be seen at the marker 4 within Figure 42. In this case the same boolean value `true` can be reused as an operand and as the result of this additional production.

The reason why this literal is connected to the node on which the condition is defined via the edge with the name `DMM_conditionBinder` will be explained in the following paragraphs.

As already mentioned, a condition within DMM belongs to a node, and limits the matching possibilities of that node. This must be somehow reflected within the Groove rule as well. Now, think of a condition that does not make any statements about attributes that belong to the node on which the condition is defined. For this, think of the artificial condition `100 <= 200`. Although this condition does not make any sense at all, it is a good and simple example for the description of the necessity of some additional structures within the Groove rules.

By the transformation described so far, this condition would result in the structure of productions that would not be connected to the node on which the condition is defined in any way. The result of the transformation is shown in Figure 43. Only two literals would be compared, but there would be no relationship to the node on which this condition is defined.

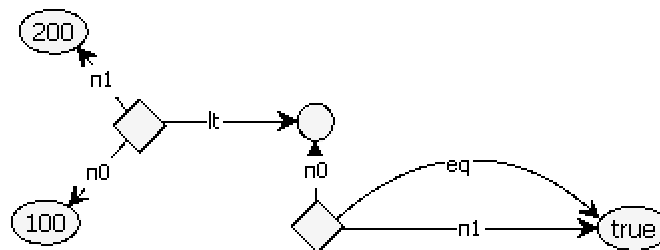

Figure 43: Transformation of the condition "100 <= 200"

Here the Condition Binder Concept *(CBC)* comes into play. The Condition Binder Concept–presented at marker 5 within the Figure 42–is a structure consisting of an edge with the name `DMM_conditionBinder` and a node, carrying the `boolean` value `true`. The result of every condition must be this value and the second operand of the additional production that is created for every condition must also refer to this value.

Given this structure, the Groove node within the Groove graph must on the one hand fulfill the name constraints, given by the type name, the role constraints–must either exist or not exists–and on the other hand must meet all the constraints resulting from the conditions. A result of the use of this concept is that the condition `100 <= 200` is bound to the node on which it is defined. Thereby, the Groove node containing this condition matches the according node within the Groove graph only if the naming and role constraints are fulfilled and additionally the condition for this node is evaluated to the `boolean` value `true`.

This concept leads to a slight adjustment of the Groove graph that is created for a given DMM host graph: Every node within the Groove graph representing an object of the host graph has to have an additional edge with the name `DMM_conditionBinder`, referring to the value `true` of the `boolean` data type.

Justification of this concept. Although it might be possible to limit the usage of conditions to the ones that use at least one attribute of the node on which they are defined, this approach was

not taken by DMM. A result of this approach would be the elimination of the structure claimed by the CBC described above, as there would always be a binding for the condition via the required attribute. Even though conditions like `100 <= 200` do not make any sense, the abdication of the CBC would reduce the expressiveness of DMM significantly but unnecessarily.

Therefore, let us look at an example, where the usage of condition that does not make any statements about the attributes of the nodes to which the condition belong, makes sense. This example given in Figure 44 justifies the introduction of the CBC. In order to understand this DMM rule, another definition for a pair of arcs within a petri net must be given. A tuple of arcs `(a1, a2)` is called consecutive[27] if the weights of the arcs are equal and `a1` is an incoming arc and `a2` an outgoing one of the same transition. Consecutive pairs of arcs are a special pair of arcs that can take a certain amount of tokens from an input place and transmit the complete amount to an output place via a given transition.

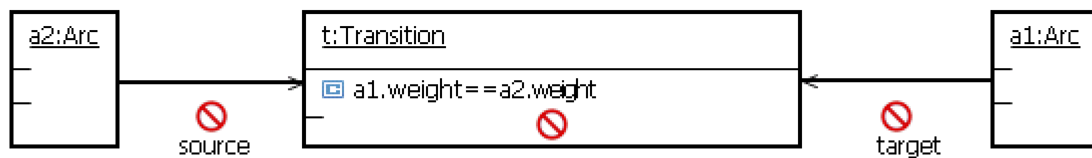

Figure 44: DMM Rule that matches all the non consecutive pairs of arcs

The task is now to find all non consecutive pairs of arcs, this means all tuples–or respectively all the pairs of arcs–that are not consecutive. This task can be solved with the help of the rule given in Figure 44. One special characteristic has to be mentioned about the given rule. It contains a node with a condition that does not use any of its own attributes–the type `Transition` does not even have any attributes–but makes a statement only about the value of attributes that belong to other nodes.

Apparently, every movement of the condition to another node would change the meaning of the rule. For instance, let us move the condition to the node `a2`. The resulting rule would match only to objects of the type `Arc` that would have the same value for their attribute `weight` and no transition in between. But then, not all the non consecutive pairs of arcs would be matched by this rule. E.g, arcs with different weights, which are non consecutive, would not be matched.

Therefore, in this case, it is necessary to place a condition on a node, whose attributes are not used by the condition. But then the necessity evolves to use the structure defined by the CBC.

Two additional Groove structures have to be created for every instance of the type `Condition`.

At first, every DMM node that contains conditions has to be transformed to a Groove node that contains the structures claimed by the CBC. This is a special literal node with the `boolean` value `true` and an edge with the name `DMM_conditionBinder`, connecting the literal to the Groove node.

Secondly, a final production has to be created for every condition that checks whether the result of the expression used within the condition is equal to the `boolean` value `true`. This production takes the production result of the expression and the literal node–created for the CBC–as input operands and connects the result of the additional production to the same Groove literal node.

Transformation directive 8: The transformation of conditions

27 The term “consecutive pair of arcs” is not defined in the domain of petri nets but was invented in order to demonstrate the usage of the CBC.

6.4.3 Transformation of Assignments

This subsection illuminates the transformation of assignments used within DMM rules to structures within the according Groove rules. In general, Groove has no special construct or concept that is equivalent to that of DMM assignments. But with the help of the deletion and addition of edges a pretty similar behavior can be achieved and thus assignments can be transformed to Groove.

To demonstrate this, the example DMM rule given in Figure 6 on page 15 is reused in an extended fashion. These extensions were performed in order to demonstrate the transformation of the Two Value Concept *(TVC)*. The TVC is described in subsection 3.1 on 15. The example DMM rule given in Figure 45 contains a DMM rule that specifies an addition of the number of tokens situated at a given place by the weight of an incoming arc. Furthermore, another artificial assignment was added that sets the `capacity` attribute to the new number of tokens increased by `1`. The corresponding Groove rule is shown in Figure 46.

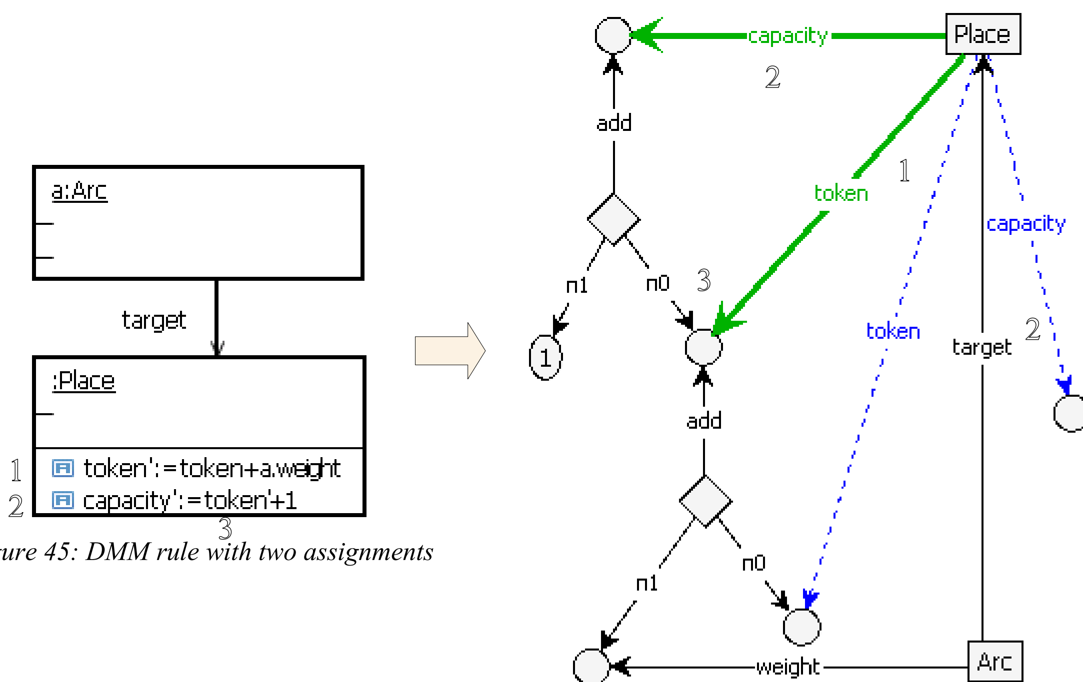

Figure 45: DMM rule with two assignments

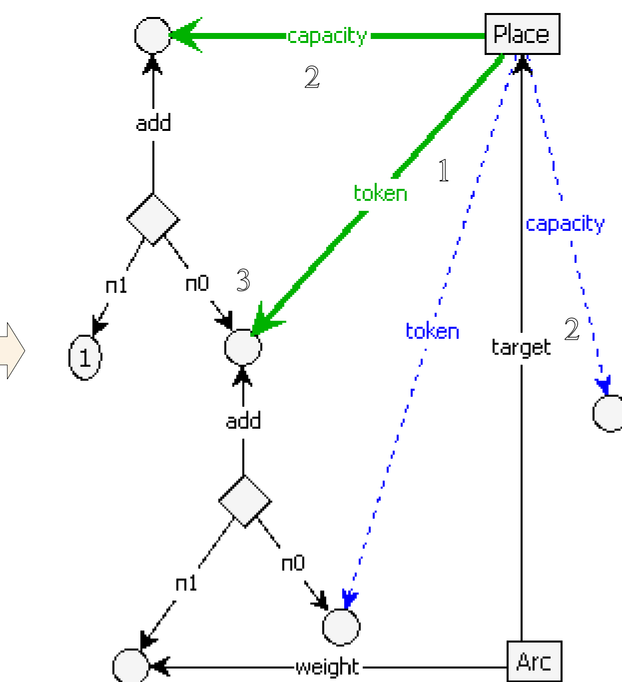

Figure 46: According Groove rule, representing an assignment

The Groove rule shown in Figure 26 has already given some hints on how assignments could be realized. Based on these hints, the transformation of assignments to Groove structures can be created.

First of all, every assignment to an attribute has to overwrite the old value of the attribute. This means that the old value has to be deleted explicitly and the new one created explicitly within the according Groove graph. Otherwise, more than one edge having the same name–that represents an attribute–originating from the same source node can exist within the Groove graph. This would lead to different ambiguous values for one attribute.

The Groove structure achieving this can be seen at marker 1, shown in Figure 46. This Groove rule states that the edge with the name `token` must be deleted–referring to the old value of the attribute–and a new edge to the result of the expression having exactly the same name `token` must be created. A similar structure is created for the assignment of the new value of the attribute `capacity`. This can be seen at marker 2.

There is only one exception to this transformation. For instances of the type `Assignment` that

are defined on nodes with the role `create` the old value of an attribute does not have to be deleted explicitly, since the attributes belonging to these nodes, respectively objects, do not have an old value.

One special aspect must be mentioned for the input operands of the addition that defines the new value of the `capacity` attribute. One of the operands is the new value of the attribute `token` itself. In order to use this new value, the production result node that corresponds to the expression defining the new value of the attribute `token` is used. This production result node is utilized as an operand of the Groove structure that represents the expression `capacity':=token'+1`. This described structure can be found at marker 3.

Because of this usage of the result of an production that defines the new value of another attribute, the already defined TVC can be completely transformed to according Groove structures.

Every instance of the type `Assignment` is transformed to a Groove structure that deletes the Groove edge referring to the old value of the attribute and creates a new edge to the new value of the attribute. Because the new value is defined by an expression that was already transformed before, this results in the creation of an edge–having the name of the attribute–to the production result that was created for the whole expression before.

Only assignments defined on nodes with the role `create`, the old value of an attribute does not have to be deleted.

Transformation directive 9: The transformation of assignments

6.4.4 Additional Treatments

The three transformations that were described in the preceding three subsection are nearly everything that had to be done in order to augment and adjust the transformation of the extended DMM rules to Groove rules. Nevertheless, some minor adjustments and special treatments had to be performed in order to obey the meaning of the different node roles or handle the correct quantification of a node with respect to its conditions and assignments.

This subsection depicts three aspects of the transformation that were treated in a particular way. It starts by giving hints on the transformation of nodes with the role `create` in subsection 6.4.4.1. Thereafter, subsection 6.4.4.2 describes how nodes of the role `not exists` and their conditions and assignments are modified. Finally, some modifications to the transformation of quantified nodes are depicted in 6.4.4.3.

6.4.4.1 Handling Nodes with the Role "create"

If a node has the role `create` within a DMM rule, it does not suffice to create only a corresponding object within the host graph. Especially the attributes need some more efforts. The object within the host graph can really be seen as an object used within an object oriented programming language. Similarly to the creation of objects within object oriented programming languages, where the constructor initializes all the attributes with default values and all the references with `null`, newly created objects within the host graph have to be initialized in a similar way.

This subsection relies on the example rule given in Figure 47 or respectively Figure 48. The DMM rule within the first figure states that a new object of the type `Place` has to be created in the host graph. The according Groove rule must ensure the creation of two additional structures.

The first structure states that all the attributes have to be created and initialized with default values according to their data type. This means that a Groove edge has to be created for each at-

tribute that belongs to the node with the role `create`, having exactly the same name like the attribute. This structure can be seen at marker 1.

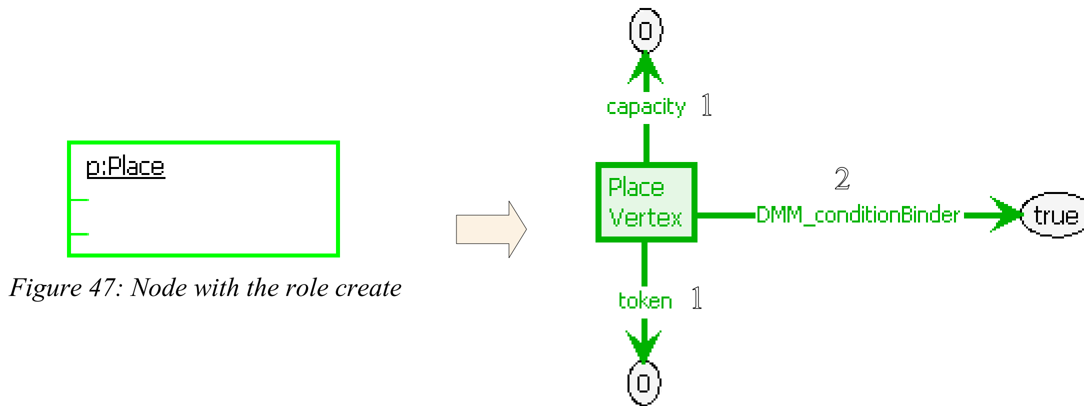

Figure 47: Node with the role create

Figure 48: According Groove rule, representing the creation of an object

Secondly, the structure used by the Condition Binder Concept *(CBC)* has to be created. This structure is visualized at marker 2. As explained in 6.4.2 on page 83, all the conditions defined on a node are bound to the corresponding Groove node via an additional edge with the name `DMM_conditionBinder` and an additional node carrying the `boolean` value `true`. Thus, in order to match the Groove graph like it was described in subsection 6.4.2, every node within the Groove graph must have this structure. Consequently, if a new Groove node–representing an object–is created, the structure claimed by the CBC has to be created for this node.

It might of course be possible to assign a particular value to an attribute belonging to a newly created Groove node. Therefore, the production result of the expression used within the assignment states the value of the newly created attribute.

Nodes with the role `destroy` and the deletion of objects from the host graph does not entail such a special handling. The graph transformation framework Groove deletes the outgoing Groove edges of a Groove node automatically and does not preserve values of any data type that do not have any incoming edges. Thus all the edges, representing the attributes, and additionally the Groove nodes representing the values of these attributes are deleted implicitly when the Groove node that contains these attributes is deleted. Similarly, the structure required by the CBC is removed automatically if the according Groove node is deleted.

For every DMM node with the role `create`, two additional Groove structures are created.

The first structure creates an edge for each attribute that belongs to the DMM node, originating in the Groove node that represents the DMM node under consideration. The other end of each edge is connected to a Groove node that represents the default value of each attribute. The name of each edge is set to the name of the corresponding attribute.

The second structure creates the edge and the Groove node claimed by the CBC. The name of the edge is set to `DMM_conditionBinder` and the node is set to carry the `boolean` value `true`. The additional edge connects the Groove node that corresponds to the DMM node with the role `create` and the Groove node representing the `boolean` literal.

Transformation directive 10: The transformation of nodes with the role create

6.4.4.2 Handling Nodes with the Role “not exists”

Besides the additional treatment of nodes with the role `create`, some additional extensions were made to the transformation of DMM nodes with the role `not exists`. As nodes with the role

not exists are not allowed to have any assignments–see section 3.1 on page 16–, only conditions defined on this nodes affect the transformation to Groove and will be presented in the following paragraphs.

As depicted in section 3.1, conditions that are defined on nodes with the role not exists state which objects within the host graph are not allowed to exist. Therefore, let us assume that there is a DMM node with the role not exists, with the type Place and an additional condition that is defined on the node stating token == 100. Then the rule might still match the host graph if objects of the type Place exist, as long as the token attribute does not have the value 100. But, if the object of the type Place with the attribute value of token set to 100 is found, the rule does not match the host graph. Thus, conditions defined on nodes with the role not exists limit the kind of objects that are not allowed to be present within host graphs.

In order to illustrate the transformation of nodes with the role not exists and conditions defined on these nodes, let us look at the DMM rule given in Figure 44 on page 85. The rule that is shown in that figure matches all the non consecutive tuples. The resulting Groove rule is visually presented in Figure 49.

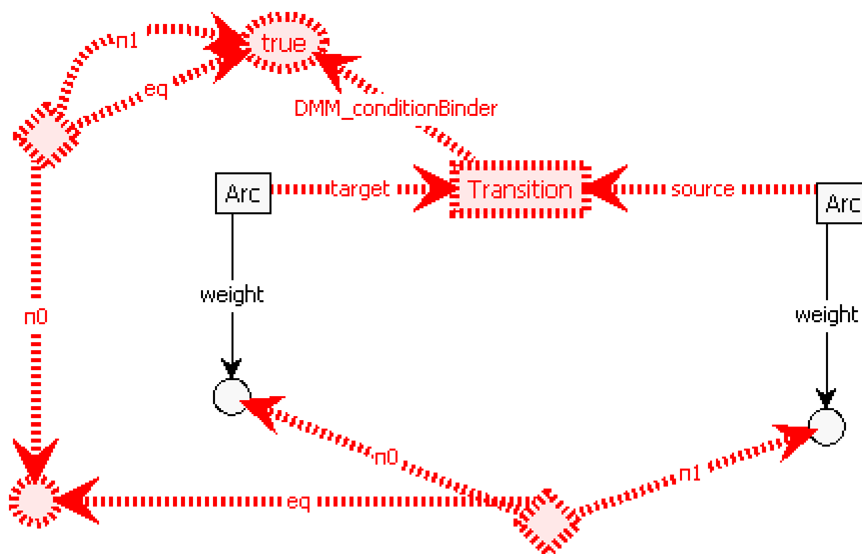

Figure 49: According Groove rule, matching all the non consecutive pairs of arcs

At first glance, nearly everything is not allowed to be present within the Groove graph such that the given Groove rule matches. But, let us have a closer look at the rule and investigate the meaning of it. It states that at least two Groove nodes having the self edge with the name Arc must be present within the Groove graph. Furthermore, as the value of the attribute weight is used, an edge with this name and a round node–representing the unknown value–must exist within the Groove graph.

The red Groove nodes state that no node with the name Transition is allowed to be present. Furthermore, this node must not have an outgoing edge with the name DMM_conditionBinder to a node carrying the boolean value true. Finally, this node is not allowed to be the result of the comparison between the values of the two attributes with the name weight. Thus, if any node of this not exists structure is not present within the Groove graph, the rule can be applied. Consequently, if a consecutive pair of arcs is tried to be matched, the structure that is painted red will be present and thus the rule will be not applicable. By this, all the consecutive pairs of arcs are not matched.

Technically, the not existence of a node is achieved by adding a self edge with the Groove aspect value that states that the given self edge is not allowed to be present within the Groove graph.

Besides the node that represents the DMM node with the role `not exists`, the structure resulting from the transformation of the condition has to have the same Groove aspect value.

> For every node with the role `not exists`, the whole structure resulting from conditions defined within this node is modified in the following way:
>
> A special self edge is added to each Groove node–either representing a production, a production result, a literal or the value of an attribute–that results from the transformation of the conditions. This self edge has the Groove aspect value that states that the particular Groove edge, or respectively Groove node, is not allowed to be present within the Groove graph.

Transformation directive 11: The transformation of nodes with the role not exists

6.4.4.3 Handling Quantified Nodes

This final subsection depicts the adjustment of the transformation of quantified nodes. These nodes were described in 2.3.2 on page 10. This subsection bases heavily on the transformation of quantified nodes without conditions and assignments that was illuminated in 6.1.1 on page 69.

The augmented transformation is vividly explained with the help of a part of the example given in Figure 7 on page 18. The rule shown in that figure describes when a transition is allowed to fire and what happens if it fires. It uses the extended runtime metamodel of the running example, containing attributes for the tokens, capacities and weights. Note that the example used for this subsection is only a part of the example given in that figure.

Figure 50 shows a rule that ensures, if matched, that all the input places of a transition have more tokens then the weight of the arc that connects this input place to the transition. The interesting point is that the condition is defined on a nested node and thus has to be checked for fulfillment for different objects within the host graph. Similar to the transformation of quantified nodes for which a Groove node connected to a Groove quantification node is created, the structures created for the conditions and assignments defined on quantified nodes have to be bound somehow to the same quantification node.

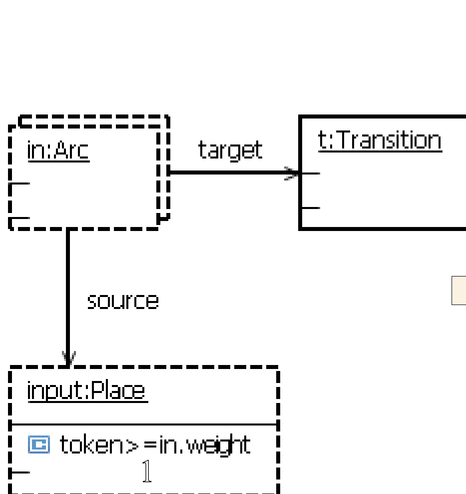

Figure 50: Nested node with condition

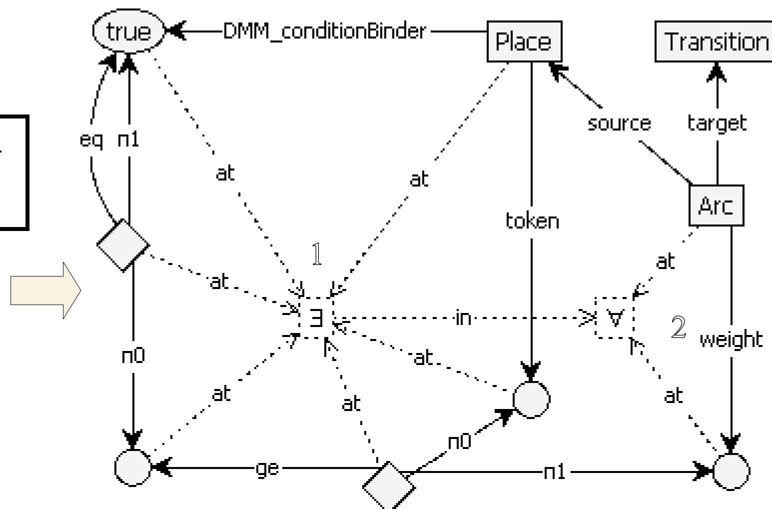

Figure 51: According Groove rule, representing a nested node with condition

At first, one might think that the binding is achieved via the CBC or respectively via the attribute to which the new value is assigned. But, this binding does not suffice. Here another kind of binding for the Groove structures resulting from conditions and assignments has to be created.

To give a vivid example, let us look at the nested node `input:Place` given in Figure 50. The

resulting Groove rule is shown in Figure 51. This nested node is transformed to a Groove node that is connected via an edge with the name `at` to an ∃ node. In order to express–within Groove– that a production should be evaluated in the context of such a Groove node this production has to be bound to the same ∃ node as well. This is done with the help of the same kind of edges as it was done in the case of the Groove node under consideration. An additional edge with the name `at` is created for each production and each production result resulting from the transformation of the condition `token >= in.weight`. This can be seen at marker 1.

Besides conditions, constructs like assignments and attributes have to be bound to the according quantification node within Groove in the same way. An example can be seen at marker 2 in Figure 51. Here the attribute `weight` belonging to the node with the name `Arc` is bound to the ∀ node. This means that every node with the name `Arc` has an own value for the attribute `weight`. These values do not necessarily have to be equal.

The Groove structure resulting from the transformation of attributes, conditions and assignments that belong to quantified nodes is bound in the following way:

An edge with the name `at` is created from each production, production result, literal or value of attribute to a Groove quantification node. Which quantification node is chosen depends on the quantification of the according DMM node on which the corresponding attribute, condition or assignment is defined.

If, on the one hand, such a construct belongs to a node that belongs to a uqs cluster, the resulting structure is bound to the according ∀ node. If, on the other hand, it belongs to a node that itself belongs to a nested cluster, the resulting structure is bound to the according ∃ node.

Transformation directive 12: The transformation of quantified nodes

7 Conclusion and Outlook

This final chapter gives a brief overview over the changes made to DMM and its tooling as a result of the introduction of attributes to this modeling language. It states to which extent goals described in section 1.2 on page 2 were achieved, which problems remain open and which aspects can be extended. These descriptions are given in section 7.1 and 7.2.

7.1 Conclusion

Due to the introduction of attributes and the according concepts–like conditions and assignments–, different adjustments were performed to the semantics of DMM constructs and concepts, and different modifications and extensions were incorporated into the available tooling.

First of all, an expression language was elaborated that defines how the attributes and literals are used in order to express conditions and assignments. Therefore, an extendable grammar was defined and well-formedness rules were formulated. A lot of additional constraints were necessary in order to use the concepts of attributes, conditions, and assignments correctly. During these elaborations, the semantics of a lot of other concepts of DMM were illuminated and formulated precisely in order to avoid ambiguities. For instance, the concepts concerning quantified nodes were reinvented and the concepts of invocations reformulated.

Secondly, the DMM metamodel itself was extended in order to be able to abstract the entered sequence of characters and to create a processable representation of these sequences. Here, the DMM metamodel was extended in a special way, making parts of this metamodel reusable within other metamodels. Especially the separation of types concerning the expression language to a separate package achieves this goal of reusability.

By the implementation of the concepts related to the introduction of attributes and the incorporation into the different available tools, all the theoretical elaborations were made practical. Therefore a parser and a new validation framework were created. Furthermore additional extensions were made to the graphical editor and the transformation to Groove. Thus, all newly introduced constructs to DMM can now be used within the graphical editor and are transformed to according Groove constructs. Consequently, tooling support is now present for all the newly introduced DMM constructs.

That is why it is by now possible to specify the dynamic semantics of modeling languages whose metamodels make use of attributes. This increases the expressiveness of DMM to a great extent. Without attributes, it would be much harder–if not impossible–to define the dynamic semantics of the extended running example within this bachelor thesis. The firing if a transition for petri nets that consist of arbitrary weights and capacities can now be easily specified without ambiguities.

Thus, as a conclusion, all goals described in section 1.2 have been achieved and all changes to the DMM tools have been performed. Therefore, the concepts of attributes, conditions and assignments is now completely integrated in DMM.

7.2 Outlook

Nevertheless, a lot of different extensions can still be made to the Dynamic Meta Modeling and its tooling. Due to the extensibility of the developed grammar, new operations can be added easily, nearly without changing the according tools. Furthermore, extensions in terms of the addition of well-formedness rules and the adjustment of the traversal of the well-formedness rules can

simply be made within the validation framework that was created from scratch.

Finally, different enhancements to the graphical editor can be added. For instance, the automatic completion of attribute names would be a feature that would improve the usability of the graphical editor to a great extent.

Besides these rather technical improvements, the current concepts and the meaning of the current constructs can be broadened and further constructs and concepts added to DMM. At this place, two different examples shall be given.

At first, it is a common usage in case of invocations within DMM to pass nodes as parameters to other rules. Similarly, it might be imaginable to extend this concept by enabling to pass attributes to invoking rules. This would result in a new application area: It would be possible to use DMM for the specification of algorithms in a much more natural way.

Besides this possibility to extend DMM, it might also be possible to add the concept of local variables. These are constructs that can be used exactly as attributes, but do not have to be defined within the runtime metamodel. Therefore, it would be possible to reuse the values of local variables without adding these additional pieces of information to the runtime metamodel.

This shows that there are still different possibilities to improve the Dynamic Meta Modeling formalism and its available tooling.

List of Figures

Figure 1: Relationship between the different DMM and related elements....................................6

Figure 2: Metamodel and runtime metamodel for petri nets...7

Figure 3: Exemplary DMM rule that specifies the firing of a transition..8

Figure 4: Runtime metamodel for petri nets, including attributes...13

Figure 5: DMM rule that demonstrates the usage of conditions..15

Figure 6: DMM rule that demonstrates the usage of assignments..15

Figure 7: DMM rule for the specification of a firing transition...18

Figure 8: Overview of the modifications of the DMM metamodel...33

Figure 9: The metamodel for the expression language..34

Figure 10: The new visual representation of nodes, conditions and assignments.........................36

Figure 11: Visual representation of the collaboration between the generated software components 38

Figure 12: The overview of the validation framework..44

Figure 13: The structure of the validator.result package...45

Figure 14: The overview of the validator.constraints package...47

Figure 15: Relation between constraints and constraint lists...48

Figure 16: Dependencies between the different constraint lists..49

Figure 17: Overview of the validator package...51

Figure 18: Eclipse's Architecture...53

Figure 19: Layers that are defined by the Meta Object Facility...54

Figure 20: Lifecycle used within Eclipse Modeling Framework...55

Figure 21: Models used within the Graphical Modeling Framework...57

Figure 22: Visualization of the priority calculation..61

Figure 23: Mapping Model instance for the definition of conditions...63

Figure 24: Groove rule demonstrating the usage of aspects...69

Figure 25: Groove rule that demonstrates the usage of quantification nodes...............................70

Figure 26: Groove rule containing a possibility to express attributes..71

Figure 27: A DMM rule containing nodes and edges with different roles...................................74

Figure 28: The resulting Groove rule, demonstrating the transformation of DMM nodes and edges...74

Figure 29: Firing of a transition without capacities and weights...75

Figure 30: The according Groove rule using Groove quantification nodes.................................75

Figure 31: Exemplary DIS with the invoked rule deleteToken..77

Figure 32: Necessary Groove structure to match invoked Groove rule..77
Figure 33: Empty DIS..77
Figure 34: Demonstration of invocations 79
Figure 35: According Groove structure, representing an invocation...................................79
Figure 36: Invoked Smallstep rule...80
Figure 37: According Groove rule...80
Figure 38: Invocation on quantified node..81
Figure 39: According Groove rule, with invocation on quantified node...............................81
Figure 40: Attributes within the Groove graphs...82
Figure 41: DMM Rule with a condition...83
Figure 42: According Groove rule, representing a condition...83
Figure 43: Transformation of the condition “100 <= 200”..84
Figure 44: DMM Rule that matches all the non consecutive pairs of arcs..............................85
Figure 45: DMM rule with two assignments..86
Figure 46: According Groove rule, representing an assignment...86
Figure 47: Node with the role create...88
Figure 48: According Groove rule, representing the creation of an object.............................88
Figure 49: According Groove rule, matching all the non consecutive pairs of arcs.................89
Figure 50: Nested node with condition..90
Figure 51: According Groove rule, representing a nested node with condition.....................90

List of Tables

Table 1: Interdependence between the node's role and the newly introduced constructs..............16
Table 2: Possible references to attributes that belong to quantified nodes...................................18
Table 3: Exemplary elements on each layer of the definition of the expression language............20
Table 4: Summarization of all possible tokens and their regular expressions..............................21
Table 5: Priorities for all the different operators...23
Table 6: Summarization of all possible operations and their characteristics................................24
Table 7: The grammar for the elaborated expression language...26
Table 8: Table 8: The list of well-formedness rules..32
Table 9: Relationship between DMM and Groove..72

List of Code extracts

Code extract 1: Exemplary token definition within JavaCC...40

Code extract 2: An MLN that defines how the nonterminal <orExp> should be derived.............42
Code extract 3: Finding a constraint according to the ID of an audit rule....................................60

List of Transformation directives

Transformation directive 1: Preliminary actions before the actual transformation.......................73
Transformation directive 2: Transformation of the basic constructs..75
Transformation directive 3: Transformation of quantified nodes...76
Transformation directive 4: Creation of the DMM Invocation Stack...78
Transformation directive 5: Transformation of invoking and invoked rules................................80
Transformation directive 6: Transformation of invocations with a quantified target node...........81
Transformation directive 7: The transformation of expressions...83
Transformation directive 8: The transformation of conditions...85
Transformation directive 9: The transformation of assignments..87
Transformation directive 10: The transformation of nodes with the role “create”........................88
Transformation directive 11: The transformation of nodes with the role “not exists”..................90
Transformation directive 12: The transformation of quantified nodes..91

Bibliography

[1] Harel, Davic; Rumpe, Bernhard, *Modeling Languages: Syntax, Semantics and all that Stuff*, 2004

[2] *UML 2.1.2 Superstructure Specification*, http://www.omg.org/docs/formal/07-11-02.pdf, 25.05.2008

[3] *OCL 2.0 Specification*, http://www.omg.org/docs/ptc/05-06-06.pdf, 28.05.2008

[4] Hausmann, J.H., *Dynamic Meta Modelling*, PhD thesis, University of Paderborn, 2005

[5] Röhs, Malte, *A Visual Editor for Semantics Specifications - Using the Eclipse Graphical Modeling Framework*, Bachelor thesis, University of Paderborn, 2008

[6] *Graphical Modelling Framework*, http://wiki.eclipse.org/GMF, 25.05.2008

[7] *GRaphs for Object-Oriented VErification*, http://groove.cs.utwente.nl/groove-home/, 11.09.2008

[8] Ehrig, Hartmut, *Attributed Graphs and Typing: Reationshi between Different Representations*, Bulletin of the EATCS, Volume 82, Pages 175-190, 2004

[9] Baumgarten, Bernd, *Petri-Netze - Grundlagen und Anwendungen*, Spektrum Akademischer Verlag, Heidelberg, 1996

[10] Daniela Fisseler, *Improving the inheritence concept of Dynamic Meta Modelling*, Diploma thesis, University of Paderborn, 2007

[11] Christian Soltenborn, *Analysis of UML Workflow diagrams with Dynamic Meta Modeling techniques*, Diploma thesis, University of Paderborn, 30.06.2006

[12] Von Alfred V. Aho; Jeffrey D Ullmann; Ravi Seth; Monica S. Lam, *Compiler: Prinzipien, Techniken und Werkzeuge*, Pearson Education Deutschland, 30.01.2008

[13] *Java CC*, https://javacc.dev.java.net/, 25.05.2008

[14] *Introduction to JavaCC, Tutorial*, http://www.engr.mun.ca/~theo/JavaCC-Tutorial/javacc-tutorial.pdf, 18.07.2008

[15] Uwe Kastens; William McCastline Waite; Anthony M. Sloane, *Generating Software from Specifications*, Jones & Bartlett Publ, 08.01.2007

[16] *EMF Validation Framework*, http://www.eclipse.org/modeling/emf/?project=validation, 27.07.08

[17] *Eclipse*, http://www.eclipse.org/, 04.09.08

[18] David Carlson, *Eclipse Distilled*, Addison-Wesley Professional, 24.02.2005

[19] *The Eclipse 3.0 platform: Adopting OSGi technology*, http://www.research.ibm.com/journal/sj/442/gruber.pdf, 04.09.08

[20] *Meta Object Facility (MOF) Core Specification, Version 2.0*, http://www.omg.org/docs/formal/06-01-01.pdf, 19.08.08

[21] Dave Steinberg; Frank Budinsky; Marcelo Paternostro; Ed Merks, *EMF: Eclipse Modeling Framework (2nd Edition) - Preliminary Version*, EMF: Eclipse Modeling Framework (2nd Edition, 02.02.2009

[22] *Rational Rose, Product Site*, http://www-01.ibm.com/software/rational/?S_TACT=105AGY59&S_CMP=13&ca=dtl-13, 04.09.08

[23] *Eclipse Development, using the Graphical Editing Framework and the Eclipse Modeling Framework*, http://www.redbooks.ibm.com/redbooks/pdfs/sg246302.pdf, 20.09.08

[24] Arend Rensink, Harmen Kastenberg, and Tom Staijen, *User Manual for the GROOVE Tool Set*, University of Twente, 10.04.2007

[25] Rheker, Thomas, *A Bidirectional Transformation between EMF Models and Typed Graphs*, Bachelor thesis, University of Paderborn, 2008

Eidesstattliche Erklärung

Ich erkläre hiermit an Eides statt, dass ich die vorliegende Arbeit selbständig angefertigt habe; die aus fremden Quellen direkt oder indirekt übernommenen Gedanken sind als solche kenntlich gemacht.

Die Arbeit wurde bisher keiner anderen Prüfungsbehörde vorgelegt und auch noch nicht veröffentlicht.

Ich verpflichte mich, meine Bachelorarbeit den Richtlinien der Studienordnung entsprechend aufzubewahren und sie auf Verlangen jederzeit herauszugeben.

Paderborn, Oktober 2008

____
Eduard Bauer