Using rule overriding to improve reusability and understandability of Dynamic Meta Modeling specifications

Christian Soltenborn *, Gregor Engels
University of Paderborn, Warburger Straße 100, 33098 Paderborn, Germany

A R T I C L E  I N F O

Article history:
Received 14 May 2010
Received in revised form 25 October 2010
Accepted 8 December 2010
Available online 12 January 2011

Keywords:
Semantics
Metamodel
Dynamic Meta Modeling
Graph transformation
Inheritance

A B S T R A C T

Dynamic Meta Modeling (DMM) is a visual semantics specification technique targeted at languages based on a metamodel. A DMM specification consists of a runtime metamodel and operational rules which describe how instances of the runtime metamodel change over time. A known deficiency of the DMM approach is that it does not support the refinement of a DMM specification, e.g., in the case of defining the semantics for a refined and extended domain-specific language (DSL). Up to now, DMM specifications could only be reused by adding or removing DMM rules.

In this paper, we enhance DMM such that DMM rules can override other DMM rules, similar to a method being overridden in a subclass, and we show how rule overriding can be realized with the graph transformation tool GROOVE. We argue that rule overriding does not only have positive impact on reusability, but also improves the intuitive understandability of DMM semantics specifications.

© 2010 Elsevier Ltd. All rights reserved.

1. Introduction

In today’s software engineering world, typically a lot of people with different backgrounds are involved in a software project. This is due to the increased complexity of software, but also because software systems are used in basically every area; as a consequence, software engineers have to work together with domain experts from different fields.

One way to support communication between those differently skilled groups of people is to use visual modeling languages. This approach works best if the language that is used supports concepts of the actual domain; so-called Domain-Specific Languages (DSLs) might even enable the domain experts to do large parts of the modeling themselves (in contrast to a general-purpose language like the Unified Modeling Language (UML) [1]).

One requirement for effectively using DSLs is a precise definition of the language’s semantics. This is often the case for the static semantics of a language (the UML is no exception here): Valid sentences of a language are e.g., described by means of metamodels, i.e., class diagrams describing the language’s concepts as well as their relation to each other. Additional, context-sensitive constraints—which cannot be expressed with class diagram constructs—are added, e.g., using a language like the Object Constraint Language (OCL) [2].

For the definition of the dynamic semantics of a language, the situation is not as good. Often, natural language is used to describe how models of a certain language behave. For instance, the UML specification states that the semantics of Activities is based on token flow, but this information is only contained in the text accompanying the definition of the static semantics.

Such an informal description of the language’s behavior almost always leaves room for different interpretations and is therefore in conflict with the requirement
that the language's meaning needs to be defined precisely. Additionally, other requirements which are often put on DSLs are seriously affected: First, to effectively work with a DSL, a sophisticated tool support is needed, and second, the quality of complex models cannot be checked manually; therefore, the language should be analyzable.

These requirements can only be fulfilled by a language whose syntax and semantics are specified formally. Unfortunately, formal specifications often lead to another problem: They are difficult to understand for language users who are not familiar with the underlying formalism. This has two severe drawbacks: First, the language engineer's job of creating the semantics specification is more difficult, and second, end users of the language cannot refer to the semantics specification as a reference when, e.g., discussing the semantics of a particular element.

Consequently, a semantics specification technique is needed which is not only formal, but is also easily understandable at least for the target language users (users who are at least familiar with the language's metamodel). This is where Dynamic Meta Modeling (DMM) [3,4] comes into play.

DMM specifications are easily understandable for a number of reasons: Firstly, as we will see later, a DMM rule is an (annotated) object diagram instantiating the runtime metamodel; this visual and familiar appearance has proved to be easy to comprehend [5]. Secondly, DMM supports a number of object-oriented concepts, which are expected to be well-known by the target language users.

DSLs are often developed incrementally, i.e., an existing DSL is modified to suit the needs of another but related domain. Therefore, if a DSL has been equipped with a formal specification of syntax and semantics, it is desirable to be able to reuse that specification for the modified DSL.

In its current state, existing DMM specifications can be reused as a base for similar languages, but only with strong restrictions: DMM rules can be added or removed from a DMM specification, but rules cannot refine other rules (similar to a class refining methods of its superclasses).

It turned out that this severely hampers reusability of DMM specifications. As a consequence, we decided to introduce a notion of rule overriding into the DMM language. Within this paper, we introduce this concept of rule overriding.

In the next section, we give a brief overview on related work and point out the differences between the existing approaches and ours. Section 3 provides an introduction to DMM in its current state, illustrating the different parts of a DMM specification. Since we expect UML activities to be familiar to the readers of our paper, we use a simplified version of that language as a running example. Section 4 then introduces and discusses our notion of rule overriding. The last section concludes and gives an outlook on future work.

This paper is an enhanced version of [6]. In particular, a detailed description of the implementation of rule overriding using the GROOVE toolset [7] has been added to Section 4, more elaborated examples have been provided, and the related work has been enhanced.

2. Related work

This section will discuss research which is related to ours. There are several approaches to define the semantics of visual modeling languages by means of (typed) graph transformations (see e.g., [8,9]). We first do a brief comparison to generic frameworks suited for semantics specification. Next, we will discuss work which provides some support for reusability, such as modularization, prioritization, or support for inheritance. Finally, we will compare DMM to common rule-based model transformation approaches.

One approach to specifying behavioral semantics of visual languages has been suggested by Chen et al. [10]: They define the semantics of a language by anchoring it to another language (which they call semantic unit) whose semantics is well-defined. The advantage is that the language engineer does not have to come up with an own semantics for his language – instead, he only has to map his language constructs to the ones of the semantic unit. However, this only works if the two languages are closely related to each other; consequently, the authors claim that an appropriate set of semantic units covering the needs of the most common behavioral languages is yet to be identified. In contrast, DMM leaves the burden of specifying the operational semantics to the language engineer, but provides more flexibility, since there are no restrictions on the semantic domain (i.e., the runtime metamodel).

Bottori et al. [11,12] define action patterns and their generalization generative patterns for the sake of specifying “semantic building blocks”, for instance in the context of token flow. A target language's metamodel is then mapped to a metamodel of the semantic domain, and the defined patterns are instantiated in the context of elements of the semantic domain; the result are rules describing the elements' semantics (i.e., rules which make elements of the target language flow like tokens). Again, the approach only works if appropriate patterns are available which reflect the target language's intended semantics.

We now turn to approaches which directly or indirectly support reusability. The first area of interest we discuss is notions of inheritance. In de Lara et al. [13], show how to integrate attributed graph transformations with node type inheritance, therefore allowing to formulate abstract graph transformation rules (i.e., rules which contain abstract nodes). The resulting specifications tend to be more compact, since a rule containing abstract nodes might replace several rules which would otherwise have to be defined for each of the concrete subtypes. The resulting formalism does not provide support for the refinement of rules (and is therefore comparable with the expressiveness of the current state of DMM).

Ferreira et al. [14,15] develop a notion of typed graph transformations which supports several object-oriented features, including inheritance and polymorphism. They focus on delivering a framework which is as close to object-oriented systems described by e.g., Java code as possible, whereas the models targeted by DMM are expected to have a less complex semantics, since they
usually will be abstractions of concrete systems described by code. As such, DMM emphasizes on keeping the specification language simple.

Another tool to help with reusing existing semantics specifications is prioritization, since it gives additional control over the application of rules. The AGG toolset [16] supports the concept of rule layers: First, all (matching) rules of layer 0 will be applied followed by all rules of layer 1 and so on. This allows the implementation of a simple control flow of graph transformation. This mechanism could probably be used to realize the concept of rule overriding, but this has to be done “by hand”, i.e., the modeler has to manually add structures to the rules (which finally result in the desired overriding behavior). This is not necessary for DMM, since that structure will automatically be built during the transformation process of a DMM specification into a GROOVE grammar (as we will see in Section 4).

Another area of related work is modularization. In [17], Krewowski et al. introduce the concept of graph transformation units (GTUs) as a way to structure large graph transformation systems. In a nutshell, a GTU consists of a set of graph transformation rules, an optional import of other GTUs, and control conditions. Its semantics is defined by means of the interleaving semantics of all (own and imported) graph transformation rules. The control condition is used to reduce nondeterminism of the resulting graph transformation system, since only rule applications consistent with that condition are considered. GTUs obviously are a powerful means to define the semantics of modeling languages in a reusable way; however, we believe that in comparison to DMM, a considerable larger amount of knowledge of the language engineer is needed to benefit from its expressiveness, partly because the control conditions have to be delivered in addition to the rules.

A completely different approach has been suggested by Legros et al. [18]: They introduce generic and reflective graph transformation rules into the Fujaba/MOFLON [19] framework. These rules can combine elements of the model, the metamodel, and the metametamodel level, allowing to define rules which reflectively inspect a corresponding graph and act with respect to the available meta-information. The approach does not support reusability explicitly. However, it might give rise to specifications which are so general that modifying them is not even needed, although this was not the intention of the authors and is most likely not always possible.

Finally, we want to investigate model transformation languages, many of which have a lot of similarities to DMM: The transformations are often modeled by means of rules having a declarative as well as an imperative part.

For instance, the Eclipse projects Atlas Transformation Language (ATL) [20] and Epsilon Transformation Language (ETL) [21] allow to define model transformations based on the source and target language’s metamodels; they are both very similar to the Relations part of Query/View/Transformation (QVT) specification [22] provided by the Object Management Group (OMG). Each rule describes the transformation of a particular language element.

In the context of this paper, the most interesting feature of both languages is that a rule can extend another rule. However, in the case of ATL, the rule extension mechanism is a static one: The language’s compiler makes sure that the precondition of an extending rule becomes the union of the preconditions of all (transitively) extended rules, and that the extending rule’s behavior is an aggregation of all extended rules’ behaviors. ETL works very similar to ATL with respect to the rule extension semantics. Note that the semantics of the QVT rule extension mechanism is not precisely defined by the QVT specification (see e.g., [23]).

Another approach to specifying model transformations is the work by Guerra et al. [24]. They describe Triple Patterns, i.e., typed Triple Graph Grammar rules which are equipped with constraints on attributes of the involved objects. The rules are then translated into executable forward and backward graph transformation rules; the attributes’ values are computed by constraint solvers. The approach supports extensibility in the sense that the Triple Patterns are compositional: a triple graph satisfies two patterns in conjunction if it satisfies the two patterns separately. However, their approach does not have explicit support for reusability of specifications.

3. Dynamic Meta Modeling

In this section, we will give an introduction to DMM and we will discuss the object-oriented concepts supported by DMM in its current state. For this, we will use a DMM specification for a simplified notion of UML activities as an example. An overview of DMM is provided in Fig. 1.

DMM is targeted at languages having an abstract syntax which is defined by means of a metamodel as suggested by the OMG, i.e., a model describing models. The models, i.e., the sentences of the language must then be consistent with that metamodel. Often the MetaObject Facility (MOF) [25] is used for the specification of metamodels, which—from the expressiveness point of view—basically is a subset of UML class diagrams. To follow the OMG’s layered model, the language’s metamodel is on level M2, a concrete model (i.e., an object diagram consistent to the class diagram of level M2) is on level M1, and a behavior of the reality which is represented by the model (e.g., a business process in the case of UML Activities) is on level M0.

The abstract syntax of a simplified version of UML activities is depicted as Fig. 2. According to that model, an Activity consists of an arbitrary number of ActivityNodes and ActivityEdges which connect these nodes. There are different kinds of ActivityNodes, each having their own semantics (yet to be defined). Fig. 3 shows a simple Activity consisting of an InitialNode, a FlowFinalNode, two DecisionMergeNodes and two Actions, which are connected by ActivityEdges.

The semantic domain of a language is then specified using runtime metamodeling. This means that the semantic domain has its own metamodel, to which the abstract syntax metamodel is mapped. This metamodel is referred to as the runtime metamodel; it often is an enhanced version of the language’s syntax metamodel. For example, the Activity’s runtime metamodel is depicted as Fig. 4.
It has elements like ActivityExecution and Token, which allow to express states of execution of the Activity under consideration. These states are formed by the locations of the Tokens; an ActivityExecution takes care of executing an Activity by managing a set of Tokens.

DMM tries to achieve maximum understandability partly by reusing object-oriented concepts, which are expected to be well-known by the target language users. This is in fact the first case of reuse: The elements added for the sake of describing runtime information are described on the MOF class level and then instantiated (i.e., within state graphs and rules).

The dynamic semantics is then specified by developing a set of operational rules which describe how instances of the runtime metamodel change through time. For this, the instances are mapped to typed graphs (see e.g., [26]), i.e., graphs whose nodes are typed over the runtime metamodel.
The operational rules are then defined as graph transformation rules, working on the derived typed graphs.

As mentioned above, a DMM rule is a graph transformation rule. This means that it consists of a left-hand graph $G_L$ and a right-hand graph $G_R$. A graph transformation rule *matches* a host graph $G$ if a morphism between $G_L$ and $G$ can be found. If this is the case, the rule can be *applied* to $G$. This basically results in the occurrence of $G_L$ in $G$ being replaced by $G_R$, leading to a new graph $G_u$. DMM uses the *single pushout* (SPO) approach; this means that if a node is deleted, all attached edges are deleted automatically (in contrast to the *double pushout* (DPO) approach where the deletion of the attached edges would have to be modeled explicitly). See [8] for a more elaborated explanation of SPO and DPO.

DMM also supports *negative application conditions* (NAC)—structures which must not be present for a rule to match—and *universally quantified structures* (UQS). The latter allows to match all nodes which fulfill the specified structural requirements, independent of their actual number within the state graph; we will see examples for a NAC and a UQS below.

Let us investigate an example DMM rule named `initialNode.flow()`#, which is depicted as Fig. 5. It defines the semantics of the `InitialNode`, which is as follows: As soon as the Activity starts, all `InitialNodes` produce one `Token`. If the `InitialNode` has more than one outgoing `ActivityEdge`, an edge will be chosen nondeterministically. The `Token` then starts flowing through the Activity.

Before we show how this is modeled in rule `initialNode.flow()`#, note that the visual representation of DMM rules merges the rule’s left-hand and right-hand graphs into one graph. This is done by annotating elements only contained in the right-hand graph with `{create}` and elements only contained in the left-hand graph with `{destroy}`. NACs are annotated with a “stop sign” (see right started edge in Fig. 5).

Let us now investigate rule `initialNode.flow()`# shown in Fig. 5: The structure to the left makes sure that it matches for an `InitialNode` and an outgoing `ActivityEdge`. For every such combination, applying the rule will create a `Token`, and that `Token` will be owned by the `ActivityExecution` managing the `InitialNode`'s Activity. The result will be a new state for each combination, and the states will differ in the `ActivityEdge` carrying the `Token`.

Finally, the rule will match exactly once for each combination. This is ensured by the two started links in the middle of the rule: The one annotated with a stop sign is a NAC, therefore making sure that the rule only matches `InitialNodes` that do not have a started link. The other one is annotated with `{create}`, therefore creating a started link. Since in all new states, the
InitialNode will have a started link, and since no other rule ever deletes these links, the rule will never match again.

A second example DMM rule is depicted as Fig. 6. The rule's name is action.flow(); as expected, it defines the semantics of the Action elements. According to the UML specification, an Action can be executed if all its incoming ActivityEdges carry at least one Token. This is modelled by a UQS: The shape of the node typed ActivityEdge indicates that the rule matches for all ActivityEdges targeting the Action. The shape of the node typed Token indicates an existential quantification: all edges need to be associated with a distinct Token object. The UQS can therefore be read as: match an Action if all its incoming ActivityEdges are associated with at least one Token.

So far, DMM rules are not different from common typed graph transformation rules. The first (and most important) difference is that every DMM rule has a so-called context node. This is a distinguished node that can be seen to own the behavior the rule describes: the according node's type is annotated with \{cn\}. Consequently, a DMM rule will in general perform changes in the context of that node, i.e., on elements directly or indirectly related to it (this is no restriction, though). A DMM rule therefore has similarities to a method (which also typically applies changes in the context of the object it is invoked on). This similarity is strengthened by another enhancement to standard graph transformation rules: DMM rules come in two flavors: bigstep rules and smallstep rules. Bigstep rules (whose name ends with a hash sign #) are applied to a state graph as soon as they match, whereas smallstep rules need to be explicitly invoked by other rules to be applied. Therefore, a DMM rule has a (possibly empty) list of invocations of DMM smallstep rules.

As the names of our example rules show, they are both bigstep rules. Rule initialNode.flow()# does not contain any invocations, whereas rule action.flow()# contains an invocation execute(ae) on the node typed Action. Since the invocation mechanism is crucial to DMM as well as rule overriding, we will explain it in detail in the next section.

### 3.1. Rule invocation

In common graph transformation approaches, the modeler does not have explicit control over the application of the graph transformation rules. Instead, the rules have to be built such that they can only match "when it makes sense". Often it is not possible to perform all desired changes on a state graph within one rule. In such cases, one has to manually add triggers which enforce that one or more rules match and are applied.

In DMM, this problem is tackled by allowing both bigstep and smallstep rules to explicitly invoke smallstep rules. An invocation happens within the context of a certain node, the so-called target node. This node becomes the context node of the invoked rule within the application of that rule. Obviously, invoking a DMM rule on a target node is very similar to calling a method on an object.

Furthermore, smallstep rules can have parameters. Thereby, an invoking rule can pass a number of its nodes to an invoked rule for special treatment, just as objects can be passed when calling a method. Nodes serving as parameters of a node are annotated with \{p[nr]\}, where \[nr\] is the index of the parameter in the list of parameters a smallstep rule may have. The passed objects are bound to the parameter nodes of the rule when applying the invoked smallstep rule (as we will see in the next section). The invocation we have seen in rule action.flow()# (Fig. 6) passes its ActivityExecution node to the invoked execute(ae) smallstep rule.

Finally, every smallstep rule has an implicit signature, which is formed by the rule's name and the types of context node and parameters. An invocation within an invoking rule must be compatible to an existing smallstep rule, which means that the types of target node and parameter nodes must be the same as, or subtypes of,
the corresponding types of the invoked rule, and that the rule’s name is equal to the one of the invocation.

Note that there is one important difference between DMM rules and method calls on objects in an object-oriented language: For instance in Java, each method signature must be unique within its type, whereas it is often the case that a DMM rule set contains several rules with the same signature. These rules will typically differ in their left-hand graphs (and therefore cover different situations—this is the way conditions are modeled in DMM), but this does not have to be the case. For instance, we can model nondeterminism by using rules with the same left-hand graphs.

Let us investigate an example smallstep rule named `action.execute(ActivityExecution)` which is depicted as Fig. 7. The rule realizes a basic semantics for Actions by passing its single parameter node to a smallstep rule `action.processTokens(ActivityExecution)` whose goal is to delete a Token from each incoming ActivityEdge of the Action, and to create a Token on each of the outgoing ActivityEdges (this rule is not shown).

Note that visually, a smallstep rule can only be distinguished from a bigstep rule by means of the hash sign #. However, we will see below that there are other differences rising from the realization of the invocation mechanism.

Technically, invocations are realized by introducing an “invocation stack”. A bigstep rule can match whenever the stack is empty, whereas a smallstep rule can only match if an according invocation is on top of the stack. Note that we do not formalize the invocation stack in all details; however, the next section will give a much more precise idea on how the invocation stack is realized.

All together, invocations do not only give more control over the application of rules, but can (and should) also be used to decompose complex rules into smaller ones which are easier to understand and can be reused in other contexts (this has in fact been the purpose of introducing rule `action.processTokens(ActivityExecution)` mentioned above).

The invocation concept makes the modeler’s job significantly easier, but it has one drawback: A DMM invocation can fail. This happens in case an invocation is on top of the stack, but at that time, the state graph does not match any of the smallstep rules with an according signature. A DMM specification allowing for situations like this to occur is considered to be incorrect. DMM deals with this issue by providing a generic helper rule which only matches in that very situation and thus points the language engineer to his error.

Before we will see in the next section how DMM rules are actually executed, we want to collect the important facts within two definitions. We start with the definition of a DMM rule. Note that for the sake of simplicity, we do not provide all the details—the interested reader is pointed to [4].

**Definition 1 (DMM rule).** A DMM rule is a tuple \( R = (\text{name}, G_1, G_R, \text{NACs}, \text{contextNode}, \text{params}, \text{invocations}) \) where \( G_1 \) and \( G_R \) are graphs typed over a metamodel \( M \), \( \text{NACs} \) is the set of negative application conditions, \( \text{contextNode} \in N_G \) is the context node of the rule (\( N_G \) is the set of nodes of \( G_R \)), \( \text{params} \in N_G \times \cdots \times N_G \) is the (possibly empty) list of parameter nodes, and \( \text{invocations} \) is the list of invocations of other DMM rules (which are pushed on the invocation stack after application of the invoking rule).

**Definition 2 (Rule matching).** Let \( G \) be a graph typed over a metamodel \( M \), let \( R \) be a DMM rule as defined in Definition 1 typed over the same metamodel. \( R \) matches \( G \) if the following conditions hold:

1. The invocation stack is either empty if \( R \) is a bigstep rule or has an according invocation on its top if \( R \) is a smallstep rule.
2. A morphism \( m \) from \( G_1 \) to \( G \) can be found such that the types of the matched nodes in \( G \) are of the same type or a subtype of the matching nodes in \( G_1 \).
3. \( m \) cannot be extended to \( m' \) such that \( m' \) is a morphism from any of the rule’s \( \text{NACs} \) to \( G \).

Having defined DMM rules and rule matching, we are now ready to discuss how all this is realized technically.

### 3.2. Translating DMM rules into GROOVE rules

To compute the transition system describing a model’s behavior, a DMM specification is transformed into a GROOVE graph grammar. GROOVE is a powerful tool set allowing to automatically apply graph transformation rules to graphs [7]. The overall workflow is depicted as Fig. 8.

![Fig. 8. Computing a transition system from a DMM specification and a UML model.](Image 41x186 to 54x197)
The general translation of DMM rules into GROOVE rules is overall straight-forward: DMM nodes and edges become GROOVE nodes and edges, the same holds for more advanced concepts such as negative application conditions and UQS.

The typing of nodes is more interesting: For this, GROOVE offers a special construct. The idea is to add information about a type hierarchy to a GROOVE grammar; GROOVE then makes sure that typing is respected when computing rule matches and rule applications.

Let us investigate the GROOVE rule resulting from the translation of DMM rule `initialNode.flow()#`, which is depicted as Fig. 9. Before we can do that, we need to familiarize ourselves with GROOVE’s concrete syntax. Like DMM, GROOVE merges the two sides of each graph transformation rules into one (annotated) graph. Elements to be created are depicted by fat solid green lines, elements to be deleted are shown using thin dashed blue lines. The application context (i.e., the elements contained on both the left and the right side of the rule) is depicted with thin solid black lines. Finally, elements belonging to NACs are depicted with fat dashed red lines. Within this paper, we have additionally annotated GROOVE rules as follows: Elements to be created are annotated with (+), elements to be deleted are annotated with (−), and elements belonging to NACs are annotated with (!).

With this knowledge, it becomes immediately clear how DMM rule `initialNode.flow()#` has been translated— with one exception: The `DMMInvocation` nodes on the right side of the rule are apparently related to the invocation mechanism. To understand that mechanism, we now want to investigate a GROOVE state graph, i.e., the one resulting from translating an actual model; the state graph is depicted as Fig. 10.

Let us now investigate the invocation mechanism. We have already seen that bigstep rules like `initialNode.explicitInvocation()#` and `action.explicitInvocation()#` can only match if the invocation stack is empty. This is exactly the situation we see in Fig. 10: The `DMMSystem` node represents the stack, and its only element is the special `DMMbottom` node. In other words: in the start state, the invocation stack is empty.

This empty stack is then used in the GROOVE rules resulting from the translation of bigstep rules: an empty invocation stack is added to the rules’ application context (this is what we see on the right side of Fig. 9). The effect is that these rules can only match if the invocation stack within the state graph is indeed empty, as desired.

Now, invoking a smallstep rule corresponds to manipulating the state graph’s invocation stack: A `DMMInvocation` node corresponding to the invoked smallstep rule is pushed onto the stack. This can be seen in the GROOVE rule depicted as Fig. 11: On the right side of the figure, we
can see that a DMM Invocation node is pushed onto the stack which corresponds to the invocation of a rule named execute. The target node of the invocation (which becomes the context node of the invoked rule) is marked by the DMM self edge, the passed parameter by the edge with labels DMM parameter and DMM next parameter. Note also how the UQS is translated into corresponding GROOVE structures (more information on GROOVE's notion of UQS can be found in [27]).

It remains to show how smallstep rules are created such that they can only match if they are invoked. Therefore, we first show the state graph resulting from the application of rule action.flow(), which is depicted as Fig. 12. The first difference to the start state is the existence of the Token object and the started reference: These are results from the application of rules initialNode.flow() and decisionMergeNode.flow() (the latter rule is not shown). The more interesting difference is on the rule's right side: The element on top of the invocation stack now corresponds to a smallstep rule (the stack is not empty anymore).

It is easy to see that no bigstep rule can match within this state. However, the GROOVE rule resulting from translating DMM rule action.execute(ActivityExecution), which is depicted as Fig. 13, shows the current state of the invocation stack in its application context. In other words: the rule can only match if an according rule is on top of the stack (as desired). The application of the rule then pops the top element from the stack (and performs whatever other changes are modeled with the rule. In the case of rule action.execute(ActivityExecution), this is only the invocation of rule action processTokens(ActivityExecution), resulting in an according manipulation of the invocation stack).

3.3. Computing transition systems

Using GROOVE, we can now compute a labeled transition system (LTS) which represents the complete behavior of the Activity under consideration. The LTS is computed by applying all matching rules to the start state. The
resulting transitions are labeled with the applied rule’s name. This procedure is repeated for all newly found states (note that in the case of our models, the resulting LTSs will always be of finite size).

Fig. 14 shows the LTS resulting from the example model of Fig. 3. It is very simple, but serves well to demonstrate the general approach. Note that the outgoing transitions of states s5 and s6 do not only carry the rule’s name, but also show in the context of which Action they have been applied (“A”, “B”). This is due to the usage of emphasized attributes: the Action node of rule action.execute(ActivityExecution) (Fig. 7) contains the emphasized attribute id. When computing the LTS, GROOVE will make sure that the value of the emphasized attribute at application time is added to the transition label. In the LTS, this allows to see which Action was actually executed.

The LTS can be the basis for analysis of the Activity, using standard techniques such as model checking [28]. For instance, in [29] we have used the described approach to verify whether UML activities are sound in the sense of [30].

4. Overriding rules in DMM

Assume that we have a language equipped with a DMM semantics, and we want to extend that language: Our goal is to introduce new language elements, and to specify their semantics as easily as possible. Consequently, we have to perform two tasks: First, we need to modify the language’s syntax by integrating the new elements into the already existing metamodel. Second, we need to specify how these elements behave.

As an example, we would like to introduce custom elements into the given Activity metamodel: an ExtendedToken shall be a subclass of class Token, and its purpose is to, e.g., carry additional information such as a certain object (for the sake of simplicity, this is not modeled in our example). To be able to produce ExtendedTokens, we introduce an ExtendedInitialNode class (naturally being a subclass of class InitialNode). Finally, we introduce an ExtendedAction which will process our ExtendedTokens in a certain, to be defined way; class ExtendedAction inherits from Action.

Fig. 13. GROOVE rule resulting from rule action.execute(ActivityExecution).

Fig. 14. LTS representing the semantics of the model from Fig. 3.
In other words: our extending metamodel contains three classes which are referring to and inheriting from classes from the syntax and the runtime metamodel seen in Figs. 2 and 4 (note that it would be conceptually cleaner to again separate into an extending syntax and runtime metamodel). The resulting metamodel is depicted as Fig. 15.

Let us now define the behavior of the new elements. The rules extendedInitialNode.flow()# and extendedAction.execute(ActivityExecution) are shown as Figs. 16 and 17. The task of the former is to create ExtendedTokens instead of just Tokens; despite that, the rule is similar to rule initialNode.flow()# from the original ruleset. Rule extendedAction.execute(ActivityExecution) is more complex than its counterpart, rule action.execute(ActivityExecution): the rule requires that at least one of the ExtendedAction’s incoming ActivityEdges is carrying an ExtendedToken owned by the passed ActivityExecution node. If this is the case, the rule will perform some pre- and postprocessing on the flowing tokens (we do not show the corresponding rules here).

Unfortunately, it is not that easy. As we have mentioned in the Introduction, DMM in its current state only allows to add rules to an existing ruleset. These added rules do not influence the application of the original rules, though: If one of the old rules as well as one of the newly added rules matches a state, both of them will be applied when computing a transition system, therefore leading to a branch.

This might be the desired behavior, but in some cases it is not. For instance, what does that mean for our new rules extendedInitialNode.flow()# and extendedAction.execute(ActivityExecution)? It is easy to see that rule initialNode.flow()# matches whenever rule extendedInitialNode.flow()# matches, and the same holds for rules action.execute(ActivityExecution) and extendedAction.execute(ActivityExecution). This is due to the fact that the left-hand graph of, e.g., initialNode.flow()# basically is a subgraph of the other rule’s left-hand graph. The only exception is the typing: in the new rule, some node’s type is not the same type but a subtype of the old rule’s node’s type. Referring to Definition 2, the above follows.

In a transition system, a state where rule extendedInitialNode.flow()# matches will therefore give rise to (at least) two new states. One is derived by applying rule initialNode.flow()#; in this state, a simple Token has been created. The other state is the result of an application of rule extendedInitialNode.flow()# and does contain a newly created ExtendedToken. In other words: if our model contains an ExtendedInitialNode, the resulting transition system will contain an undesired state where not an ExtendedToken but a Token has been created.

The same holds for rule extendedAction.execute(ActivityExecution): imagine a state where this rule matches. Since action.execute(ActivityExecution) also matches, our transition system will contain two paths: the desired one (including the invocation of the pre- and postprocessing) and the undesired one resulting from application of rule action.execute(ActivityExecution).

Note that removing e.g., rule initialNode.flow()# is no solution, since we then could not mix InitialNodes and ExtendedInitialNodes within one Activity any more. This is because rule extendedInitialNode.flow()# does not match within the context of a simple InitialNode.

The problem arises because up to now, DMM does not allow to refine behavior, in contrast to the addition of behavior as we did above. This is what we want to change: The problem can be solved by allowing rule extendedInitialNode.flow()# to override rule initialNode.flow()#, and to allow rule extendedAction.execute(ActivityExecution) to override rule action.execute(ActivityExecution). In the following, we discuss two different definitions of an overrides relation between DMM rules. Before we do that, we want to point out how rules should relate to each other to participate in such a relationship, and we want to discuss whether the overrides relation needs to be declared explicitly.

4.1. Prerequisites

First of all, the names of two rules participating in an overrides relation must be equal. Then, the context node of the overriding rule must be a subtype of the overridden rule’s context node. This is because we want to mimic overriding as it can be found in object-oriented languages (recall from Section 3 that the context node can be seen as owning the rule, similar to a class owning its methods). For the same reason, the parameter types of the two rules must be the same or subtypes of the overridden rule’s parameters, i.e., the first parameter of the overriding rule needs to have the same type or a subtype of the first parameter of the overridden rule and so on.

So far, our arguments have been based on similarity with well-known object-oriented concepts. There is one important difference between a method and a DMM rule, though: Correct invocation of a method only relies on syntactical constraints (and can therefore be checked by a compiler). For
a DMM rule, the situation is different: recall from Section 3 that the invocation of a rule might fail. This problem is not directly related to overriding at all. As we have seen before, a DMM specification which—when executed—gives rise to an intermediate state where a smallstep rule is invoked but cannot match is considered to be incorrect.

Now, recall that we are interested in overriding rules because we do not want them to match in cases a more specialized rule matches. In other words: overriding a rule only makes sense if the left-hand graphs of both rules are related such that if one rule matches, an overridden rule also matches. This means that the left-hand graph of the overriding rule contains the other rule's left-hand graph (modulo typing).

Note that putting this restriction on overriding rules has one big advantage for a language engineer refining an existing DMM specification which is correct in the sense discussed above: the language engineer can rely on the fact that whenever his overriding rule is invoked, the structure required by the overridden rule will be available; he only has to make sure that the elements possibly added by the overriding rule will be available at that time.

The following definition collects all requirements identified in this section:

**Definition 3.** Let \( R, R' \) be DMM rules as defined in Definition 1. \( R \) can only override \( R' \) if the following requirements are fulfilled:

1. \( R \) and \( R' \) have the same name.
2. \( R' \)'s context node has a type which is a subtype of the context node of \( R \).
3. \( R \) has the same number of parameters as \( R' \), and the parameter types are the same or are subtypes of the types of the parameters of \( R' \).
4. Let \( G \) be an arbitrary graph typed over the same metamodel as \( R \) and \( R' \). It must then be the case that \( R \) matches \( G \) implies \( R' \) matches \( G \).

### 4.2. Implicit and explicit overriding

In most programming languages, one does not have to explicitly declare if a method overrides a method of the superclass. This is possible because the signatures of all methods of a class must be pairwise distinct; therefore, a method declared in a subclass *implicitly* overrides the method of the “nearest” superclass, as long as it has the same signature. The same holds for UML classes and operations.

In DMM, the situation is different: as we have seen in Section 3.1, several rules having the same signature can exist. These rules will often have different left-hand graphs, but this does not even have to be the case. To achieve maximum flexibility, a rule therefore needs to explicitly declare the rules it overrides. This leads to the following modified definition of a DMM rule:

**Definition 4** (Overriding DMM rule). Let \( R \) be a DMM rule as defined in Definition 1. An overriding DMM rule is a tuple \( R_0 = (R, \text{overrides}) \) where \( \text{overrides} \) is the set of DMM rules overridden by this rule, such that all rules in \( \text{overrides} \) fulfill the requirements formulated in Definition 3.
Note also that we will later use the notation RO overrides RO if RO is contained in the set of overridden rules of RO (formally: RO overrides RO \( \iff R_O = (RO, overrides) \land R_O \in overrides \)).

We have seen how an overriding rule must relate to its overridden rule. From now on, we will assume that rule extendedInitialNode.flow()# overrides rule initialNode.-flow()#, and that extendedAction.execute(ActivityExecution) overrides rule action.execute(ActivityExecution) (the rules fulfill all requirements formulated above). Next, we want to discuss two semantics of rule overriding.

### 4.3. Complete overriding

The idea of the first alternative is that an overridden rule can only match if the node the rule’s context node is mapped to does not have an actual type for which an rule can only match if the node the rule’s context node is.

**Definition 5 (Rule matching (complete overriding)).** Let G be a typed graph, let R be an overriding DMM rule as defined in Definition 4. R matches G if the conditions listed in Definition 2 hold, and additionally:

1. **(4) No overriding DMM smallstep rule R’ exists such that R \( \in \) overridden \( R \) and R’ matches G.**

This notion of overriding is useful if some behavior shall never occur in the context of a subtype. Since this is the case for the ExtendedInitialNode we introduced above, we let rule extendedInitialNode.flow()# completely override rule initialNode.-flow()#. For example, our rule would mean that rule initialNode.flow()# cannot match such that its context node—its own type InitialNode—is mapped to a node of type ExtendedInitialNode, since another rule exists which overrides this rule and has ExtendedInitialNode as the type of its context node.

Having rule initialNode.flow()# not match anymore solves our problem of two rules being applied (leading to an unwanted branch in the transition system), but only partly: Assume that rule extendedAction.execute(ActivityExecution) completely overrides rule action.execute(ActivityExecution). Now, if only usual Tokens arrive at an ExtendedAction, rule extendedAction.execute(ActivityExecution) will match and be applied. Because of this, rule action.execute(ActivityExecution) will not match. However, if only Tokens arrive at ExtendedAction, rule extendedAction.execute(ActivityExecution) will not match. This “activates” rule action.execute(ActivityExecution); the behavior of the ExtendedAction falls back to that of the Action.

On the other hand, we have seen that the overridden rule action.execute(ActivityExecution) can never be applied in this situation, since there is an overriding rule having a context node typed as described above.

Here, the solution would be to add a second overriding rule, which matches in this very situation. This rule would basically be a copy of the overridden rule, with one difference: The context node would of course have type ExtendedAction. Note that this rule would fulfill the prerequisites for rule overriding formulated above as well.

Complete overriding is comparable to overriding as defined e.g., in Java: an overridden method will not be executed in the context of the subtype in which the overriding method is defined (unless explicitly called on the type’s super object within the overriding method). However, in the context of a type for which an overriding rule exists, the behavior of the supertype is completely “lost”, giving rise to the need for the second rule mentioned above. In the next section, we will discuss a notion of overriding which is more suited for this situation.

### 4.4. Soft overriding

The second approach to rule overriding differs from the first one at only one point: to prevent a rule from being applied, an overriding rule does not only have to exist, but must itself match. Before we provide the matching definition, let overridden be the transitive closure of the overrides relation of a DMM rule R, i.e., the set of rules which transitively override R.

Given that definition, we are now ready to provide the new matching definition:

**Definition 6 (Rule matching (soft overriding)).** Let G be a typed graph, let R be an overriding DMM rule as defined in Definition 4. R matches G if the conditions listed in Definition 2 hold, and additionally:

1. **(4) No DMM smallstep rule R’ exists such that R \( \in \) overridden \( R \) and R’ matches G.**

The main difference is that in this definition it does not suffice for rule R’ to exist to prevent rule R from matching G—additionally, R’ itself needs to match G. It is easy to see that this indeed solves our problem from the last section: In case that at least one ExtendedToken arrives at an ExtendedAction, rule extendedAction.execute(ActivityExecution) will match and be applied. Because of this, rule action.execute(ActivityExecution) will not match. However, if only Tokens arrive at ExtendedAction, rule extendedAction.execute(ActivityExecution) will not match. This “activates” rule action.execute(ActivityExecution); the behavior of the ExtendedAction falls back to that of the Action.

There is another, more subtle difference between Definitions 5 and 6: only smallstep rules can participate in a soft overriding relation. The reason for this lies in the way soft overriding is translated into corresponding GROOVE structures—we will explain this in the next section. However, this is actually no restriction: if the soft overriding mechanism is needed in the context of a bigstep rule, that rule can be changed such that its content is copied into a smallstep rule, which is then invoked by the bigstep rule. The created smallstep rule can now be softly overridden.

This more sophisticated definition of overriding implies some sort of dynamic binding: it must be decided at runtime which rule to take—the first matching rule in the inheritance hierarchy of the rule’s context node will be processed.
be applied. Note that in case a rule has overridden more than one other rule and does not match itself, it is possible that more than one of the overridden rules match and are applied, leading to the according number of new states.

4.5. Translation into GROOVE rules

The implementation of matching with complete overriding as introduced in Definition 5 is straightforward to implement: while generating the GROOVE rules to represent the DMM rules, the transformation tool needs to keep track of complete rule overriding relations. For every rule which is completely overridden, the transformer identifies the types of the context nodes of the overriding rules. For each of those collected types, it then adds a negative application condition to the context node of the overridden rule, preventing it from matching in a context where an overriding rule exists.

The new translation of the overridden rule initialNode.-flow()# is depicted as Fig. 18. The mentioned negative application condition can be seen at the bottom of the

![Fig. 18. GROOVE rule resulting from translating the overridden rule initialNode.flow()#.](image)

The GROOVE state after application of rule action.flow()# is shown in Fig. 19.

![Fig. 19. GROOVE state after application of rule action.flow()#.](image)

Fig. 20. GROOVE rule action.execute(ActivityExecution) with structure realizing soft rule overriding.
node typed InitialNode—as desired, it prevents the rule from being applied in the context of an ExtendedInitialNode. Note that the translation of rule extendedInitialNode.flow()# does not change.

The implementation of matching with soft overriding as defined in Definition 6 is more difficult, as the actual rule that is to be executed has to be identified dynamically at runtime. The basic idea is to equip rules participating in a soft overriding relation with additional structures which enforce that the rules can only match if they are “activated”. An additional helper rule makes sure that the rules are activated one after the other (according to their participation in the overrides relation) until the most specialized and matching rule is found and applied.

Let us investigate this in more detail: while transforming the DMM ruleset, the transformer builds up a rule hierarchy graph, where nodes correspond to rules, and edges correspond to overrides relations between the rules. This graph will be part of the start graph. Fig. 19 shows the state after application of rule action.flow(); the rule hierarchy graph resulting from our rules action.execute(ActivityExecution) and extendedAction.execute(ActivityExecution) can be seen on the right side of the figure (compared to the start state, it has not changed yet). Note that in the start graph, the node(s) corresponding to the most specialized rule(s) carry activated edges—this is where the search for an applicable rule will start in case an overriding smallstep rule is invoked.

Additionally, every rule participating in an overrides relation is enhanced in such a way that it can only match if the rule’s corresponding node in the rule hierarchy graph carries an activated edge. If a rule is applied, the activated edge is removed from that node and moved all the way down the rule hierarchy graph, therefore activating the most specialized rules again. For this, the needed information is collected during the transformation process, and the corresponding structures are added to the rules.

Fig. 20 shows the GROOVE rule resulting from translating rule action.execute(ActivityExecution). Its semantics is as follows: on the left side, the nodes resulting from the actual DMM rule we have shown as Fig. 7 can be seen. The structure to the right makes sure that the rule indeed matches as desired: first, the action.execute node carries an activated edge which is to be deleted; in other words, the rule can only match if the rule is activated within the rule hierarchy graph as described above. Note that we will explain the NAC below that node in the next section. Above to the right, the extendedAction.execute node gets a new activated edge, corresponding to the fact that the most specialized rules are activated again (in a more complex setting, there would probably be more rules to be activated). The part below ensures that all other activated edges are deleted as well.

Finally, we need to make sure that if none of the currently active rules match, the rules on the next level of the rule hierarchy graph need to be activated. To achieve this behavior, we make use of GROOVE’s rule priorities: every GROOVE rule has an associated priority, and a GROOVE rule can only match if no rule with higher priority matches at the same time. We use this by adding a helper rule with low priority to the GROOVE rule set, which removes all activated edges from the current rule hierarchy level and moves them to the next level.

The helper rule is depicted as Fig. 21. Its semantics is as follows: the left part of the rule makes sure that it only matches if the state contains at least one DMMOverrides node carrying an activated edge. If this is the case, it will delete all activated edges. The right part is responsible for moving the activated edge to the next level: every DMMOverrides node which has an overriding and activated DMMOverrides node gets an activated edge. The rule is generic in the sense that its nodes do not carry any rule labels. As a result, the rule can take care of all parts of the rule hierarchy graph, independent of the
It now becomes clear why bigstep rules can only participate in complete overriding relations: we have seen that the realization of soft overriding requires a kind of dynamic binding. If such a rule is overridden, it will often result in several applications of the helper rule from Fig. 21 until a matching smallstep rule is found (we will see an example in Section 4.6). However, only rules having the according signatures have to be taken into consideration.

In contrast, a bigstep rule is not invoked. All bigstep rules can potentially match as long as the invocation stack is empty. Consequently, if bigstep rules would participate in a soft overriding relation, we have to try them all. This is because if bigstep rule \( R \) does not match, it might be the case that \( R' \) (which would be overridden by \( R \) ) matches. To find out if this is the case, we have to activate the rule as described above. This is not the case for complete overriding, where the added negative application condition directly influences the matching of a rule.

Note that because of the described translation to GROOVE rules, any DMM rule may only participate in one type of overrides relation (complete or soft). However, both types of overriding can be used within one DMM ruleset as appropriate.

4.6. Complex rule overriding

The presented soft rule overriding scenario serves well to demonstrate the general idea, but is rather simple. In this section, we want to investigate a more complex example. It consists of three parts: in Fig. 22, a metamodel is shown, Fig. 23 shows a sequence of states of the rule hierarchy, and in Fig. 24, a labeled transition system can be seen.

The metamodel in Fig. 22 consists of several classes, some of which are in an inheritance relation. To keep the example compact, we have chosen abstract class names such as \( A \). The example contains multiple inheritance: class \( G \) inherits from classes \( E \) and \( F \).

Let us now investigate the sequence of states in Fig. 23. They depict the rule hierarchy: The graph at the top shows the initial state. It has been computed during the transformation from the model into the corresponding GROOVE state graph. Again to keep the example compact, we have omitted the DMM_Overriding labels from the nodes of the rule hierarchy graphs, and we use the label \( o \) instead of DMMoverrides and \( a \) instead of activated.

The initial state graph can be read as follows: there exist rules \( A . foo () \), \( B . foo () \) etc., where \( A . foo () \) refers to a smallstep rule of name foo having a context node of type \( A \). The \( o \) edges show that rule \( A . foo () \) is softly overridden by rules \( B . foo () \) and \( C . foo () \) etc., and finally, rules \( B . foo () \), \( D . foo () \), \( C . bar () \), and \( G . baz () \) are currently activated by the according \( a \) edges.

To demonstrate the soft overriding mechanism, we have created all these rules as dummy rules, i.e., despite the rule hierarchy graph, they do not change the state (we have even removed the invocation stack). Each rule matches if the corresponding node carries an activating edge. It then deletes all activating edges and creates new edges on the nodes corresponding to the most specialized rules (as the right part of rule action.execute(ActivityExecution) in Fig. 20 does).

Obviously, to implement soft rule overriding within our example, we need a helper rule to move the \( a \) edges. We do not show this rule, since it is very similar to the rule shown in Fig. 21 (only the labels have been changed as described above). The sequence of state graphs in
Fig. 24. Labeled transition system showing the matching of the rules.

Fig. 23 shows the effect of the helper rule: with each rule application, all nodes having an activated predecessor node are themselves activated by means of an a edge, and all existing a edges are deleted. Note that GROOVE draws self-edges of nodes as node labels (states 2–4).

With this knowledge, we are ready to discuss the transition system depicted as Fig. 24. Its purpose is to show the activated rules in each state of the rule hierarchy graph. For this, we have adjusted the rule priority of the helper rule to be equal to those of the other DMM rules (recall from Section 4 that the helper rule normally has lower priority than the "normal" DMM rules, making sure that the rules on the next level are activated only if no smallstep rule of the current level matches).

The first state is state s1 at the top of Fig. 24. Besides the s1 label, the state carries four more labels referring to rules. These labels are in fact self-transitions: we have seen that the application of one of our example rules deletes all a edges and creates a edges at the most specialized rules. Therefore, rules B.foo() etc. do not change the state at all. They return to the initial state (as all rules in this example do).

Application of the helper rule brings us to state s2. In this state, the node corresponding to rule A.foo() is activated. However, the node of rule C.foo() is also activated. Now we can explain the reason for the NAC at the bottom of the action.execute(ActivityExecution) rule in Fig. 20: it makes sure that the rule only matches if there is no other (transitively) overriding rule which is still activated. This can be modeled with GROOVE by means of a regular expression about edge labels: label DMM_overrides+ of rule action.execute(ActivityExecution) in Fig. 20 means that the NAC is matched if there exists a node which is connected through a positive number of edges whose sequence of labels conform to the regular expression. Therefore, the rule does not match if an activated node is connected through at least one DMM_overrides edge. This is the reason why in the transition system, rule A.foo() does not match in state s2. However, the other activated rules do match; application of the rules again brings us back to the initial state.

Finally, rule A.foo() matches as the only rule in state s3. The next application of the helper rule brings us to state s4, where no activating a edges are existing any more. Therefore, neither any of our dummy rules nor the helper rule match. This state corresponds to a situation where a rule has been invoked, but none of the potential rules have matched, since none of the preconditions of any of the rules were fulfilled. We have stated above that a DMM specification giving rise to such situations is considered to be incorrect. Therefore, a special rule with very low priority comes into play, annotating the according state(s) and helping the language engineer to recognize and correct his error.

4.7. Other applications of rule overriding

In the last sections, we have seen by means of simple examples how rule overriding has been integrated into the DMM framework. In this section, we want to point to some more sophisticated uses of rule overriding we have performed so far or plan to perform.

Firstly, we want to mention our DMM specifications of the semantics of the UML’s behavioral languages: we have implemented the semantics of Activities [31], State machines [32], and Interactions [33]. While creating the specifications, we have used rule overriding at two places:

- Due to the UML metamodel with its deep and complex inheritance hierarchy, in some cases it was very convenient to be able to override behavior of a supertype, for instance in the treatment of the ObjectNode and its specializations.
- The UML specification contains explicit semantic variation points, where certain aspects of the semantics of an element are either completely left open or a default semantics is suggested, but the possibility to use variations of that semantics is explicitly stated. We have considered this by isolating semantic variation points into single DMM rules; if the need arises to change an element’s semantics, this can easily be done by inserting a subtype of the corresponding type into the language’s runtime metamodel and then specifying the new semantics by means of a DMM rule which overrides the original semantics.

Secondly, we plan to provide a DMM specification of the UML Action packages [see [1, Chapter 11]]. These packages contain several kinds of Actions which allow a rather low-level description of algorithms: for instance, the Destroy-ObjectAction allows to destroy an incoming object (and, optionally, the objects owned by that object).

Given such a DMM specification, it would be relatively easy to come up with an extending specification which specializes some of these Actions. For instance, a GarbageCollectingDestroyObjectAction could perform
some additional clean up in the context of an object to be destroyed.

5. Conclusions
In this paper, we have extended the notion of inheritance within the Dynamic Meta Modeling framework. We have argued that DMM specifications are easy to understand due to their visual, metamodel based appearance, although they are completely formal. However, we have identified a lack of expressiveness: in the current state, DMM is not capable of refining behavior, and we have seen how this negatively impacts the reusability of DMM specifications.

Consequently, we have strengthened DMM by introducing a notion of rule overriding which makes it possible to prevent the application of a rule if another, more specialized rule exists. We have integrated this enhancement into the DMM formalization, and we have defined two variants of rule overriding which cover common use cases of behavior refinement. Finally, we have shown how to implement rule overriding by means of standard graph transformation tools (i.e., the GROOVE toolset).

We believe that with rule overriding we have given the visual language engineer a powerful tool for developing language specifications which are not only easily understandable and formal, but can additionally be reused as a base for specifications of variants of the original language.

References