Abstract

Dynamic Meta Modeling (DMM) is a visual semantics specification technique targeted at languages equipped with a metamodel. In DMM, the metamodel of a language is mapped into a runtime metamodel able to express runtime states of instances of that language. In addition, graph transformation rules are defined which describe how these runtime states change in time. Given an instance of the runtime metamodel and a set of rules typed over that metamodel, a transition system can be computed which represents the semantics of the model instance under investigation.

To be easily understandable by language engineers, DMM resembles a couple of well-known object-oriented concepts. Part of this is the fact that a DMM rule has many similarities to a method in an object-oriented language.

In this paper, we enhance DMM such that DMM rules can “override” other DMM rules, similar to a method being overridden in a subclass. We argue that this does not only have positive impact on reusability of DMM specifications, but also improves the intuitive understandability of DMM rules.

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, often software engineers have to work together with domain experts from different fields.

One way to communicate between those differently skilled groups of people is to use visual modeling languages as a base for discussion. This approach works best if the language 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) [7]).

One requirement for efficiently using DSLs is that the semantics of the languages used should be defined precisely. 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 can not be expressed with class diagram constructs—are added, e.g. using a language like the Object Constraint Language (OCL) [6].

For 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 texts accompanying the definition of the static semantics. Such an informal description of the language’s behavior almost always leaves room for 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 efficiently work with a DSL, a sophisticated tool support is needed, and second, the quality of complex models can not 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 not familiar with the underlying formalism.

Consequently, a semantics specification technique is needed which is not only formal, but is also easily understandable for advanced language users (i.e., users who are at least familiar with the language’s metamodel). This is where Dynamic Meta Modeling (DMM) [5] comes into play.

DMM specifications are easily understandable for a number of reasons: First, the concrete syntax of a DMM
rule is an (annotated) object diagram instantiating the runtime metamodel; this visual and familiar appearance has proven to be easy to comprehend [8]. Second, we will see in Sect. 3 that 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: Behavior (i.e., operational rules) can be added or removed from a DMM specification, but rules can not refine other rules (as a class might refine 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 that concept of rule overriding.

In the next section, we give a brief overview on related work and point out the differences of existing approaches to 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 decided to demonstrate rule overriding using a simplified version of that language. Section 4 then introduces and discusses our notion of rule overriding. The last section concludes and gives an outlook on future work.

2. Related Work

In [1], de Lara et.al. show how to integrate attributed graph transformations with node type inheritance, therefore allowing for the formulation of 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 refinement of rules (and is therefore comparable with the expressiveness of the current state of DMM).

In [11], Taentzer et.al. show how to formulate structural properties of type graphs with inheritance using graph constraints, and they provide a translation into standard graphs. In contrast to our work, they concentrate on structure, whereas our approach modifies the behavior of rules participating in an overrides relation.

The AGG toolset [10] 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 for 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 according 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.

In [3], Fischer et.al. introduce a new graph transformation based language called Story Diagrams: they use UML class diagrams for the specification of metamodels, UML activity diagrams for the (graphical) representation of control structures, and UML collaboration diagrams as notation for graph transformation rules (which are of course typed over the metamodels). Story Diagrams are translated to Java classes and methods allowing a seamless integration of object-oriented and graph rewrite specified system parts. In contrast to DMM, the specification of Story Diagrams does not take refinement into account (although the generated Java code might contain classes having an inheritance relation and method overriding).

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 as Fig. 1.

DMM is targeted at languages having an abstract syntax which is defined by means of a metamodel as suggested by the OMG. 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 ab-
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 metamodel. For example, the Activity’s runtime metamodel is depicted as Fig. 2. It has additional elements like ActivityExecution and Token, which allow to express states of execution of the Activity under consideration. Note that classes MarkerAction and Marker are part of the refined language and will be explained in Sect. 4.

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 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 in time. For this, the instances are mapped to typed graphs, 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 $G'$. DMM also supports Negative Application Conditions—structures which must not be present for a rule to match—and Universally Quantified Structures. The latter allows to match all nodes which fulfill the specified structural requirements, independent of their actual number within the state graph.

So far, DMM rules are not different from common 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. 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 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 other DMM rules. We will explain rule invocation in more 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 rules—to be bigstep or 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.

Further more, smallstep rules can have parameters. In this way, 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.

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 the types of target node and parameter nodes must be the same or subtypes of the according types of the invoked rule. Additionally, an invocation provides the name of the rule to be invoked.

Formally, 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.

Alltogether, 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 rules which
are easier to understand and can be reused in other contexts.

Before we illustrate the above with examples, 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 [5].

**Definition 1 (DMM Rule)** A DMM rule is a tuple \( R = (\text{name}, G_L, G_R, \text{NACs}, \text{contextnode}, \text{params}, \text{invocations}) \) where \( G_L \) and \( G_R \) are graphs typed over a metamodel \( M \), \( \text{NACs} \) is the set of negative application conditions, \( \text{contextnode} \) is the context node of the rule, \( \text{params} \) is the list of nodes from \( G_L \), 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 \( R = (\text{name}, G_L, G_R, \text{NACs}, \text{contextnode}, \text{params}, \text{invocations}) \) be a DMM rule typed over the same metamodel. \( R \) matches \( G \) if the following conditions hold:

1. The invocation stack is empty (if \( R \) is a bigstep rule) or has an according invocation on its top (if \( R \) is a smallstep rule).
2. A morphism from \( G_L \) 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 \( G_L \).
3. No morphism from any of the rule’s \( \text{NACs} \) to \( G \) can be found.

### 3.2. DMM Specification for UML Activities

We now want to illustrate the above concepts with our specification for a simplified version of UML Activities. We start by introducing the runtime metamodel we derived from the original metamodel contained in the UML specification [7].

Figure 2 shows an excerpt of the runtime metamodel. While developing that metamodel, we have followed the textual description of the Activity’s semantics provided as part of the UML specification. In a nutshell, the execution of an Activity is controlled by the class \( \text{ActivityExecution} \), which is a composition of the elements needed to describe the states of execution; one of these elements is the \( \text{Token} \) class. The rules will (mainly) control the flow of tokens through the Activity.

Note that due to space restrictions, Fig. 2 also contains elements of the refined language introduced in Sect. 4. The according classes are printed with a darker background.

Figure 3 shows an example bigstep rule implementing the semantics of the \( \text{Action} \). Note that due to restrictions of the tooling we developed, neither context node nor parameters of DMM rules are visualized; in Fig. 3, the rule’s context node is the node typed \( \text{Action} \). Note also 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 e.g. Fig. 6).

Therefore, the semantics of the rule is as follows: It matches if all incoming edges of the \( \text{Action} \) carry at least one \( \text{Token} \). If this is the case, it pushes three invocations onto the invocation stack. All invocations have the \( \text{Action} \) node as their target node, and in all cases, the \( \text{ActivityExecution} \) node is passed as the only parameter.

Again due to space restrictions, the invoked small-step rules \( \text{action.collectInputs}(\text{ActivityExecution}) \) as well as \( \text{action.createOutputs}(\text{ActivityExecution}) \) are not presented in this paper; they delete the tokens from the incoming edges and produce tokens on the outgoing edges of the \( \text{Action} \).

We do provide rule \( \text{action.execute}(\text{ActivityExecution}) \), although this rule is very simple, since it does not do anything. This already allows to execute Activities containing all kinds of \( \text{Actions} \) without having to implement the more specific \( \text{Action} \)'s behavior, which can still be refined if desired (as we do in Sect.4). Of course, the application of the rule has an implicit effect: The according invocation is removed from top of the invocation stack.
3.3. Computing Transition Systems

To compute the transition system describing a model’s behavior, a DMM specification is translated into a set of GROOVE rules. GROOVE is a powerful toolset allowing for the automatic application of graph transformation rules to graphs[9].

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: Here, we have to distinguish between rule graphs and state graphs. For the latter, every node needs to carry the information about all types it has (i.e., its most specialized type and all super-types of that type). These types are added as labels to the GROOVE nodes. However, a rule only needs to specify a concrete type for every involved node. The fact that every node of a state graph carries labels with all of its types ensures that rule nodes can be mapped to the according state nodes. More details can be found in [5].

As the reader might expect, invocations are treated by introducing the already mentioned invocation stack: Every GROOVE rule resulting from a bigstep rule has an empty invocation stack in its application context, therefore enforcing that the rule can only match if the stack is empty. Accordingly, a GROOVE rule resulting from a smallstep rule has an invocation stack in its application context which has an according invocation on its top.

Figure 5 shows the GROOVE rule resulting from translation of smallstep rule action.execute(ActivityExecution). The nodes having a label Invocation are part of the invocation stack, the Action and ActivityExecution nodes from the DMM rule can be seen on the right side of the rule. The rule removes its own invocation from the stack, therefore enabling the next DMM rule (which might be a bigstep rule if the Invocation node on top of the stack represents the bottom of the stack).

The computed transition system represents the complete behavior of the Activity under consideration. It can be the basis for analysis of the Activity, using standard techniques such as model checking [2].

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 a new language element, and to specify its semantics as easy as possible. Consequently, we have to perform two tasks: First, we need to modify the language’s syntax by integrating the new element into the already existing metamodel. Second, we need to specify how this element behaves.

As an example, we would like to introduce a custom element into the given Activity metamodel: A MarkingAction shall be a subclass of class Action, and its purpose is to be marked with a Marker as soon as the MarkingAction is executed the first time. This allows to execute an Activity and, after execution, to see which MarkingActions have been executed. The first step, the extension of the existing metamodel, is straightforward; the new elements are depicted with a dark background in Fig. 2.

Now for the behavior: the rule markingAction.execute(ActivityExecution) realizing it is depicted as Fig. 6. There are two differences to rule action.execute(ActivityExecution) introduced in the previous section as Fig. 4: The rule’s context node has type MarkingAction, and the rule creates a Marker element and associates it to the context node (as long as the node does not already have a Marker).

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 rule markingAction.execute(ActivityExecution)?
It is easy to see that rule `action.execute(ActivityExecution)` matches whenever rule `markingAction.execute(ActivityExecution)` matches. This is due to the fact that the left-hand graph of `action.execute(ActivityExecution)` basically is a subgraph of the other rule’s left-hand graph. The only exception is the context node: In the new rule, its type is not the same type but a subtype of the old rule’s context node’s type. Referring to Def. 2, the above follows.

In a transition system, we will therefore end up with two states for every execution of the `MarkingAction`. One is derived by applying rule `action.execute(ActivityExecution)`; in this state, no Marker has been created. The other state is the result of an application of rule `markingAction.execute(ActivityExecution)` and does contain a newly created `Marker`.

Note that removing rule `action.execute(ActivityExecution)` is no solution, since we could not mix Actions and MarkingActions within one Activity any more. This is because rule `markingAction.execute(ActivityExecution)` does not match within the context of a simple `Action`.

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 `markingAction.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 if 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 since we want to mimic overriding as it can be found in object-oriented languages. For the same reason, the parameter types of the two rules must be the same, i.e., the first parameter of the overriding rule needs to have the same type as the first parameter of the overridden rule and so on.

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 the context nodes’ types).

Note that putting this restriction on overriding rules has one big advantage for a language engineer refining an existing DMM specification: 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 possibly added elements 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 Def. 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 parameters have the same types.
4. Let $G$ be an arbitrary graph typed over the same meta-model as $R$ and $R'$. It must then be the case that $R \text{ matches } G \implies R' \text{ matches } G$.

### 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 Sect. 3, 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, we therefore decided that a rule 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 Def. 1. An overriding DMM rule is a tuple $R_O = (R, \text{ overrides})$ where `overrides` is the set of overridden DMM rules overridden by this rule, such that all rules in `overrides` fulfill the conditions formulated in Def. 3.

Note that we will later use the notation $R_{\text{overrides}}R'_O :\iff R_O = (R, \text{ overrides}) \land R'_O \in \text{ overrides}$.

Now that we have seen how an overriding rule must relate to its overridden rule, we will from now on assume
that rule `markingAction.execute(ActivityExecution)` overrides rule `action.execute(ActivityExecution)` (note that the two 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 overriding rule exists. Definition 2 is then modified as follows:

**Definition 5 (Rule matching (complete overriding))** Let $G$ be a typed graph, let $R = (\text{name}, G_L, G_R, \text{NACs}, \text{contextnode}, \text{params}, \text{invocations}, \text{overrides})$ be an overriding DMM rule. $R$ matches $G$ if the conditions listed in Def. 2 hold, and additionally:

4) Let $n$ be the node of $G$ to which contextnode is mapped. No rule $R' = (\text{name}', G'_L, G'_R, \text{NACs}', \text{contextnode}', \text{params}', \text{invocations}', \text{overrides}')$ exists such that $R' \in \text{overrides}'$ and the type of contextnode' is the same or a subtype of the type of contextnode.

For our example, this would mean that rule `action.execute(ActivityExecution)` can not match such that its context node—itself having type `Action`—is mapped to a node of type `MarkingAction`, since another rule exists which overrides this rule and has `MarkingAction` as the type of its context node.

This solves our problem of two rules being applied (leading to an unwanted branch in the transition system), but only partly: Consider the situation where a `MarkingAction` has already been executed for the first time. As a result, the another rule exists which overrides this rule and has mapped to a node of type `MarkingAction` according number of new states.

### 4.4. Soft Overriding

The second approach of rule overriding differs from the first one at only one point: To prevent a rule from matching, an overriding rule does not only have to exist, but must itself match. Before we provide the matching definition, let $\text{overridden}_R$ 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 $R = (\text{name}, G_L, G_R, \text{NACs}, \text{contextnode}, \text{params}, \text{invocations}, \text{overrides})$ be an overriding DMM rule. $R$ matches $G$ if the conditions listed in Def. 2 hold, and additionally:

4) No overriding DMM rule $R'$ exists such that $R' \in \text{overridden}_R$ and $R'$ matches $G$

The difference is that in this definition, it is not enough 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: The first execution of an `Action` will be performed by applying rule `markingAction.execute(ActivityExecution)`, therefore annotating the `MarkingAction` with a `Marker`. As soon as the `Action` is to be executed the next time, that rule does not match any more as discussed above. Now, rule `action.execute(ActivityExecution)` comes into play and takes care of executing the `Action` without creating a new `Marker`.

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 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.

---

1 Unless explicitly called on the type’s `super` object within the overriding method.
4.5. Translation into GROOVE Rules

The implementation of the matching with complete overriding as introduced in Def. 5 is straightforward to implement: While generating the GROOVE rules to represent the DMM rules, the transformation needs to keep track of the rule overriding. For every rule which is 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 implementation of matching as defined in Def. 6 is more difficult; due to space restrictions, we only provide the idea. The interested reader may refer to [4].

As mentioned before, we basically have to implement a kind of dynamic binding. This is done as follows:

- While transforming the DMM ruleset, the transformer builds up an “inheritance tree”, where nodes correspond to rules, and edges correspond to overrides relations. This tree will be part of the start graph.

- Every rule participating in an overrides relation is changed in such a way that it can only match if the rule’s corresponding node in the inheritance tree is activated by carrying a special activator node. If a rule is applied, the activator node is removed.

- Generic auxiliary rules, which are also created during the transformation process, take care of moving the activator node from one inheritance level to the next in case none of the overriding rules of the current level matches.

Note that because of the (very briefly) 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.

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 concrete syntax, 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 specialised 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 briefly 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


