Extending DMM Behavior Specifications for Visual Execution and Debugging

Nils Bandener, Christian Soltenborn, and Gregor Engels

University of Paderborn, Warburger Straße 100, 33098 Paderborn, Germany
{nilsb,christian,engels}@uni-paderborn.de

Abstract. Dynamic Meta Modeling (DMM) is a visual semantics specification technique targeted at behavioral languages equipped with a metamodel defining the language's abstract syntax. Given a model and a DMM specification, a transition system can be computed which represents the semantics of that model. It allows for the investigation of the model's behavior, e.g. for the sake of understanding the model's semantics or to verify that certain requirements are fulfilled. However, due to a number of reasons such as tooling and the size of the resulting transition systems, the manual inspection of the resulting transition system is cumbersome.
One solution would be a visualization of the model's behavior using animated concrete syntax. In this paper, we show how we have enhanced DMM such that visual execution and debugging can be added to a language in a simple manner.

1 Introduction

One challenge of today's software engineering is the fact that software systems become more and more complex, making it hard to produce systems which work correctly under all possible executions. As a result, the *Object Management Group* (OMG) has proposed the approach of *Model-Driven Architecture* (MDA). The main idea of MDA is to start with an abstract, platform-independent *model* of the system, and to then refine that model step by step, finally generating platform-specific, executable code.

In this process, *behavioral* models (e.g., UML Activities) play an increasingly important role; they allow to model the system's desired behavior in an abstract, visual way. This has a couple of advantages, one of the most important being that such visual models can be used as a base for communication with the system's stakeholders (in contrast to e.g. Java code).

However, to get the most usage out of behavioral models, their semantics has to be defined precisely and non-ambiguously; otherwise, different interpretations of a model's meaning may occur, leading to all kinds of severe problems. Unfortunately, the UML specification [15] does not fulfill that requirement: The semantics of the behavioral models is given as natural text, leaving room for different interpretations.

One solution would be to specify the semantics of these behavioral languages with a *formal* semantics, i.e., some kind of mathematical model of the language's behavior. A major advantage of such a specification is that it can be processed automatically, e.g. for verifying the specification for contradictory statements.

Dynamic Meta Modeling (DMM) [9,13] is a semantics specification technique which results in semantics specifications that are not only formal, but also claim to be easily understandable. The only prerequisite for using DMM is that the syntax of the language under consideration is defined by means of a metamodel.

In a nutshell, DMM works as follows: In a first step, the language engineer creates a so-called *runtime metamodel* by enhancing the syntax metamodel with concepts needed to express states of execution of a model. For instance, in the case of the UML, the specification states that "the semantics of UML Activities is based on token flow". As a result, the runtime metamodel contains a Token class.

The second step consists of creating *operational rules* which describe how instances of the runtime metamodel change through time. For instance, the DMM specification for UML Activities contains a rule which makes sure that an Action is executed as soon as Tokens are sitting on all incoming ActivityEdges of that Action.

Now, given a model (e.g., a concrete UML Activity) and an according DMM specification, a transition system can be computed, where states are instances of the runtime metamodel (i.e., states of execution of the model), and transitions are applications of the operational rules. The transition system represents the complete behavior of the model under consideration and can therefore be used for answering all kinds of questions about the model's behavior.

However, investigating such a transition system is a difficult and cumbersome task for a number of reasons. First of all, we have seen that the states of the transition system are instances of the runtime metamodel, which can be pretty difficult to comprehend. Additionally, due to the so-called *state explosion problem*, the transition systems tend to be pretty large.

One solution for (at least partly) solving this problem would be to show the execution of a model in the model's own *concrete syntax*. This has two major benefits:

– It is significantly easier to find interesting states of execution, e.g., situations where a particular Action is executed.
– Investigating the states of the transition system only is an option for advanced language users, i.e., people who are at least familiar with the language's metamodel. In contrast, visualizing the model execution in concrete syntax is much easier to comprehend.

In this paper, we show how we extended the DMM approach to allow for exactly that. We will show how the language engineer (i.e., the person who defines a modeling language) can make her language visualizable and debuggable by adding a couple of simple models containing all information necessary to visualize a model's execution, and how this information is used to extend existing visual editors at runtime for the sake of showing the model execution. As a result, the language engineer can make the language under consideration visualizable and debuggable without writing a single line of code.

Structure of paper In the next section, we will give a short introduction to DMM, and we will briefly introduce the components we used to implement model visualization in a generic way. Based on that, Sect. 3 will show what information the language engineer has to provide, and how this information is specified by means of certain *configuration models*. Finally, Sect. 3 will briefly explain how we integrated our approach into the existing tooling. Section 5 will discuss work related to our approach, and finally, Sect. 6 will conclude and point out some future work.

2 Dynamic Meta Modeling

This section introduces the foundations needed for the understanding of the main section 3. It gives a very brief introduction to Dynamic Meta Modeling (DMM).

As already mentioned in the introduction, DMM is a language dedicated to the specification of behavioral semantics of languages whose syntax is defined by means of a metamodel. The general idea is to enhance the syntax metamodel with information needed to express states of execution of a model; the enhanced metamodel is called *runtime metamodel*. The actual behavior is then specified by means of operational rules which are typed over the runtime metamodel.

Given such a runtime metamodel and a set of DMM rules, a transition system can be computed. This is done as follows: First, a *semantic mapping* is used to map an instance of the syntax metamodel into an instance of the runtime metamodel; that model will then serve as the initial state of the transition system to be computed. Now, all matching DMM rules are applied to the initial state, resulting in a number of new states. This process is repeated until no new states are discovered. The resulting transition system represents the complete behavior of a particular model. It can then e.g. be analyzed with model checking techniques (see [10]). An overview of the DMM approach is depicted as Fig. 1.

Let us demonstrate the above using the language of UML Activities. A careful investigation of the UML specification reveals that the semantics of Activities is based on tokens flowing through the Activity; as consequence, two runtime states of the same Activity differ in the location of the flowing tokens.

In fact, the semantics is significantly more difficult: Tokens do not actually flow through an Activity. Instead, they are only *offered* to ActivityEdges. Only if an Offer arrives at an Action (and that Action is being executed), the Token owning the Offer moves. Figure 2 shows an excerpt of the runtime metamodel for UML Activities; elements depicted in white are part of the syntax metamodel, gray elements belong to the runtime part of the metamodel.

Now for the specification of the semantics' dynamic part: As mentioned above, this is done by operational rules. A DMM rule basically is an annotated object diagram. It *matches* a state if the rule's object structure is contained in that state. If this is the case, the rule is *applied*, i.e., the modifications induced

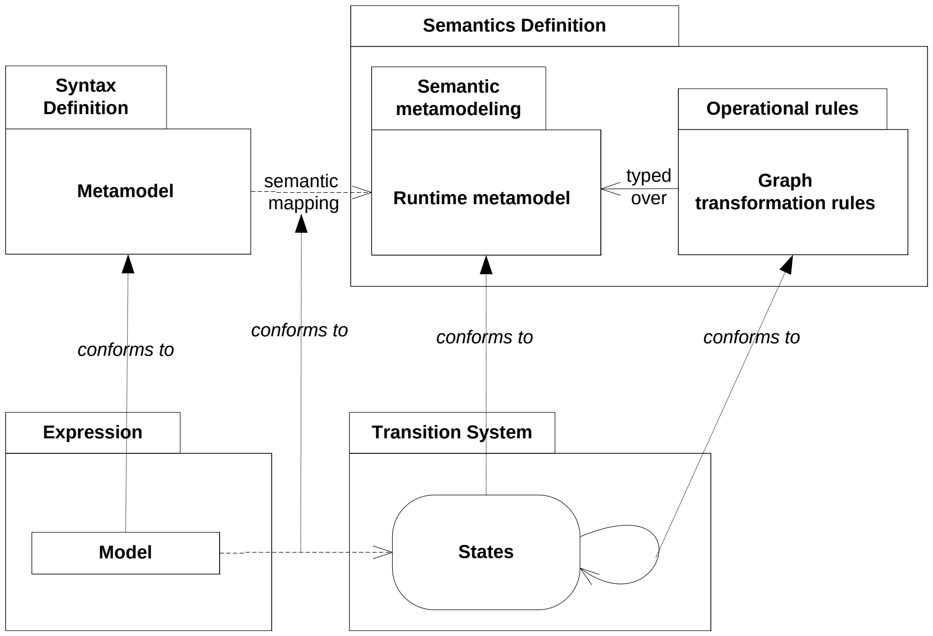

Fig. 1. Overview of the DMM approach.

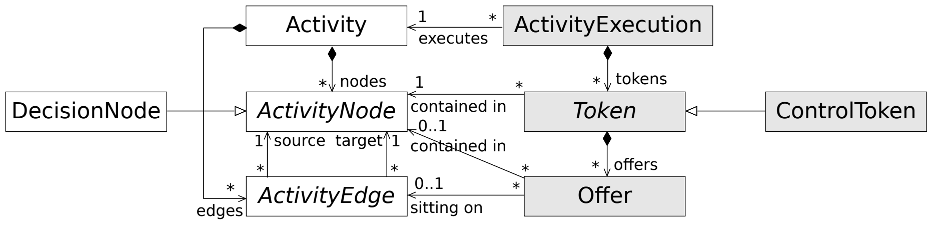

Fig. 2. Excerpt of the runtime metamodel.

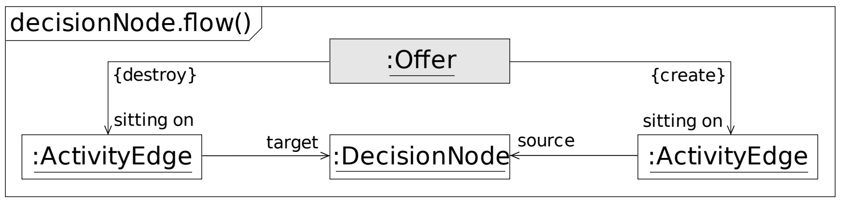

Fig. 3. DMM Rule decisionNode.flow().

by the annotations are performed on the found object structure, leading to a new state.

Figure 3 shows a simple DMM rule with name decisionNode.flow(). Its semantics is as follows: The rule matches if the incoming ActivityEdge of a DecisionNode carries an Offer. If this is the case, the Offer is moved to one of the outgoing edges. Note that usually, a DecisionNode has more than one outgoing edge. In this case, the rule's object structure is contained in the state more than once; every occurence consists of the incoming ActivityEdge, the DecisionNode, the Offer, and one of the outgoing ActivityEdges. As a result, we will end up with a new state for every outgoing edge, and the states will differ in the location of the Offer (which will sit on the according outgoing edge of that state). In other words: We end up with one new state for every possible way the Offer can go.

Note that DMM specifications usually do not aim at making models executable – if this would be the case, rule 3 would need to evaluate the guards of the DecisionNode's outgoing edges to determine the edge to which the Offer has to be routed. Instead, the transition system representing the model's behavior contains all possible executions of that model. This allows to analyze the behavior of Activities which are modeled rather informally (e.g., if the guards are strings such as "Claim valid"). See [10] for an example of such an analysis.

Technically, DMM rules are *typed graph transformation rules* (GTRs) [18]. DMM supports a couple of advanced features: For instance, *universally quantified structures* can be used to manipulate all occurrences of a node within one rule; *negative application conditions* allow to describe object structures which prevent a rule from matching; additionally, DMM allows for the usage and manipulation of attributes.

The main difference to common GTRs is the fact that DMM rules come in two flavors: *bigstep rules* and *smallstep rules*. Bigstep rules basically work as common GTRs: They are applied as soon as they match as described above. In contrast, smallstep rules have to be explicitly *invoked* by a bigstep rule or another smallstep rule;[1] as long as there are smallstep rules which have been invoked, but are yet to be processed, bigstep rules cannot match.

To actually compute a transition system from a model and a DMM semantics specification, the model as well as the set of DMM rules are translated into a graph grammar suitable for the graph transformation tool GROOVE [17]. Slightly simplified, this is done as follows: The model is translated into a GROOVE state graph which serves as the initial state of the transition system to be computed. Furthermore, each DMM rule is translated into an according GROOVE rule; features not directly supported by GROOVE are translated into structures within the GROOVE rules which make sure that the DMM rule's behavior is reflected by the GROOVE rule. For instance, to be able to handle invocation of rules, an actual *invocation stack* is added to the initial state, and the GROOVE

[1] Note that the invocation of a smallstep rule might fail in case it is invoked, but the object structure required by the invoked rule is not contained in the current state, resulting in the rule not matching; a DMM semantics specification potentially leading to such situations is considered to be broken.

rules have structures which manipulate that invocation stack and make sure that smallstep rules can only match if an according invocation is on top of the stack.

3 Visual Model Execution

With DMM, we can define the semantics of a modeling language, calculate execution states of model instances, and apply further analytical methods to it. Yet, the basic concept of DMM provides no means to visualize the execution and the therein occurring states of a model; a feature which is feasible for monitoring, better understanding, or debugging a model—especially if it is written in a visual language.

Thus, we have developed a tool for visually executing and debugging models with DMM-specified semantics, the *DMM Player* [1]. From a technical perspective, the tool is a set of Eclipse plug-ins, which is able to process models and DMM semantics specified with the *Eclipse Modeling Framework* (EMF) [5]. The visualization is realised using the *Graphical Modeling Framework* (GMF) [7], which is the standard way of providing visual editors for EMF-based models. As there are already numerous existing GMF editors for behavioral models which—however—do not support displaying runtime-information such as tokens or active states, the DMM Player also provides means to augment existing editors by such elements.

Leaving those pesky technical details behind, we will now focus on the underlying concepts which make the visualization of a model execution possible. Using UML Activities as a running example, we start with the fundamental question on how to visualize execution states and how the augmentation of existing visualizations can be specified in a model-driven way. Beneath the graphical dimensions, we will also have to consider the time axis when visualizing the execution. This will be covered in Sect. 3.2. Section 3.3 covers means of controlling the execution path when external choices are necessary. Section 3.4 introduces concepts that make debugging of models in the presented environment possible. Finally, in Sect. 3.5 we will demonstrate our approach on another language, i.e., UML Statemachines.

3.1 Visualizing Runtime Information

In order to visualize the behavior of a model, i.e., the development of its runtime state over time, it is obviously essential to be able to visualize the model's runtime state at all. However, this cannot be taken for granted. While certain visual languages have an inherent visual syntax for runtime information—such as Petri nets [16] visualizing the state using tokens on places—many visual languages only support the definition of the static structure—such as UML activity diagrams.

The specification for activity diagrams only informally describes the semantics using concepts such as tokens, which are comparable to the tokens used by Petri nets, and offers, which act as a kind of path finder for tokens. The specification does not provide any runtime information support in the formally specified metamodel and also does not give any guidelines on how to visualize runtime information. This is where DMM comes into play.

As we have seen in the previous section, the core concept of DMM translates this informal description into the runtime metamodel, which formally defines an abstract syntax for runtime states of models in the particular language. With the DMM Player, we have developed a set of concepts and techniques to define a concrete syntax for those runtime states. Similar to the enhancement by runtime information in the abstract syntax, the concept allows for building on the concrete syntax of the static part of models in order to create the concrete syntax for runtime states. The enhanced concrete syntax is defined using a completely declarative, model-based way.

In order to create such a visualization with the DMM Player, three ingredients are needed: An idea on how the concrete syntax should look like, the DMM runtime metamodel, and an existing extensible visualization implementation for the static structure of the particular language.

Our implementation of the DMM Player allows for extending GMF-based editors, as GMF offers all required extension mechanisms. The particular implementation is described in Sect. 4. In the following, we will focus on the concepts, which are—while being partially inspired by—independent from GMF. Essentially, this means that definitions for an enhanced concrete syntax may be also used in conjunction with other frameworks. This of course requires an implementation interpreting the DMM artifacts for these particular frameworks.

The first concern is how the runtime information should be visualized in concrete syntax. Beneath the obvious question on the shape or appearance of runtime information, it may also be necessary to ask what runtime objects should be included in the visualization at all. Certain runtime information may be only useful in certain contexts. In the example of UML activity diagrams, the visualization of tokens is certainly essential; we visualize tokens—aligned with the visualization in Petri nets—as filled black circles attached to activity nodes. An example for such a diagram can be seen in Fig. 4. For debugging of models and semantics, visualized offers may also be useful; offers are visualized as hollow circles. As multiple tokens and offers may be in action at once, an arrow visualizes which offers are owned by which tokens.

The diagram in Fig. 4 also shows boxes labeled with the letters EX. These boxes indicate that the particular node is currently executing. We will not go any further into the semantics of these boxes, though.

Having an idea on how the concrete syntax should look, we combine it with the formal structure of the runtime metamodel to create a so called *diagram augmentation model* which associates certain parts of the runtime metamodel with visual shapes. The word "augmentation" in the name of the model refers to the fact that it is used by the DMM Player to augment the third ingredient, the preexisting static diagram visualization, with runtime information.

Diagram augmentation models use the metamodel which is partially pictured in Fig. 5. The class DiagramAugmentationModel is the root element, i.e. each augmentation model contains exactly one instance of this class. The attribute diagramType is used to associate a particular diagram editor with the augmentation model - this diagram editor will then be used for displaying runtime states.

The actual elements to be visualized are determined by the classes AugmentationNode and AugmentationEdge. More precisely, as both classes are abstract, subclasses of these classes must be used in an instance of the meta model. The subclasses determine the implementation type of the visualization.

The way the elements are integrated into the existing diagram is determined by the references between AugmentationNode and AugmentationEdge on the one side and EReference and EClass on the other side. The latter two classes stem from the Ecore metamodel, which is the EMF implementation of the meta-metamodel standard MOF. They represent elements from the DMM runtime metamodel the visualization is supposed to be based on.

In the case of an AugmentationNode, the following references need to be set: The reference augmentationClass determines the class from the runtime metamodel which is visualized by this particular node; however, this is not sufficient, as the class needs to be somehow connected to elements that already exist in the visualization of the static structure. For instance, a token is linked to an activity node and should thus be visually attached to that node. This connection is realized by the reference named references; it must point to an EReference object which emanates from the referenced augmentationClass or one of its super classes. The EReference object in turn must point towards the model element the augmenting element should be visually attached to. Thus, this model element must stem from the static metamodel and must be visualized by the diagram visualization to be augmented. The reference containment is only relevant if the user shall be able to create new elements of the visualized type directly in the editor; those elements will be added into the containment reference specified here.

Figure 6 shows the part of the augmentation model for UML activities which specifies the appearance of control tokens, which are a subclass of tokens. The references link specifies that the reference named contained_in determines to which ActivityNode objects the new nodes should be attached to. The references link is part of the class Token. However, as the link augmentationClass specifies the class ControlToken as the class to be visualized, this particular ShapeAugmentationNode will not come into effect when other types of tokens occur in a runtime model. Thus, other augmentation nodes may be specified for other tokens.

3.2 Defining the Steps of Executions

Being able to visualize individual runtime states, creating animated visualizations of a model's behavior is straightforward. Sequentially applying the rules of the DMM semantics specification yields a sequence of runtime states which can be visualized with a brief pause in-between, thus creating an animation.

However, the sequence of states produced by DMM is not necessarily well suited for a visualization. In many cases, subsequent states only differ in parts

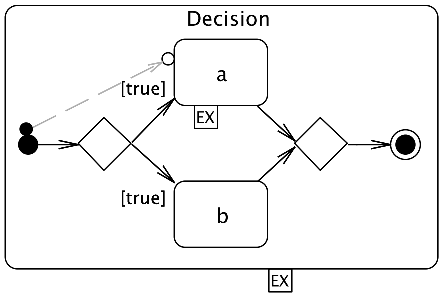

Fig. 4. An UML activity diagram with additional runtime elements

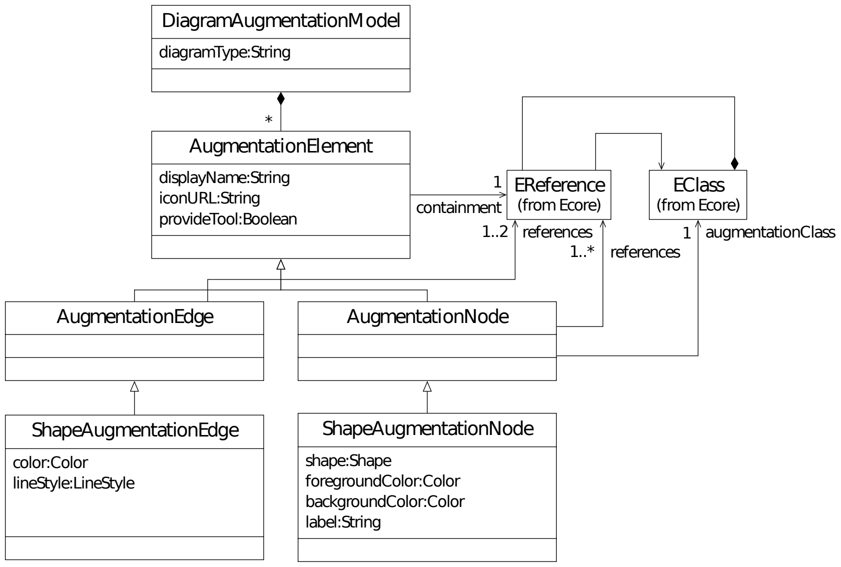

Fig. 5. Excerpt of the metamodel for diagram augmentation models

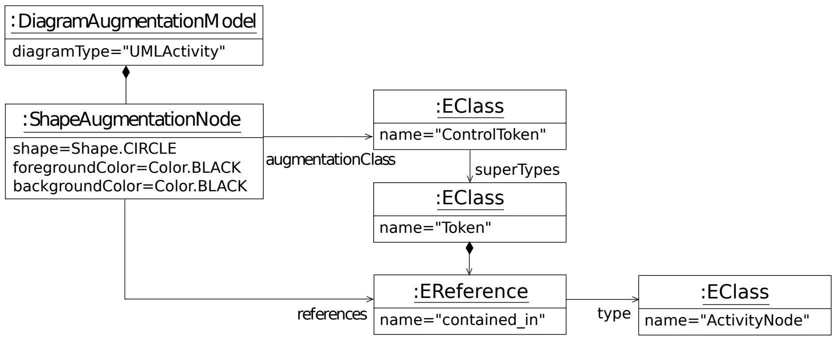

Fig. 6. Excerpt of the augmentation model for UML activities

that are not visualized. These parts are primarily responsible for internal information which is specific to the particular implementation of the semantics, but is not relevant for the behavior of the final model. Including these steps in the animation would cause strange, irregular pauses between visual steps.

Furthermore, DMM semantics may produce states with temporary inconsistencies. These states also result from implementation details of the particular DMM semantics specification. DMM rules may modify a model using several consecutive invocations of other rules; this is necessary if the semantics of a language element is too complex to be described within one rule. Each invoked rule produces a new state, which however might be wrong—when viewed from pure semantics perspective without implementation details. In the case of the particular implementation the state is of course still correct, as the subsequently invoked rules correct this inconsistency, thus making it a temporary inconsistency.

Figure 7 shows an example for a temporary inconsistency. In the first state, an offer has reached the final node. The activity diagram semantics now demands that the corresponding token should be moved to the node reached by the offer. The DMM implementation of the semantics however creates a new token on the target node before removing the original token from its location. This creates the exhibited temporary inconsistency with two tokens being visible at once.

The visualization of temporary inconsistencies might be interesting for the developer of the DMM semantics implementation; for a user only interested in viewing the behavior of a model, such states should not be visualized.

Thus, we need a way of selecting the states that should be displayed to the user. There is a number of different approaches to that problem which we will briefly discuss in the following.

If the visualization of temporary inconsistencies is desired and only the aforementioned problem of steps without visual changes needs to be addressed, a very simple solution is obvious: Using the diagram augmentation model, it is possible to determine what elements of the runtime model are visualized. The DMM Player can use this information to scan the consecutive states for visual changes; only if changes are detected, a visual step is assumed and thus promoted to the user interface.

If temporary inconsistencies are to be avoided in the visualization, other measures need to be applied. A simple and straight-forward approach would be to visualize only the state when the application of a Bigstep rule has been finished; application in this context means that the changes by the Bigstep rule and by its invocations have been performed. As temporary inconsistencies are typically raised by an invocation and again fixed by a consecutive invocation, temporary inconsistencies will be fixed when all invocations have been finished and thus the application of a bigstep rule has been finished.

Yet, the structuring concept of Bigstep and Smallstep rules has not been designed for visualization purposes; thus, it is also possible to find cases in which a state produced by a Smallstep rule should be visualized while the application of the invoking Bigstep rule has not been finished yet. Just restricting the visualization to states left by Bigstep rules is thus too restrictive.

An obvious solution would be the explicit specification of all rules that should trigger a visual step. This is, however, also the most laborious solution, as each rule of the semantics specification needs to be checked. In the case of the DMM-based UML activity semantics specification, this means the inspection of 217 rules.

Our solution for now is a combination of the approaches. Thus, in addition to the specification of individual rules triggering visual steps, the DMM Player also allows for the specification of all Bigstep rules. Furthermore, it is possible to specify whether the visualization should be triggered before or after the application of the particular rules.

For creating a suitable animated visualization with the DMM UML activity semantics, it is sufficient to trigger a visual step after the application of all Bigstep rules and after the application of only one further Smallstep rule, which takes the responsibility of moving a token to the new activity node that has accepted the preceding offer.

3.3 Controlling Execution Paths

A limitation of the behavior visualization using an animated sequence of states is its linearity. In some cases, the behavior of a model may not be unambiguously defined. For instance, this is the case in the activity diagram we have seen before in Fig. 4; the left decision node has two outgoing transitions. Both are always usable as indicated by the guard [true]. In a transition system, such a behavior is reflected by a fork of transitions leading from one state to several distinct states. In an animation, it is necessary to choose one path of the fork. At first sight, it is evident that such a choice should be offered to the user.

The DMM Player can offer this choice to the user by pausing the execution and visualizing the possible choices; after the user has made a choice, execution continues.

However, there are cases in which it is not feasible for the user to choose the path to be used for every fork in the transition system. This is primarily the case for forks caused by concurrency in the executed model. Even though a linear execution does not directly suffer from state space explosion, concurrency might require a decision to be made before most steps of a model execution.

As the semantics of concurrency can be interpreted as an undefined execution order, it is reasonable to let the system make the decision about the execution order automatically. Forks in the transition system which are caused by model constructs with other semantics—such as decision nodes—should however support execution control by user interaction.

The problem is now to distinguish transition system forks that should require user interaction from others. More precisely, as a single fork can both contain transitions caused by decision nodes and by concurrency, it is also necessary to identify the portions of a fork that are supposed to form the choices given to the user.

A basic measure for identifying the model construct that caused a fork or a part of it is considering the transformation rules that are used for the transitions forming the fork. In the case of the DMM semantics for UML activities, the transitions which cause the forks at decision nodes are produced by the Bigstep rule decisionNode.flow() (see also Fig. 3). Forks which consist of transitions caused by other rules can be regarded as forks caused by concurrency.

Just considering the rules causing the transitions is not sufficient, though. Concurrency might lead to forks which consist of decisionNode.flow() transitions belonging to different decision nodes in the model. If each of those decision node has only one active outgoing transition, there is no choice to be made by the user but just choices purely caused by concurrency.

Thus, it is necessary to group the transitions at a fork by the model element they are related to. This model element can be identified by using the rule match associated to the particular transition (see Sect. 2 for a brief explanation of rule matching and application). The match is a morphism between the nodes of the particular DMM rule and elements from the model. Generally, one node from a rule that is supposed to trigger a user choice can be used to identify the related model element. In the case of the DMM activity semantics, this is the :DecisionNode element itself.

With these components, we can build an algorithm for identifying the instances of transition system forks in which the user should be asked for a choice: Group the transitions that are possible from the current state by the rule and by the elements bound to the grouping node defined for the particular rule. If there is no grouping node or the rule is not supposed to cause user choices, the particular transition forms a group of size one. Now one of these groups is arbitrarily chosen by the software; this reflects possible concurrency between the single groups. If the chosen group contains more than one transition, the user is requested to make a choice. Otherwise the single transition in the arbitrarily chosen group will be used to gather the next state.

This algorithm enables us to ensure that the user is only required to make choices regarding single instances of certain model constructs, such as decision nodes. A remaining problem is how to give the user an overview over the possible choices. We can again utilize nodes from the rules that are supposed to trigger user choices. Those rules contain as a matter of principle always a node which represents the different targets which can be reached while the aforementioned grouping node stays constantly bound to the same model element. If the model element represented by the target node is part of the diagram, this diagram element can be used for identifying the different choices.

In the case of UML activities, this is the target ActivityEdge node in the rule decisionNode.flow(). The DMM Player can now use these model elements to visualize the possible choices using generic marker signs as is depicted in Fig. 8. The user may easily select one of the choices using the context menu of one of these model elements.

3.4 Debugging Concepts

As we have seen up to now, the main objective of the DMM Player is the visualization of a model's behavior by means of animated concrete syntax. This

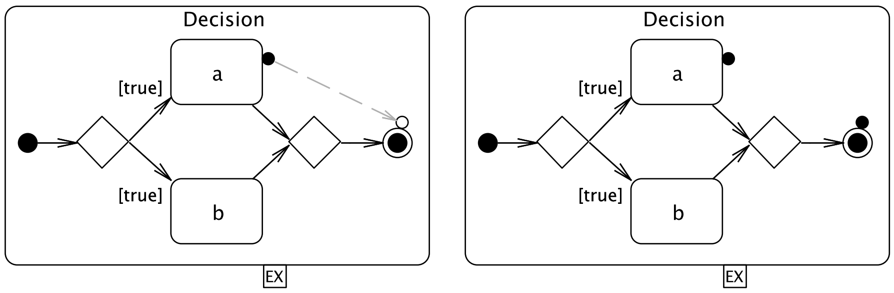

Fig. 7. A sequence of states exhibiting a temporary inconsistency

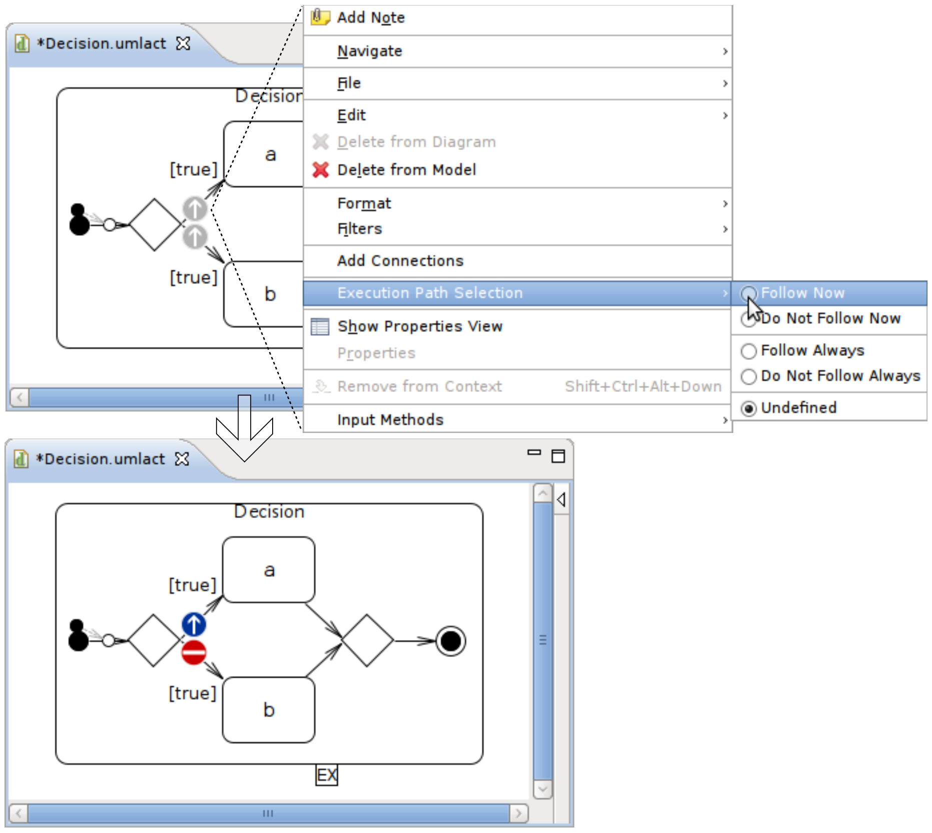

Fig. 8. UI for choosing execution paths

is very useful when trying to understand the details of a model's behavior, for instance if the model contains flaws. One feature which would obviously be of great use in such scenarios is the possibility to stop the execution of a model as soon as certain states of execution are reached. In other words: The transformation of concepts known from debuggers for classical programming languages to the DMM world would stand to reason.

The main concept of classical debuggers is the breakpoint; in source code, this is a line marked in such a way that the program execution is suspended just before the statements on that line are executed. When the execution is suspended, the values of program variables can be inspected by the user. Transferred to DMM, rules are the main units of execution; thus, the DMM Player supports setting breakpoints on DMM rules. Before (or, configurable, after) a rule is applied, the DMM Player suspends the execution. Using the property editors supplied by GMF, the user may now inspect the state of the model.

A variant of breakpoints are watchpoints. These cause the execution to be suspended when the condition defined by the watchpoint becomes true for variables in the executed program. *Property rules* can be seen as an analogical concept in the graph transformation and DMM world. Property rules do not modify the state of a model, and so they do not change a language's semantics. Their only purpose is to recognize certain states by matching them. Using a rule breakpoint, it is possible to suspend execution as soon as a property rule matches.

3.5 Example: UML Statemachines

To further demonstrate the usefulness of our approach, within this section we provide a second language for which we have applied our approach. Despite its visual similarity to UML Activities, we decided to use UML Statemachines. We would have preferred to use UML Interactions; unfortunately, we had issues with the GMF editor for Interactions as provided by the Eclipse UML2 tools [8], which we used in the preliminary version 0.9.

Let us briefly discuss the language of UML Statemachines. Syntactically, a Statemachine mainly consists of states and transitions between those states. At every point in time, a Statemachine has at least one active state. There are different kinds of states, the most important ones being the *Simple state* and the *Complex state* (the latter will usually contain one or more states). The semantics of transitions depends on their context: For instance, an unlabeled transition from a complex state's border to another state models that the complex state can be left while any of its inner state(s) is active. More advanced concepts like *history nodes* allow to model situations where, depending on different past executions, the Statemachine will activate different states.

A sample Statemachine is depicted as Fig. 9 (note that this figure already contains runtime information). The first active state will be state *A1*. From this state, either state *A2* or *A3* will be activated. In case of state *A3*, the *Complex State 1* will be entered. The state marked *H** is a so-called *deep history state*; it makes sure that in case state *Complex State 1* is activated again, all states which were active when that state was left are reactivated.

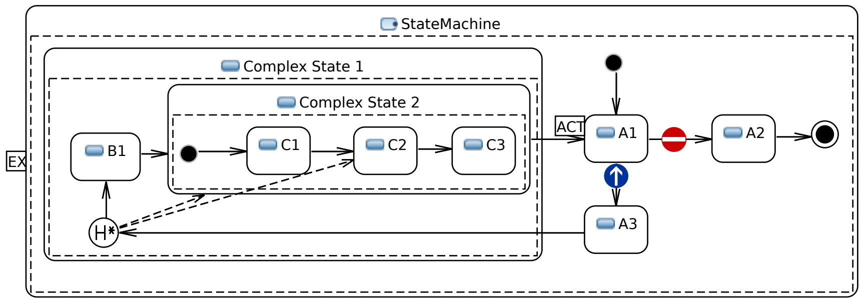

Fig. 9. Example state of Statemachine execution

We now want to briefly investigate the DMM semantics specification of UML Statemachines. As we have seen before, states of execution[2] of a Statemachine are determined by the active states. As a consequence, the runtime metamodel of Statemachines contains the concept of a Marker which references the currently active states (and will be moved by according DMM operational rules). To remember the last active states in case a complex state is left that contains a history state, the runtime metamodel introduces the HistoryMarker.

Next, we want to discuss how the execution of a Statemachine is visualized. In Fig. 9, we have already seen a Statemachine augmented with runtime information. Active states can be recognized by an attached box with the label *ACT*. These boxes represent the Marker instances from the runtime metamodel.

Further runtime information can be seen around the deep history state H*. The dashed arrows pointing away from that state signify the states that will be activated as soon as the complex state containing the history state is entered again. Thus, these arrows represent HistoryMarker instances. The part of the augmentation model that realizes the arrows representing history markers can be seen in Fig. 10. The ShapeAugmentationEdge instance specifies the class to be additionally visualized, i.e., the HistoryMarker and its references which determine the end points of the visualized edge.

We are now ready to explain the runtime state of the Statemachine which can be seen in Fig. 9. The currently active state is *A1*. Since from that state, either state *A2* or *A3* can be reached, the DMM player has already asked for a user decision – as the icons show, the user has decided to follow the transition leading to state *A3*.

Moreover, the visualization reveals that *Complex State 1* had already been active in the past. This is because there do exist HistoryMarker edges. The edges point to the states which had been active within state *Complex State 1* before it was left through the transition between *Complex State 2* and *A1* (i.e., *Complex*

[2] Note that *state* is overloaded here; as before, *state of execution* refers to the complete model.

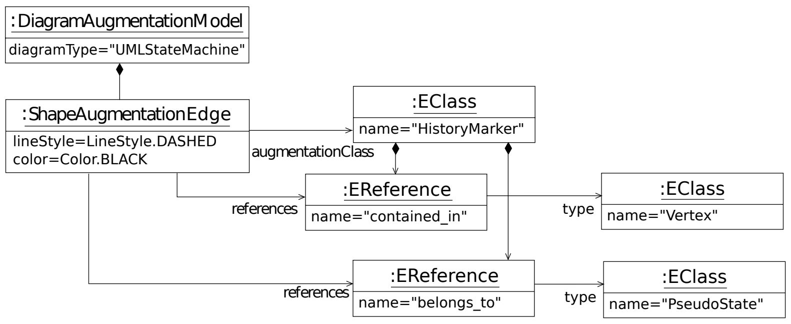

Fig. 10. Excerpt of the augmentation model for UML Statemachines

State 2 and, within that state, *C2*). Therefore, after two further execution steps, these states will be set active again.

4 Implementation

This section will give insights into our implementation of the concepts described in the last section. However, our explanations are rather high-level – the reader interested in more technical details is pointed to [1]. A high-level view of the DMM Player's architecture is provided as Fig. 11.

As mentioned above, the DMM Player builds upon Eclipse technologies; still, the concepts of DMM are completely technology-independent. The implementation can be divided into two mostly independent parts: The diagram augmentation and the model execution. Both are connected by the EMF [5] model just using its standard interfaces; the model execution process changes the model. The diagram augmentation part listens for such changes and updates the visualization accordingly.

The model execution part utilizes EProvide [19], a generic framework for executing behavioral models inside of Eclipse. EProvide decouples the actual execution semantics and the method to define them using two layers:

On the first layer, EProvide allows to configure the *semantics description language*, which provides the base for the actual definition. The DMM Player registers DMM as such a language. The second layer defines the actual *execution semantics* for a language using one of the languages from the first layer. Thus, a DMM semantics definition—such as the UML activities definition—is defined at this level.

EProvide essentially acts as an adaptor of the semantics description languages to the Eclipse UI on one side and EMF-based models which shall be executed on the other side. The DMM Player code receives commands from EProvide along with the model to be executed and the semantics specification to be used. The most important command is the step, i.e., the command to execute the next atomic step in the given model. The DMM implementation realizes that step

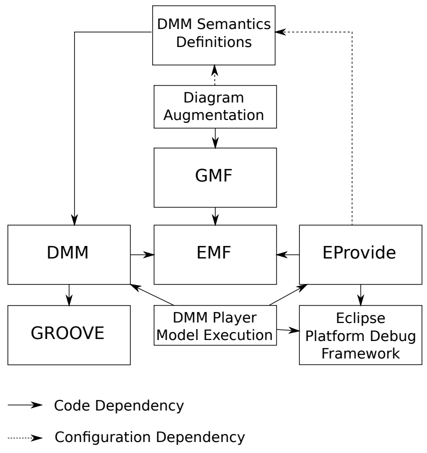
DMM Semantics Definitions
Diagram Augmentation
GMF
DMM
EMF
EProvide
GROOVE
DMM Player Model Execution
Eclipse Platform Debug Framework
Code Dependency
Configuration Dependency

Fig. 11. Architecture of DMM tooling

by letting the backing graph transformation tool GROOVE [17] perform the application of the according rule, and by translating the manipulations of the GROOVE rule back to the EMF model which is visualized.

The advanced features, such as the definition of visual steps—which actually combines multiple steps into single ones—, the user control of execution paths, and the debugging functionality, are realized directly by the DMM Player. The EProvide module MODEF [3] also offers debugging functionality which, however, could not be directly utilized, as it makes a quite strong assumption. It is assumed that the model's state can be deduced from one single model element. Since this is not the case with DMM, where a state is a complete model, we needed to bypass this module.

The implementation of the DMM debugging facilities makes use of the Eclipse Debugging Framework. At every point in time, the DMM Player keeps track of the GROOVE rule applied in the last step to derive the current state, as well as the rules matching that new state. If a breakpoint or watchpoint is reached, the execution is suspended as desired.

The diagram augmentation part of the DMM Player uses interfaces of GMF [7] for extending existing diagram editors. GMF offers quite extensive and flexible means for customizing editors using extension points and factory and decorator patterns.

GMF uses a three-layer architecture to realize diagram editors: Based on the abstract syntax model on the lowest level, a view model is calculated for the mid layer. The view model is simply a model representation of the graph to be visualized, i.e., it models nodes that are connected by edges. On the third layer, the actual UI visualization components are created for the elements from the second layer. Thus, specific elements get a specific look.

The DMM Player hooks into the mapping processes between the layers; between the abstract syntax model and the view model, it takes care that the elements defined in the augmentation model are included in the view model. Between view model and the actual UI, it chooses the correct components and thus the correct appearance for the augmenting elements.

Thus, the DMM Player provides a completely declarative, model driven way of augmenting diagram editors; there is no need to writing new or altering existing source code. Figure 8 shows screenshots of the activity diagram editor that comes with the Eclipse UML2 Tools which has been augmented by runtime elements using the DMM Player.

The DMM Player is designed to be generically usable with any DMM semantics specification. Thus, it offers extension points and configuration models that just need to be adapted in order to use a semantics specification with the DMM Player.

5 Related Work

The scientific work related to ours can mainly be grouped into two categories: Visualization of program execution and animation of visual languages. For the former, we only want to mention eDOBS [12], which is part of the Fujaba tool suite. eDOBS can be used to visualize a Java program's heap as an object diagram, allowing for an easy understanding of program states without having to learn Java syntax; as a result, one of the main usages of eDOBS is in education. In contrast to that, the concrete syntax of such eDOBS visualizations is fixed (i.e., UML object diagrams), whereas in our approach, the modeler has to come up with his own implementation of the concrete syntax, but is much more flexible in formulating it.

In the area of graph grammars and their applications, there are a number of approaches related to ours: For instance, in [14, 2, 11], the authors use GTRs to specify the abstract syntax of the language under consideration and operations allowed on language instances. The main difference to our approach is that in [14, 2], the actual semantics of the language for which an editor/simulator is to be modeled is not as clearly separated from the specification of the animation as in DMM, where the concrete syntax just reflects what are in fact model changes caused solely by (semantical) DMM rules. In the Tiger approach [11], a GEF [6] editor is generated from the GTRs such that it only allows for edit operations equivalent to the ones defined as GTRs; however, Tiger does not allow for animated concrete syntax.

Another related work is [4]; the DEViL toolset allows to use textual DSLs to specify abstract and concrete syntax of a visual editor as well as the language's semantics. From that, a visual editor can be generated which allows to create, edit, and simulate a model. The simulation uses smooth animations based on *linear graphical interpolation* as default; only the animation of elements which shall behave differently needs to be specified by the language engineer.

There is one major difference from all approaches mentioned to ours: As we have seen in Sect. 3.1, DMM allows for the easy reuse of existing (GMF based) editors. As a result, the language engineer only has to create the concrete syntax for the runtime elements not contained in the language's syntax metamodel, in contrast to the above approaches, where an editor always has to be created from scratch; reusing and extending an existing editor at runtime is not possible.

6 Conclusions

Visually executing a behavioral model as animated concrete syntax is an intuitive, natural way to understand the model's behavior. In this paper, we have shown how we have extended our DMM approach to allow for exactly that.

For this, we have first given a brief overview of DMM and the involved technologies in Sect. 1. Based on that, and using the running example of UML Activities, we have explained in Sect. 3 how the information needed to visually execute a model is added to DMM specification, and we have shown how this information is used to reuse existing GMF editors for the animation task by adding the according functionality to the editors at runtime, transparently to the user. Finally, we have discussed work related to ours in Sect. 5.

We believe that with the concepts and techniques described in this paper, we have achieved the next step towards a comprehensive toolbox for engineering behavioral visual languages. In a next step, we will integrate the described techniques more tightly into our DMM workflow. An obvious such integration could work as follows: As mentioned in Sect. 2, models equipped with a DMM semantics specification can be analyzed using model checking techniques. Now, if the result of the model checker is a counterexample (i.e., if a property does *not* hold for the model under consideration), the DMM Player can be used to visualize that counterexample, visually showing under which circumstances the property is violated.

Another area of our research is motivated by the fact that different people working with DMM might want to see different amounts of detail while simulating a model. For instance, the language engineer probably wants to see temporary inconsistencies while developing the semantics of a language, whereas these states should be hidden from end users (as we have argued in Sect. 3.2). Moreover, there might even be people which are only interested in an even higher view of a model's behavior; for instance, they might not care about the location of the offers. To suite the needs of these different kinds of users, we plan to extend our approach such that the augmentation and rulestep models can be *refined*. This would allow to start with a specification of the visualization which reveals all execution details, and then to refine that specification step by step, each refinement fulfilling the information needs of a different kind of language users.

References

1. Bandener, N.: Visual Interpreter and Debugger for Dynamic Models Based on the Eclipse Platform. Diploma thesis, University of Paderborn (2009)

2. Bardohl, R., Ermel, C., Weinhold, I.: GenGED – A Visual Definition Tool for Visual Modeling Environments. In: Pfaltz, J.L., Nagl, M., Böhlen, B. (eds.) Proceedings of AGTIVE '03. LNCS, vol. 3062, pp. 413–419. Springer, Berlin/Heidelberg (2003)
3. Blunk, A., Fischer, J., Sadilek, D.A.: Modelling a Debugger for an Imperative Voice Control Language. In: Reed, R., Bilgic, A., Gotzhein, R. (eds.) Proceedings of SDL 2009. LNCS, vol. 5719, pp. 149–164. Springer, Berlin/Heidelberg (2009)
4. Cramer, B., Kastens, U.: Animation Automatically Generated from Simulation Specifications. In: Proceedings of VL/HCC '09. IEEE Computer Society (2009)
5. Eclipse Foundation: Eclipse Modeling Framework. `http://www.eclipse.org/modeling/emf/`, online, accessed 9–1–2010
6. Eclipse Foundation: Graphical Editing Framework. `http://www.eclipse.org/gef/`, online, accessed 9–15–2010
7. Eclipse Foundation: Graphical Modeling Framework. `http://www.eclipse.org/modeling/gmf/`, online, accessed 5–5–2009
8. Eclipse Foundation: UML2 Tools. `http://www.eclipse.org/modeling/mdt/?project=uml2tools`, online, accessed 9–15–2010
9. Engels, G., Hausmann, J.H., Heckel, R., Sauer, S.: Dynamic Meta-Modeling: A Graphical Approach to the Operational Semantics of Behavioral Diagrams in UML. In: Evans, A., Kent, S., Selic, B. (eds.) Proceedings of UML 2000. LNCS, vol. 1939, pp. 323–337. Springer, Berlin/Heidelberg (2000)
10. Engels, G., Soltenborn, C., Wehrheim, H.: Analysis of UML Activities using Dynamic Meta Modeling. In: Bosangue, M.M., Johnsen, E.B. (eds.) Proceedings of FMOODS 2007. LNCS, vol. 4468, pp. 76–90. Springer, Berlin/Heidelberg (2007)
11. Ermel, C., Ehrig, K., Taentzer, G., Weiss, E.: Object Oriented and Rule-based Design of Visual Languages using Tiger. In: Proceedings of GraBaTs '06. ECEASST, vol. 1. European Association of Software Science and Technology (2006)
12. Geiger, L., Zündorf, A.: eDOBS – Graphical Debugging for Eclipse. In: Proceedings of GraBaTs '06. ECEASST, vol. 1. European Association of Software Science and Technology (2006)
13. Hausmann, J.H.: Dynamic Meta Modeling. Ph.D. thesis, University of Paderborn (2005)
14. Minas, M., Viehstaedt, G.: DiaGen: A Generator for Diagram Editors Providing Direct Manipulation and Execution of Diagrams. In: Proceedings of VL '95. IEEE Computer Society (1995)
15. Object Management Group: UML Superstructure, Version 2.3. `http://www.omg.org/spec/UML/2.3/`, online, accessed 9–15–2010
16. Petri, C.A.: Kommunikation mit Automaten. Ph.D. thesis, University of Bonn (1962)
17. Rensink, A.: The GROOVE Simulator: A Tool for State Space Generation. In: Pfaltz, J.L., Nagl, M., Böhlen, B. (eds.) AGTIVE 2003 – Revised Selected and Invited Papers. LNCS, vol. 3062, pp. 479–485. Springer, Berlin/Heidelberg (2004)
18. Rozenberg, G. (ed.): Handbook of Graph Grammars and Computing by Graph Transformation, Vol. 1: Foundations. World Scientific Publishing Co., Inc., River Edge, NJ, USA (1997)
19. Sadilek, D.A., Wachsmuth, G.: Prototyping Visual Interpreters and Debuggers for Domain-Specific Modelling Languages. In: Schieferdecker, I., Hartman, A. (eds.) Proceedings of ECMDA '08. LNCS, vol. 5095, pp. 63–78. Springer, Berlin/Heidelberg (2008)