Visual Interpreter and Debugger for Dynamic Models Based on the Eclipse Platform

Diploma Thesis

Submitted by Nils Bandener

Supervisor: Prof. Dr. Gregor Engels
Assessor: Prof. Dr. Heike Wehrheim
Co-Supervisor: Christian Soltenborn

UNIVERSITÄT PADERBORN
Die Universität der Informationsgesellschaft

September 2, 2009

Contents

What do I mean by the word 'mean'?
What do I mean by the word 'word'?
What do I mean by 'what do I mean'?
What do I mean by 'do' and what do I do by 'mean'?
And what do I do by do by do and what do I mean by wasting your time like this?
Good night.

MICHAEL PALIN in Monty Python's Flying Circus, Episode 12 [CIG+89]

1 Introduction

All languages—human languages, formal languages, or programming languages—need to associate their possible expressions, the *signifiers*, with its intended meaning, the *signified*. The signified is, however, just an imaginary entity; every communication about this entity needs to use again some signifier, maybe from the original language, possibly from another language [Sau16].

The meaning of a single expression and also the definition of the meaning of all possible expressions of a language is called *semantics*.[1] A language's semantics is complemented by its *syntax*, which defines structural properties of the expressions which can be made in the particular language.

But how can a semantics definition actually define the meaning of a language? Due to the mentioned impossibility of naming an signified without a signifier, the meaning of a language needs to be defined in another language. The latter language can be called the *semantic domain* of the new language. A *semantic mapping* links the semantic domain with the syntax elements of the new language and thus manifests the actual semantics of the new language [HR04].

That semantic domain should be well chosen for its task; generally, it should be well-known, in order to alleviate the dilemma that one needs to use another language to define a language. If one intends to use formal methods on the language to be defined, the semantic domain should already support formal methods.

A common task of a software engineer is translating between different languages; for example, a translation occurs, when a task specified in some problem domain is implemented using a programming language. Such a translation may be impeded by the so-called *semantic gap*. The gap opens as soon as two languages express a certain situation in such a different manner that the translation from one to another is no longer obvious for a human translator. Thus, the semantic gap is a human factor, its width varies depending on the person and the languages.

When a software project for an often re-occurring problem domain is realised, *domain-specific languages* (DSL) can be used to reduce the semantic gap between the problem domain and the language,[2] as opposed to general purpose languages such as Java, C or UML. A domain-

[1] Furthermore, semantics is also the name for the study of meanings. It is a bit ironic that the word semantics has quite overloaded semantics.

[2] In the context of domain-specific languages, the term "language" includes both modeling and programming languages.

specific language is specially tailored to the particular problem domain and thus eases the realisation of the project.

Thus, domain-specific languages can alleviate the problem of semantic gaps; yet, they introduce another (solvable) problem: they are languages and thus they need semantics which have to be defined in some way. The smart way to do so is to use another domain-specific language which is specialized in specifying semantics.

The *Dynamic Meta Modeling* (DMM) approach [Hau05] aims at providing such a domain-specific language; it is specialized on modeling languages—both domain-specific and general purpose—whose syntax is specified using object-oriented meta models like MOF [Obj06a]. For specifying the semantics, DMM uses a concept based on graph transformations; however, its features have been specialised and extended to match the object-oriented approach. Such features include structuring the semantics by invocations, code-reuse by rule overriding, and a notation which is related to the object-oriented UML communication diagrams.

In its current state, DMM offers a specialized tool for specifying the semantics. Testing and executing the semantics is however done by GROOVE [Ren04, Ren09], a general purpose tool for executing and analysing graph transformations. Special transformation tools have been created to convert the DMM-specific models to graphs that can be processed by GROOVE.

The use of the existing tool GROOVE facilitates the implementation of DMM significantly, as a graph transformation engine is a quite complex matter. However, the use of GROOVE also introduces a new semantic gap, which is quite significant. As an example, we will look at UML activity diagrams; a DMM semantics specification for UML activities is being developed simultaneously to this thesis. Consider the UML activity diagram in figure 1.1 and compare it with figure 1.2, which shows a screen shot of GROOVE displaying the graph that was produced by the transformation tool that converts DMM models to GROOVE graphs. To emphasise it: Both figures show the same information just in different languages.

The semantic gap should be obvious. It results from two reasons: The activity diagram is shown in its *concrete syntax*, i.e., the primary notation for the user interface. The concrete syntax gives a distinctive look to meta model element instances and lets the user arrange them in an arbitrary layout. However, each model has an underlying *abstract syntax*. This is the actual instance of the meta model; it is thus suitable for automated processing. The example model in abstract syntax consists of 22 objects, which is still a manageable number in a user interface. The transformation of the model to a GROOVE graph, however, yields a graph containing 328 nodes. The GROOVE graph contains more nodes because it uses nodes for expressing information that could be expressed otherwise in the source model. As you can see, such a graph overburdens the layout algorithm employed by GROOVE.

If the developer of a DMM semantics searches for the reason of a fault, unfortunately, he needs to examine these GROOVE state graphs. As the execution of a model modifies the state, there is also more than one state to be examined. He might use a DMM tool which provides a function that can convert a GROOVE state back into the abstract syntax of the original language. This is, however, still quite cumbersome as it involves a number of manual actions and switching the tools. Generally, it can be said that the task of debugging DMM semantics is currently neither easy, nor pleasant.

An ideal working environment for a developer would be an *integrated development envi-*

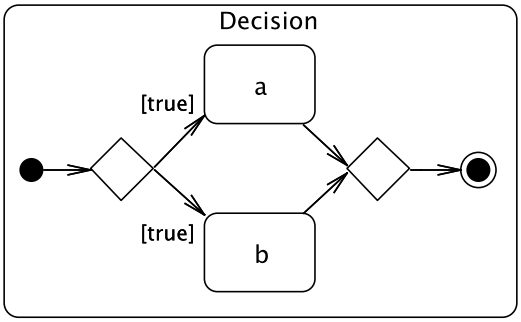

Figure 1.1: Example of a simple activity diagram with a decision node and two actions

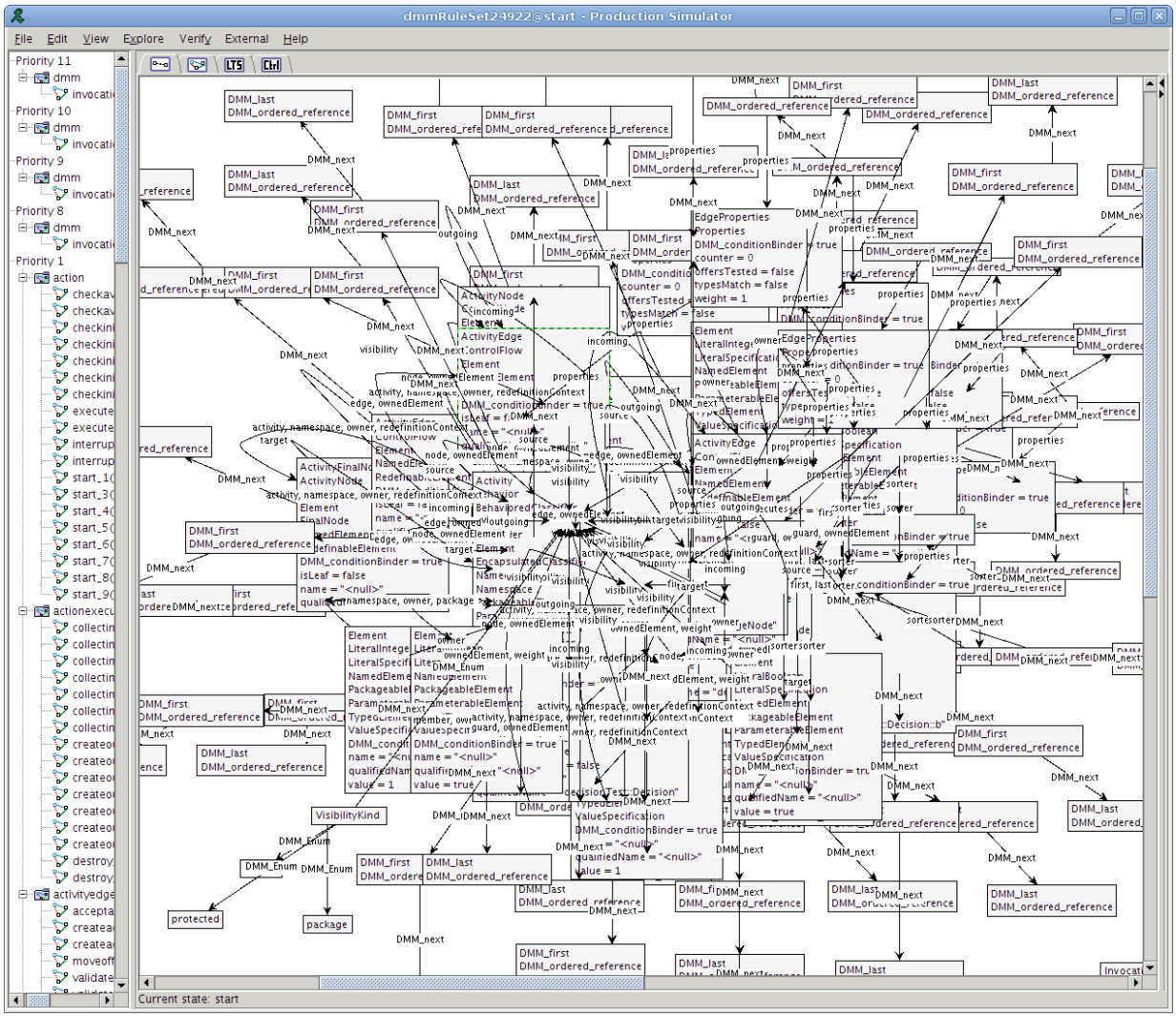

Figure 1.2: Screen shot of GROOVE displaying the graph representing the activity from figure 1.1

ronment (IDE) for DMM. Such an environment would require neither tool-switching nor language-switching; the developer of DMM semantics would not need to care about the tool GROOVE. While the use of external components is absolutely legitimate, this should occur transparently to the user.

Integrated development environments are already well established in the world of "classic" programming languages such as Java. Typical components of an IDE include a code editor, a code launcher, and a debugger.

DMM already provides an editor for its semantics; it is realized using the *Eclipse Platform* [Ecl09a], a framework which is designed for building IDEs. The technology used for creating the DMM semantics editor can be also used for creating editors for the actual programs implemented in DMM-specified languages. Furthermore, already existing editors based on this technology might be re-used for DMM; for instance, there is already an editor for the aforementioned UML activity diagrams. Ideally, this editor can be integrated into a DMM IDE and used for executing, exploring and debugging activities.

1.1 Goals

The general goal of this thesis is to make the first step towards an integrated development environment for DMM by providing execution and debugging capabilities that discharge the user of using GROOVE. More concretely, the following goals are targeted:

- Provide a function that *visually executes* a model;[3] this means that the execution is performed step-wise while the current execution state is visualized in the concrete syntax of the model.
- The *speed of the execution*, i.e., the length of the delay between two steps, shall be configurable.
- It should be possible to define the *length of a step*; at the simplest level, a step corresponds to the execution of one rule. However, it might be feasible that one step comprises the execution of more than one rule.
- There should be a generic way to enable *existing model editors*—for instance, the UML activity editor—to visualize the model execution performed by DMM. Especially, this includes the visualization of runtime-specific information in concrete syntax directly in the existing editor.
- A *debugger* shall give the user the opportunity to *suspend* the execution in order to let him explore a certain state more extensively. Suspension may occur both manually and automatically in certain, predetermined situations. The functionality of the debugger should be aligned to the commonly known functionality of debuggers in integrated development environment for classic programming languages.

[3] The execution will be still internally performed using GROOVE, which interprets the rules of a DMM semantics. Thus, the title of this thesis calls the tool a *visual interpreter*.

- Furthermore, the user should be able to *choose the execution path* whenever it is not uniquely defined by the executed model. It should be possible to make this choice both transiently and persistently.

- Besides the development of DMM semantics, the *development of a model instance* in a language with completed DMM semantics should be also regarded as a valid use case of the tool.

The resulting functions shall be integrated using the Eclipse Platform in one environment, which already provides the editor for DMM semantics. Due to the integration, the meaning of the word "tool" becomes ambiguous. The whole environment may be regarded as a tool; a single editor for a diagram may be regarded as a tool as well. For the ease of understanding, we will identify the set of the targeted functions as the *planned tool* and give it the name *"DMM Player"*.

1.2 Overview

The structure of the document at hand is aligned to the software development process. That means that the chapters roughly represent the phases of the process.[4] This has the side-effect that information about a certain function may be scattered across the chapters. In order to alleviate this disadvantage, the document will structure the chapters by concepts and provide forward and backward references between them.

It is assumed that the reader of this thesis has a basic knowledge of UML and graph transformations. Thus, a specific introduction to these topics will not be given. Still, the thesis will cover the relevant areas of these topics along the more specific sections which base on them.

Still, an introduction to furter concepts and techniques that are essential for this thesis will be given in chapter 2. This includes basics of DMM and debugging techniques.

Chapter 3 continues with the requirements analysis for the DMM Player. In this chapter, requirements induced by the involved processes and methods will be identified and formalised as use cases.

The conceptual work that is required for realising the requirements identified before is done in chapter 4. The developed concepts are mainly related to controlling the model execution in certain ways and situations.

Chapter 5 will move on to a more software technical level; it will present the architecture of the DMM Player and shed light on some selected technical topics.

Chapter 6 concludes this thesis with a summary and an outlook.

[4]This structure was chosen because there is no 1-to-1 mapping between requirements, conceptual solutions, and implementations. Thus, several requirements may be fulfilled by a single component. A structuring by concepts is thus not possible.

1.3 Typographic Conventions

In order to ease the reading of this thesis, this section will give an overview over the typographic conventions used in it.

- The thesis uses footnotes quite often; the information in footnotes is either a backward reference or further information that might be interesting but not essential for the particular topic. Thus, footnotes can be safely skipped.
- Words printed in *italics* either signify a newly introduced term or emphasis.
- Words printed in a sans-serif typeface are references to model elements or computer processable data such as class names, URIs, etc.

Foundations 2

This chapter provides an overview over concepts and technologies that are essential to this thesis and the DMM Player. This will both provide the reader the necessary knowledge for understanding the subsequent chapters, but also partially serve as the base for the requirements analysis which requires an analysis of the current situation.

We will begin with a closer conceptual look at Dynamic Meta Modeling in section 2.1. The following section 2.2 will survey the current state of the implementation of DMM. As the DMM Player will provide a debugger, section 2.3 will examine common concepts and techniques used by debuggers. Section 2.4 concludes the foundations with a brief review of further software components that are relevant to the DMM Player.

2.1 Dynamic Meta Modeling

Considering visual modeling languages such as the UML, it is often possible to observe a certain imbalance in the language specification: While the abstract syntax is commonly defined by formal constructs like grammars or meta models, the semantics is most often only specified using natural language. Thus, a formal specification is missing in this case. The specification of the execution semantics of UML activity diagrams is such a case.

A specification that is only available in human language has several drawbacks: The specification is always subject to interpretation; there is the chance of misunderstandings or several, inconsistent interpretations. Furthermore, an informal specification does not allow any automated processing such as execution of a program implemented in the respective language or model checking on a program or the specification itself.

The goal of *Dynamic Meta Modeling* (DMM) [Hau05] is to provide a technique for specifying the execution semantics of visual modeling languages. DMM wants to provide an approach which is superior to other approaches by minimizing the semantic gap between the semantics specification and the modeling language. Thus, DMM provides a specialised language, which is both easily understandable for its target audience, and adequately expressive for formulating semantics.

We will now look at the core concepts of DMM in the sections 2.1.1—2.1.3. Section 2.1.4 will review a development process for DMM semantics. Section 2.1.5 will introduce an example application of DMM which will be also later revisited as running example.

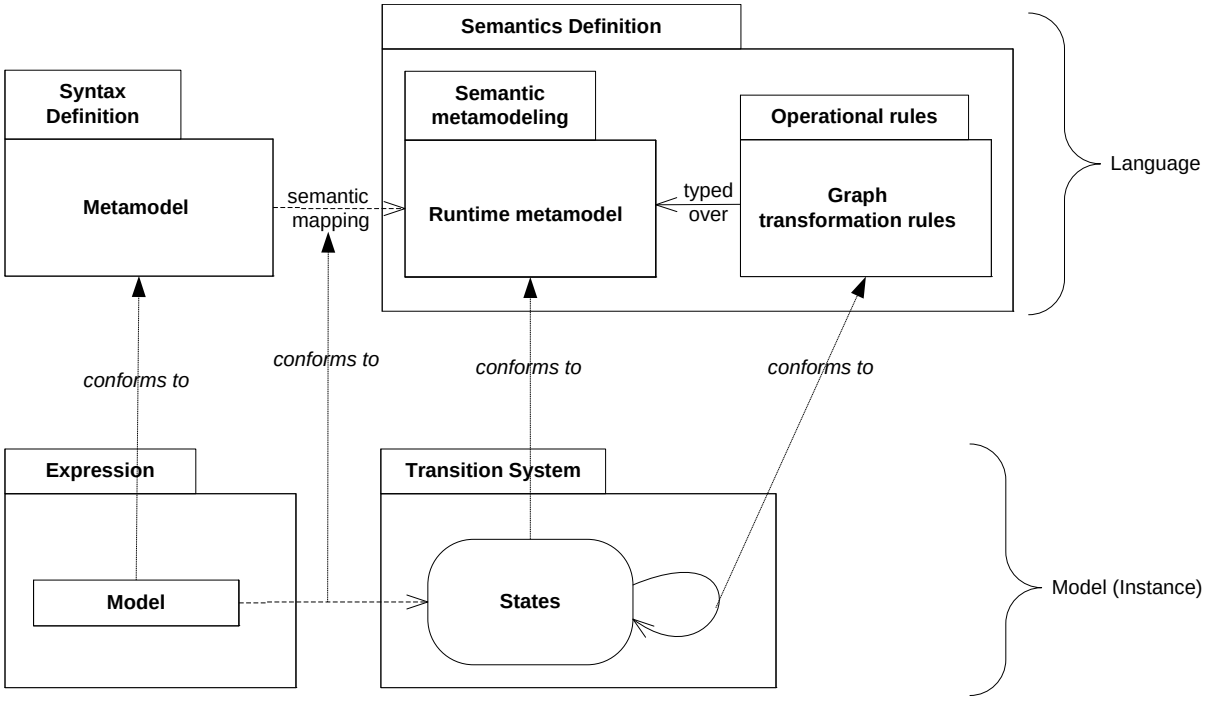

Figure 2.1: Outline of the DMM approach (according to [Sol06])

2.1.1 DMM at a Glance

The core of a DMM-based semantics specification consists of two parts (Figure 2.1 provides an overview over the concepts used by DMM and their relationships): Graph transformation rules [Roz97] specify the dynamic semantics. The rules operate on instances of a meta model specially created for the semantics of the particular language.

This *runtime meta model*[1] is typically based on the original *syntactic meta model*, which stems from the syntax definition of the language. Its purpose is to extend the static meta model by concepts that are only applicable to the execution of a model. Such concepts may be tokens, a reference to an active state or other elements that indicate the current execution state of the model. A semantic mapping relates elements from the static meta model to the corresponding elements from the semantic meta model. More on the runtime meta model in section 2.1.2.

The *graph transformation rules* specify how an instance of the runtime meta model—or in short: a *state*—will evolve during the execution of the model. The so-called left hand side of a rule determines whether a rule can be applied to a state at all. When a rule is applied, the current state is modified analogously to the differences between the left hand side and the right hand side of the rule. This way, a rule is able to move tokens from one place to another or perform any other modification to a state. Section 2.1.3 will give a more in-depth look on rules.

Applying all applicable rules to the initial state and additionally recursively to all resulting

[1]The runtime meta model is sometimes also called *semantic domain meta model* or *enhanced meta model*.

states will result in a—possibly infinite—set of all reachable states. Reaching a state from another state with a single rule application is called a *transition* from the former to the latter state. A tuple of the set of states and the relation representing the transitions is a *transition system*. Labeling the transitions with the names of the rules that caused them will constitute a *labelled transition system*.

A transition system can be denoted as a directed graph in which the nodes represent the reachable states and the edges the transitions.[2] The graph may be cyclic, indicating that the transformation of a state will yield a state that was already reached before.

2.1.2 The Runtime Meta Model

As stated before, the runtime meta model provides the structures needed for representing the execution states of a language. Obviously, an execution state of a model needs to be somehow interweaved with the syntactic structure of the model. In the case of UML activity diagrams, tokens are a pure part of the execution state and do not occur in the syntactic meta model. However, tokens are always located on activity nodes, which are declared in the syntactic meta model.

DMM treats the syntactic meta model and the runtime meta model as separate entities; the connection between the two meta models is realized by a mapping from the syntactic to the runtime meta model. This mapping provides means to create instances of the runtime meta model out of instances of the syntactic meta model. Typically and pragmatically, the runtime meta model will be an extended version of the syntactic meta model, i.e., the runtime meta model will add classes and associations, but not transform classes from the syntactic meta model in some way. Figure 2.2 shows as example a fragment of the syntactic meta model for UML activity diagrams; The corresponding runtime meta model is shown in figure 2.3. Note the additional generalisation that connects elements that originate from the syntactic meta model with new elements.[3]

If the syntactic meta model already contains all structures for representing the runtime information, the mapping may be the identity function; i.e., the runtime meta model will be equal to the syntactic meta model. This may be for example the case with petri nets; the dynamic information of a petri net is the number of tokens on a place. In a syntactic meta model designed for petri nets, this information will be already present, because the number of tokens on a place is also part of the syntax. This is due to the fact that petri nets may provide an initial marking that assigns tokens to places.

As seen before, DMM allows a mapping that arbitrarily modifies the elements of the syntactic meta model to get the runtime meta model. In the example in figure 2.3 a new super class is added to the classes Action and ObjectNode. Unfortunately, this way of extending the meta model makes it hard or even impossible to reuse existing artifacts that have been created for the syntactic meta model. Examples for such artifacts are data structures or editors. In order to solve this problem, a more restricted method of extending the model is currently

[2]Obviously, a complete graph of a transition system can only be denoted if the transition system is finite.
[3]Generalisation is not the only way of extending the syntactic meta model, though.

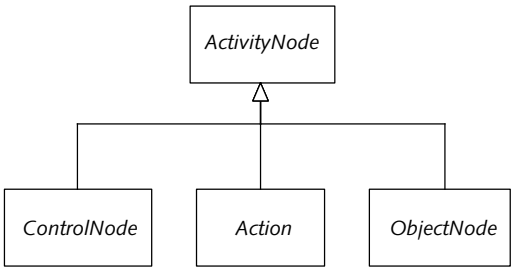

Figure 2.2: Fragment of the syntactic meta model of a UML activity diagram

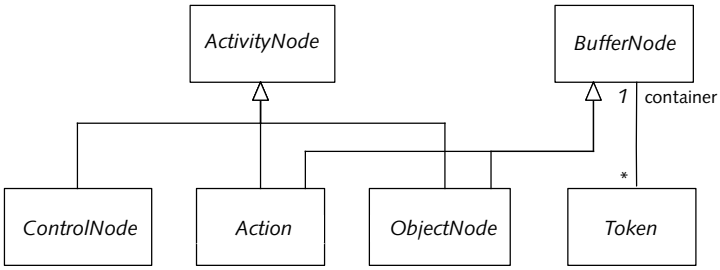

Figure 2.3: A fragment of the original runtime meta model for UML activity diagrams according to [Hau05]

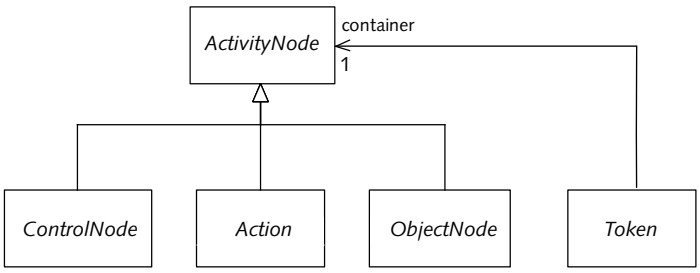

Figure 2.4: Fragment of a simplified runtime meta model for UML activity diagrams

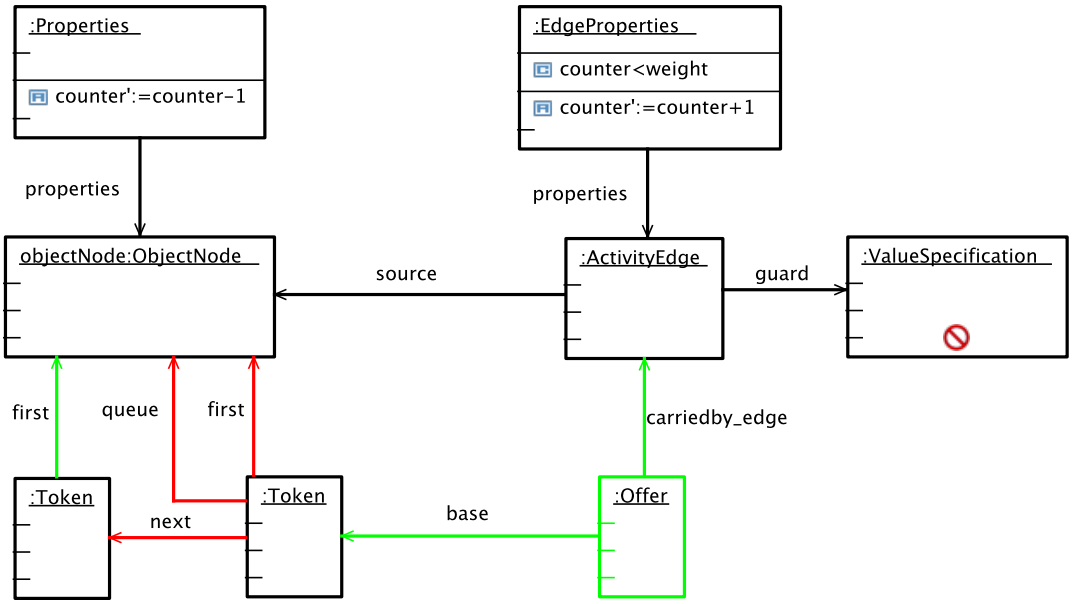

Figure 2.5: The rule objectNode.dequeue()# from the DMM UML activity semantics [Hor09]

practised, which allows the reuse of existing artifacts. A runtime meta model created using this method does not modify existing elements in any way. Therefore, all associations between new runtime classes and existing classes are realized uni-directional pointing to the existing classes. However, the advantage of being able to reuse existing code is slightly diminished by a reduced expressiveness. At least the current tool support[4] does not allow multiplicities on the non-navigable end of associations. Furthermore, the grouping of existing classes using new super-classes is also no longer possible, as can be seen in the example in figure 2.4. The token class now needs to reference a class that is more general than actually required.

2.1.3 DMM Graph Transformations

Compared with traditional graph transformation rules, the rules employed by DMM exhibit quite a few specializations which have been made in order to provide an optimum of usability and adequacy for their tasks.

Let us look first at the topmost structure of a rule (see figure 2.5 for an example of a DMM rule). Contrary to the traditional structure dividing a graph transformation rule in a so-called *left-hand side* and *right-hand side*, DMM uses a view that integrates both sides into a single diagram. The left-hand side[5] is composed of the nodes and edges colored black or red. An alternative notation for black-and-white media is the annotation of originally red elements

[4]EMF Ecore models are currently used for representing DMM runtime meta models. More later in section 2.2.2.

[5]The left-hand side specifies when the rule matches, i.e., when it is applicable. This is the case when the nodes and edges of the left-hand side constitute a morphism to the instance of the runtime meta model—or in the traditional words of graph transformations: to the host graph.

with the text {destroyed}. The right-hand side[6] is composed of the elements that are colored black or green. The alternative notation for green elements is {new}. Summarizing, black elements are those elements that occur both in the left-hand and the right-hand side and which thus do not cause any change. Green elements cause an object or link to be newly created when the respective rule is applied. Red elements cause the respective elements to be deleted from the current instance of the runtime meta model.

DMM rules use a notation similar to UML communication diagrams and object diagrams. Like the objects in those diagram types, each node of a DMM rule has a textual label, which denotes the name of the node followed by a colon and the type of the node. The name of the node is optional and may be omitted; in this case, the label of the node starts with the colon. Any class of the associated runtime model may be used as the type of a node. A node only matches an object in the runtime model if its type is equal to or a super type of the runtime model object's type.

The rightmost node of the rule in figure 2.5 carries an icon looking slightly like a prohibitory traffic sign. This indicates a *negative application condition* (NAC). A rule is only applicable, if nodes or edges that marked this way are not present in the morphism from the rule to the runtime model.

DMM differentiates three types of rules: *Bigstep rules, smallstep rules,* and *premise rules*. The type of a rule can be distinguished by the rule name: The name of bigstep rules ends with the hash sign #; premise rules are prefixed with the string "P_". Smallstep rules can be recognized by having neither of the formerly mentioned markings.

The semantic difference between bigstep rules and smallstep rules stems from another special feature of DMM, *invocations*. Both bigstep and smallstep rules may perform an invocation in order to initiate the execution of a further smallstep rule. Bigstep rules are not invoked, but automatically applied when they are applicable. However, pending invocations inhibit the application of bigstep rules; until all invocations have been completed, no bigstep rules are applied.

In a DMM rule, an invocation is denoted by a labeled arrow pointing to a node. The label text looks similar to a method invocation from conventional programming languages. See figure 2.6 for an example of a DMM rule performing two invocations.

If a rule performs more than one invocation, the order of the invocations needs to be defined. This is accomplished by assigning each invocation a sequence number which is displayed at the beginning of the invocation label.

Invocations may have parameters, which are denoted in the round brackets after the name of the invocation; thus, the invocation getToken(action, actionExecution) from the example in figure 2.6 has two parameters: action and actionExecution. The names of the parameters define the source of the values of the actual parameters; for each parameter name, the rule must contain a node having the same name. Just like method calls in classic object-oriented programming languages, each invocation has one further parameter that does not appear in the parameter

[6] Together with the left-hand side, the right-hand side of a rule specifies how the host graph shall be transformed when the respective rule is applied. Elements present in both sides will be left untouched. Elements that are only present in the left-hand side will be deleted, while elements that are only present in the right-hand side will be created.

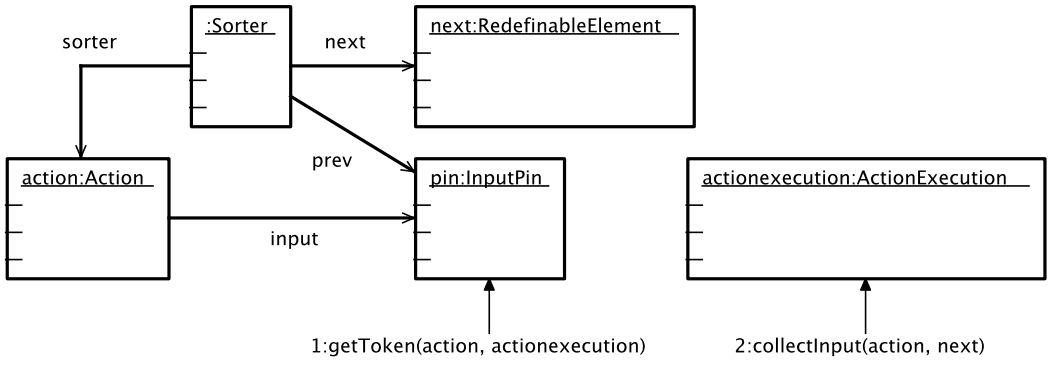

Figure 2.6: The rule actionexecution.collectInput(action:Action, pin:InputPin) from the DMM UML activity semantics [Hor09]

list. The node an invocation points to is the so-called *target node* of the invocation. The object bound to that node is passed during the actual invocation just like the other actual parameter values.

A smallstep rule invoked this way receives the actual parameter values of the invocation as values that are bound to its parameter nodes. Parameter nodes are just like other nodes with the exception that the node has been included in the formal parameter list of the node. In the diagram, this can be recognized by comparing the names of the nodes with the parameter names that are visible in the rule's signature. The signature of the smallstep rule shown in the example in figure 2.6 is actionexecution.collectInput(action, pin). Thus, when this rule is invoked, the values of the nodes with the names action and pin will be initialized with the values of the actual parameters that have been specified during the invocation. Furthermore, the name preceding the dot in the rule's signature signifies the name of the *context node*, the node that will be initialized with the value bound to the target node during the invocation.[7]

After a smallstep rule has been invoked and the parameter nodes have been bound to their respective actual values, the remaining nodes are finally subject to the usual matching in search for an morphism to objects of the runtime model.

This leads us to a topic that has not been covered yet: An invocation is not directly tied to a single smallstep rule; in fact, an invocation triggers a search for a rule that matches its criteria. This is performed dynamically, i.e., the search takes place at runtime. The criteria of the search are:

- The invoked rule must have the same name as the invocation.
- The types of the formal parameters of the invoked rule—including the context node—must be compatible to the types of the actual parameter values of the invocation. That

[7] To come back to the analogy to object-oriented programming languages: Just like defining methods in a class in those languages, the context node also provides means to structure the rules of a rule set by assigning them to a class they are supposed to primarily operate on.

means that the type of a formal parameter must be equal to or a super class of the type of the actual parameter.

- A match on the runtime model is found for the remaining nodes of the invoked rule.

This scheme allows the realization of switches in the control flow. Depending on the current runtime model, one invocation may call another rule out of a set of several rules with the same name but different left-hand sides. However, there is no direct way for a DMM specification to specify an else-case that matches if the specific conditions that were checked before did not match. The developer of a DMM specification must make sure that the left-hand sides of smallstep rules cover all states in which they may be invoked.

If a rule performs an invocation with a certain name, but none of the smallstep rules with that name have a matching left-hand side, the DMM specification is incorrect. A concrete occurrence of such a situation is called *DMM specification failure*.

DMM rules offer quite a number of further features, such as the aforementioned premise rules, universally quantified nodes, conditions and assignments on attributes [Bau08], rule overriding [Fis07, EFS09], etc., etc. As these features are not directly relevant for this thesis, a description of these features is omitted here. The interested reader may refer to the bibliography for further reading.

2.1.4 Specification Processes for DMM Semantics

After having seen what can be done with DMM, we move on to the topic on how to develop DMM semantics.

A test-driven specification process for DMM semantics has been proposed in [SE09]. Besides some lower level implementation guidelines given in [Hau05], this is currently the only explicit documentation of a development process for DMM semantics.

At the top-level, the process is divided into two phases:

The goal of the first phase is to establish an understanding for the semantics of the particular language by creating examples in that language and defining the anticipated behaviour of each example by traces of execution events. Each example should cover a single language construct in a minimal, but theoretically executable way. The developer should start at the most basic language constructs and gradually proceed to the more advanced language constructs. After creating an example, the developer needs to figure out the behaviour of the example and identify relevant events during its execution (e.g., the execution of an action). These events are informally recorded in the aforementioned traces of execution events. When all language constructs have been covered, the first phase is finished.

The second phase comprises the actual development of the DMM semantics (see figure 2.7 for a diagram describing the second phase). Beginning with the basic language elements,[8] the

[8] Note that, while the first phase worked on language *constructs*, the second phase is specified to work on language *elements*, i.e., on a finer level. The first phase cannot use just single elements, as a model consisting only of a single element is not necessarily executable. The second phase uses elements because semantics are typically structured element-wise.

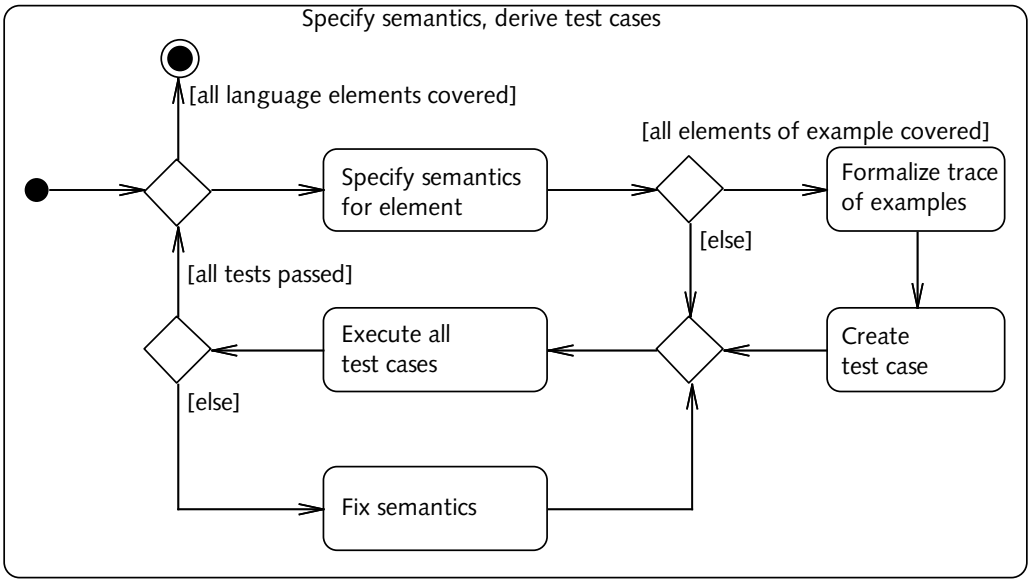

Figure 2.7: Test-driven semantic specification process (according to [SE09])

developer writes DMM rules that handle the execution semantics of the particular element. The runtime meta model is also created or extended as needed. As soon as the semantics of an example created in the first phase are completely covered by the available rules, that example is turned into a test case. This requires modifying the execution event traces in such a way that the events correspond to the newly created rules and their names. The traces are then formalized using the temporal logic dialect CTL [CES86]. From now on, whenever a new DMM rule has been created or an existing rule has been modified, the available pairs of example models and CTL expressions are used to ensure that the current semantics actually deliver the expected behaviour. Obviously, whenever a test fails, the developer needs to analyse the reasons for the failure and correct the semantics accordingly. Suitable model checking software can provide a counter-example for each failed test. A counter-example is a path through the transition system of the example model for which the CTL expression does not hold.

2.1.5 Example: DMM-specified UML Activity Behaviour

We already encountered at some places references to a DMM semantics specification for UML activity diagrams. Now, it is time to take a slightly closer look at this specification.

The semantics specification for UML activities can be called the reference implementation for DMM. The first comprehensive work about DMM [Hau05] already provided a specification for UML activities as a case study. However, this specification covered only the most basic parts of UML activities. The realization of the first complete DMM specification is currently subject to a diploma thesis [Hor09] which is scheduled to be finished at the same time as the thesis at hand. The base for these specifications is of course the official UML2 specification [Obj09] which defines the semantics of activities with natural language.

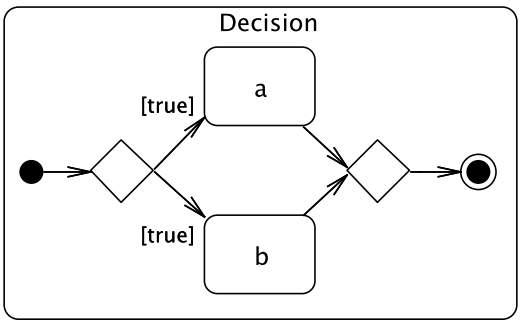

Figure 2.8: Example of a simple activity diagram with a decision node and two actions

UML activities feature quite extensive semantics; the original UML2 specification for activities spans 401 pages.[9] At the time of writing, the DMM semantics comprise 42 bigstep rules, 172 smallstep rules and 3 premise rules. A rule contains an average of 4.99 nodes and 4.45 edges.[10]

We will try anyway to put that all in a nutshell: Basically, the execution of an activity diagram is controlled using tokens—differentiated in control tokens and object tokens—and offers. Initially, an activity does not contain any tokens or offers. During the initialization phase of the execution, control tokens are placed on all initial nodes and on all actions with no incoming edges. Furthermore, the whole execution is represented by an activity execution object. Whenever a control token is put on an action, the execution of that action starts. This is is represented by a newly created action execution object that is attached to the action. Initial nodes or finished actions send out offers to their outgoing edges. Offers serve as a pathfinder for the tokens they are linked to; i.e., offers search a way over flow edges and decision nodes until they reach an activity node that can carry the token. If an activity contains several tokens or offers, they flow concurrently until they are synchronized at a join node or an action awaiting several input tokens. If a token reaches a flow final node, it is deleted. If there are still other tokens in the activity, execution will continue. If a token reaches an activity final node, all tokens in the activity will be deleted and the execution will come to an immediate end.

Of course, an explanation condensed like this needs to omit many special features. Such features include object tokens that transport data between actions, edge weights, exceptions, invocations, structured activities, etc.

We will now look at a concrete activity diagram which will also be used as a running example in the upcoming sections of this thesis. Figure 2.8 shows this activity diagram; it contains—from left to right—an initial node, a decision node, two opaque action nodes, named a and b, a merge node and a final node. Both control flow edges running from the decision node have the boolean literal [true] as guards. This means that it is undetermined whether the action a or

[9] That are the chapters 11 and 12 of the UML2 superstructure specification on the pages 217–294 and 295–417. This number includes the syntactic specification, examples and a lot of white space, though.

[10] Fortunately, the DMM implementation provides a function for automatically gathering statistics about a rule set.

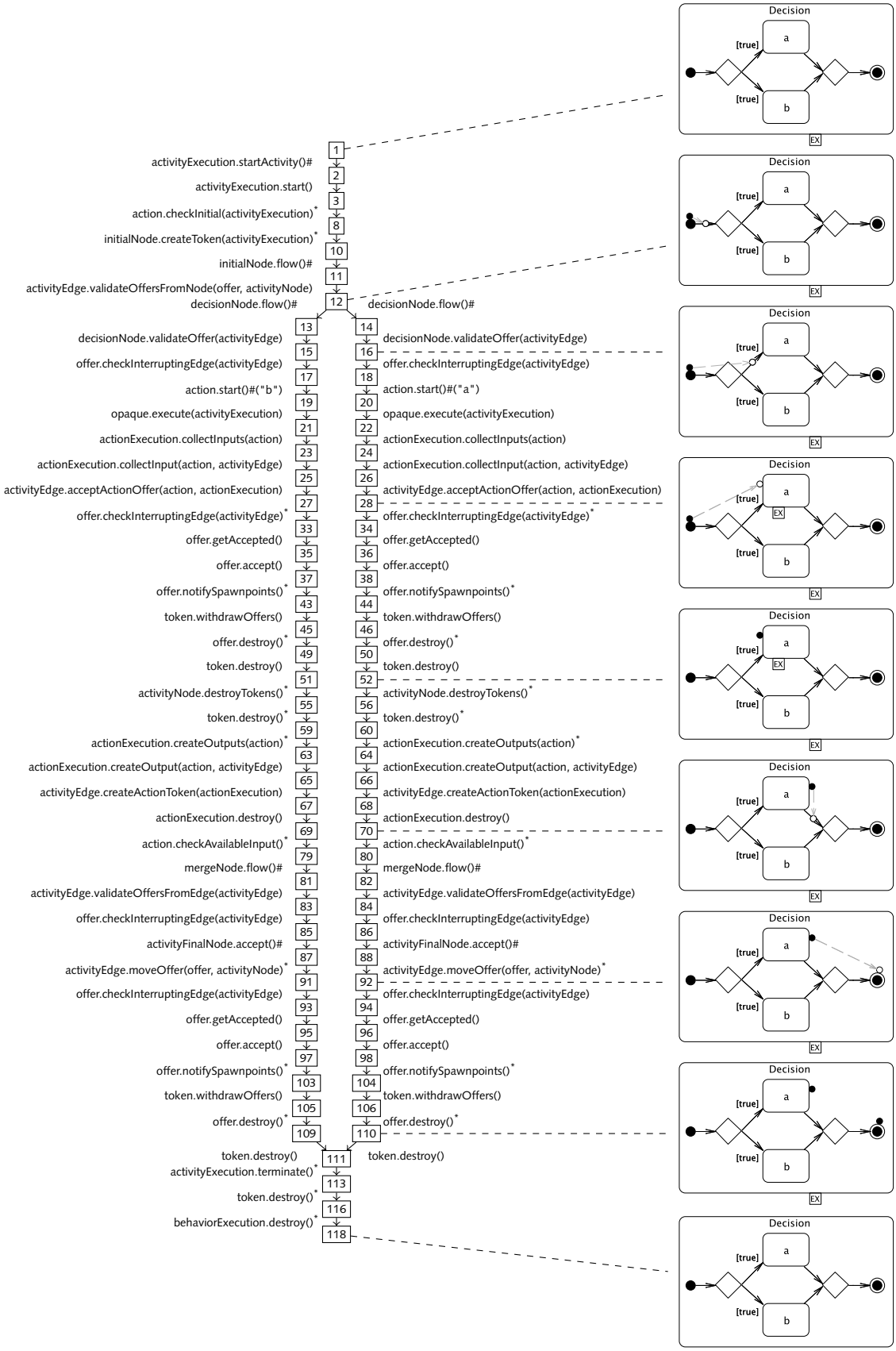

Figure 2.9: The transition system of the example activity diagram with the concrete view of selected states.

b will be executed before a single execution of the activity.

The DMM rules for UML activities can now be used to calculate the transition system specifying all possible states of the model and the transitions between them. This transition system is depicted in figure 2.9.[11]

State 1 is the initial state of the activity; as you will already have noticed, the concrete syntax of the state is depicted in the small diagram that is connected to the state node of the transition system. Compared with the base activity in figure 2.8, the initial state additionally shows a node labeled EX attached to the activity. This is the concrete syntax representation of an ActivityExecution object which stems from the transformation of the model to the runtime meta model which is performed before the actual execution.[12]

The transitions between the states 1 and 10 initialise the activity with one token and one offer that belongs to the token. The concrete syntax of state 12 shows, that the offer—represented by the small hollow circle—already flowed onto an edge. The black, filled circle represents the token which is still located on its initial position. The relation between the token and the offer is represented as a dashed edge between both elements. The visualization of this relation becomes important in activities with more than one token and offer.

The decision node with the non-deterministic guards causes the fork in the transition system. The left branch—beginning with state 13—contains the execution of the action b. Action a is executed on the other branch. This can be seen—besides by looking into the actual states—by looking at the transitions representing the rule action.start()#, which occur between the states 17 and 19, and 18 and 20, respectively. In the definition of the rule action.start()#, the name of the action to be executed is marked as a so-called *emphasised attribute* which causes the actual value of the attribute to be shown on the respective transition.

When the offer reaches an action node, the execution of that action starts. This can be seen in state 28 by the EX node attached to the action a. Afterwards, the token is transferred to the action (state 52). The execution of the action destroys the input token and creates a new token as its output. State 70 shows the token with its offer on the outgoing edge.

In state 92, the offer reaches the final node; the token reaches it in state 98. Note that the branches of the transition system did not merge yet, even though the token already passed the merge node. However, the state still contains information specific to the branch and thus keeps the branch alive. The branch specific information will be cleaned up in the following steps.

The final transitions perform a clean-up of the model by deleting runtime objects such as the remaining token and the activity execution.

[11] The transition system was calculated using the tool GROOVE, which will be described later in section 2.2.3. In order to keep the transition system—relatively—small, it has been slightly reworked afterwards. Transitions which are triggered by DMM helper rules—which are common to all rule sets and not part of the particular semantics specification—have been removed from it. Furthermore, invocations on universally quantified nodes cause a chain of transitions representing the same rule for each object of the collection the invocation is performed on. Such chains have been also collapsed into one single transition. To indicate the reworked parts, transitions which actually comprise several transitions are marked with an asterisk*. The state numbers also indicate the omitted states at the respective places.

[12] The concrete syntax for the objects from the runtime model has also been created as part of this thesis. The figures of the activities in concrete syntax next to the transition system have been actually created with the DMM Player.

2.2 Implementation of Current DMM Components

We will now examine the more technical sides of DMM. We will look at the current state of the software that realizes the concepts of DMM and at the third party software components that help reaching that goal.

The current state of the DMM implementation consists essentially of a graphical editor for DMM rule sets (cf. section 2.2.4) and transformation tools that convert DMM rule sets and runtime models into a format that can be processed by the external tool GROOVE (cf. section 2.2.3). The runtime meta model and its instances are created using functions provided by the Eclipse Modeling Framework (cf. section 2.2.2). The base for all these components is provided by the Eclipse Platform, which will be described in the next section.

2.2.1 Application Integration with the Eclipse Platform

As described before, DMM consists of a number of independent components, partially of dedicated components and partially of already existing components that have been developed independently of DMM. The integration of all these components into one environment is provided by the *Eclipse Platform* [Ecl09a].

The platform itself is just meant as a base for software applications of any kind. It provides advanced UI capabilities in the form of UI widgets and controllers and—on the larger scale—a configurable multi document interface, called the *Eclipse Workbench*. Furthermore, the platform provides a plug-in or component management infrastructure based on OSGi [OSG09].

The actual functionality of an Eclipse application is provided by the plug-ins. Numerous plug-ins for Eclipse have become available. Among those, there are also plug-ins providing frameworks for modeling tasks such as the *Eclipse Modeling Framework* (EMF) and an EMF-based implementation of the UML2 meta model which will be described in the following sections.

The dedicated DMM components are also provided as Eclipse plug-ins; thus they directly integrate into the Eclipse workbench.

2.2.2 Model Representation with EMF

The *Eclipse Modeling Framework* (EMF) [SBPM09] is described as a "a modeling framework and code generation facility for building tools and other applications based on a structured data model" [EMF09].

EMF can be used to specify a meta model with the visual modeling language *Ecore*.[13] With an Ecore-specified meta model, EMF allows to generate Java code for representing instances of the meta model at runtime. The generated code also provides the possibility to save model instances as XMI files, the standard interchange format for model data [Obj07].

[13] Ecore is an EMF-specific implementation of *EMOF* (Essential Meta-Object Facility), the basic variant of *MOF* (Meta-Object Facility) [Obj06a]. MOF has been standardised by the OMG and is used for specifying the meta model of the UML2 [Obj09].

EMF and Ecore is used for all meta modeling tasks of the DMM implementation: The meta model for DMM rule sets is specified using Ecore; likewise, all runtime meta models of DMM specifications need to be specified with Ecore.

When such a rule set and runtime model are available, they can be executed; the current implementation does not support this directly, but transforms the models to graphs and processes them with an external tool. More on that in the next section.

2.2.3 Graph Transformations with GROOVE

The most demanding part of a DMM implementation is certainly that one which performs the actual graph transformations. Thus, it is a good practise to use existing, well-approved software components for this task. GROOVE (Graphs for Object-Oriented Verification) [Ren04, Ren09] is such a component. The Java-based GROOVE tool set allows for the definition of graph transformation rules and the execution of these on a state graph. This may happen either step-by-step or automatically, producing a transition system of all reachable state graphs. Furthermore, GROOVE allows model checking operations such as the evaluation of CTL expressions on a transition system. GROOVE also provides graphical user interfaces for editing the rules and the state graphs and for visualizing and browsing transition systems. For programmatic processing, GROOVE can be also controlled by its Java method interface.

GROOVE is actively developed at the University of Twente in the Netherlands. It is licensed under the Apache License version 2.0, a free and non-viral software license [Apa04].

Obviously, using a generic tool for graph transformations yields not only advantages, but also some drawbacks. The drawbacks are primarily caused by the need for mapping the DMM data—namely the runtime model, its meta model and the transformation rules—to graphs that can be processed by GROOVE.

GROOVE graphs are simple directed graphs whose nodes and edges can be labeled.[14] The GROOVE counterpart of a DMM runtime model is called a *state graph*; DMM rules are mapped to GROOVE rules. The basic mapping is simple: All nodes and edges in a DMM rule or a DMM runtime model are mapped to a GROOVE node or edge, respectively.

However, GROOVE does not support typing, i.e., the nodes of the state graph cannot be assigned a type that is defined in a separate type graph. Such a type graph would represent the runtime meta model from the DMM side. The matching performed by GROOVE is solely based on comparing the labels of the nodes and edges in the state graph with their counterparts in the rules. In order to support the polymorphism from the object-oriented DMM models in GROOVE, each node of the state graph is labeled with all types it can be bound to. Thus, objects typed over meta models with relatively deep inheritance hierarchies will cause a considerable blow-up of the GROOVE state graph.

That kind of expansion of all super types is not required for the mapping of DMM rules to their GROOVE counterparts. Labeling each node of a GROOVE rule with the type that has been originally specified in DMM is sufficient, because this states the requirement that a node

[14] This is a simplification of the actual situation. In fact, only edges can be labelled in GROOVE; labels on nodes are actually labels on self-edges on the respective nodes. This detail is not relevant for this thesis, though.

from the state graph must have *at least* the labels that are specified in the rule.

However, advanced features of DMM rules such as conditions or assignments [Bau08] or universal quantified structures [Ren06, Bau08] require more elaborate expressions on the GROOVE side, thus also complicating the GROOVE rules significantly. We will skip the explanation of the actual mapping of these features here, as it is not of particular interest for this thesis, but quite complex.

In the current implementation, the mapping is realized by the following components: The DMM2GROOVE component performs the mapping from DMM rules to GROOVE rules. The component EMF2GROOVE [Rhe07] handles the transformation from EMF-based runtime models to GROOVE states and also the backwards transformation from GROOVE to EMF. Both components produce XML files conforming to the *Graph Exchange Language* (GXL) [HSSW09], as this format is used by GROOVE for the external representation of its graphs.

2.2.4 Visual Editor for DMM Rules

As seen before, DMM uses a specialized concrete syntax for its transformation rules in order to provide a language that can be intuitively related to known modeling techniques, especially in the environment of the UML and MOF. A visual editor for DMM rule sets that uses this concrete syntax has been developed as part of a bachelor's thesis [Röh08].

It is based on the *Graphical Modeling Framework* (GMF) [GMF09a] which allows the development of rich graphical editors for EMF-based models in a declarative manner. We will take a closer look at GMF in section 2.4.1.

In order to develop a rule set with the editor, a runtime meta model must be available in Ecore representation. After the user has connected the meta model with the new, empty rule set, he may create DMM rules that are typed over the respective meta model.

2.3 Introduction to Debuggers

The central goal of the thesis is to create a debugger for DMM. Thus, we need to find out what a debugger actually is. The following section 2.3.1 will introduce the basic concepts of debuggers.

It should be noted that debuggers stem from the world of classic, text-based programs and are primarily used in this context. Thus, most literature silently assumes text-based programs. However, we will later see that the concepts of debuggers can be quite well generalized to other programming techniques. For examining the foundations, the following sections will still assume the classic way of programming, though.

2.3.1 The Purpose of Debuggers

According to the Dictionary of Computing [Dai04] a *debugger*, or *debug tool* is ...

> "[...] a software tool that allows the internal behavior of the program to be investigated. Such a tool would typically offer trace facilities [...], allow the planting of breakpoints (i.e. points in the program at which execution is to be suspended so that examination of partial results is possible), and permit examination and perhaps modification of the values of program variables when a breakpoint is reached."

The definition indicates that a debugger—despite its name—does no actual debugging, i.e., some kind of automated fixing of programming faults. Rather, a debugger offers functions that aid a software developer in searching for the causes of an error which occurred before. These functions allow the user to follow and examine the internal state of the program while it is executed. The internal state includes data such as the program counter, the call stack and the values of variables. Thus, the user may find deviations from the expected state which lead to the final error.

Generally, a debugger might also be just used for gaining a better understanding of the functioning of a program. This means that a debugger is not limited to the task of debugging, but may also prove helpful for re-engineering software, or, on a smaller scale, for familiarising with programs written by someone else [Spi03].

2.3.2 Functions of Debuggers

Debuggers generally offer a very typical set of functions [Blu03]. The functions can be differentiated in two groups: Functions that control the execution of a program and suspend it at certain points and functions that allow inspection and possibly manipulation of the program state. The former group comprises the following features:

- Locations in the program source may be marked with *breakpoints*.[15] When the program is executed and the execution reaches a location marked this way, the debugger automatically suspends the execution. This usually happens before the statements in the particular location are executed. When the program is suspended, the debugger can be used to inspect the current internal state of the program, such as the values of variables (see below).

- Some debugger implementations allow breakpoints to be controlled with a further *breakpoint condition*. This condition is a boolean expression using the program variables that are available in the scope of the breakpoint. The breakpoint may be configured to suspend the execution only when the expression is true.

[15] These breakpoints are sometimes also called *code breakpoints* to differentiate them from *data breakpoints*.

- *Watchpoints*[16] can be described as breakpoint conditions that are independent of a certain location in the program source. Watchpoints react on the change of the value of a variable. The debugger suspends the execution if the condition that has been configured with the watchpoint is true.

- If a program is suspended, the debugger allows user-controlled *stepwise program execution*. This means that the user may trigger the execution of the next step in the program. After that step, the execution is automatically suspended again. The user may control the size of the step by choosing between several stepping functions. *Step over* proceeds to the next function call or variable assignment in the current function. It does not step into functions that are called by the current function, though. If the user wishes the debugger to do so, he needs to use *step into function*, which is of course only applicable if the next step is a function call. Finally, *step return* allows the user to execute the rest of the current function in one step and suspend execution before the next step of the calling function is performed.

The functions for *viewing and manipulating the program state* are typically only available when the execution is suspended. As mentioned before, the debugger then allows the user to view the values of variables that are in scope. The scope includes the local variables of the functions that are on the call stack; depending on the type of the language, the scope may also include other reachable variables, such as object variables of the object a method has been called on. Some debuggers also allow browsing of structured data such as objects. If the debugger allows the modification of variables, generally the value of every viewable variable may be also modified. Exceptions may hold for variables of non-primitive types; it depends on the debugger how far those variables may be controlled.

The definition cited before also mentioned *trace facilities* as a typical feature of debuggers. Such a feature records all executed statements in a linear list. The user may use the list later for intellectually reconstructing the execution. Trace facilities are not common to all debuggers, though. For many applications of debuggers, the amount of data that will accumulate during a program execution would make handling such traces very hard.

2.3.3 User Interfaces of Debuggers

The user interface of a debugger may vary. Originally, debuggers came as stand-alone tools which could be controlled by a command-line interface. For editing and viewing the program that was currently debugged, tools needed to be switched. *Integrated Development Environments* (IDE) united the debugger and the source code editor in a single, graphical user interface. Thus, when the program execution is suspended, the debugger visualizes the line that will be executed next in the source code editor. Breakpoints can be set and seen directly in the source code. Selecting a variable in the source code may cause its current value to be displayed.

For getting a better impression of debuggers in IDEs, we will look at an example: A popular IDE for Java programs is the Eclipse Platform equipped with the *Java Development Tools*

[16] Sometimes also called *watch expressions* or *data breakpoints*

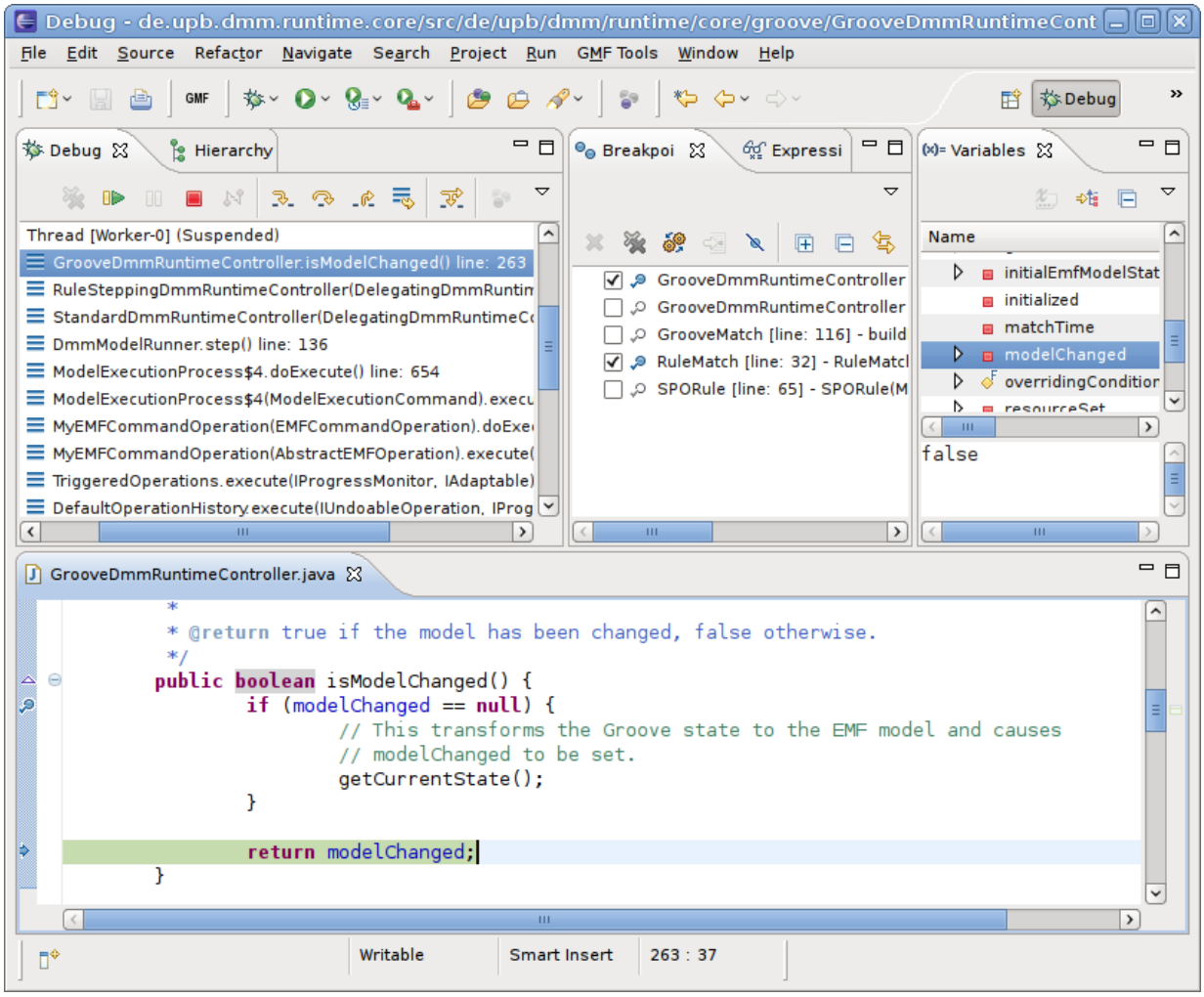

Figure 2.10: Screen shot of the Eclipse Debug view for a Java program

(JDT) [JDT09a].[17] Figure 2.10 shows a screen shot of this IDE while the execution of a program is suspended by the debugger. The upper half of the view shows—from left to right—the Debug view with buttons for stepping and resuming execution and the current call stack of the program, the Breakpoints view with the currently defined breakpoints and the Variables view showing the variables in scope and the value of the selected variable.

The lower half shows the source code in which the next step will be executed. The concrete location of the statement to be executed is marked by a green background and a blue arrow in the left border of the view. Above the blue arrow a blue dot can be seen which signifies the location of a breakpoint.

[17] Due to the plug-in structure of Eclipse, the naming may be slightly confusing: JDT is the name for the plug-in set providing the functionality. The edition of Eclipse providing the Eclipse Platform together with JDT in one package is called *Eclipse IDE for Java Developers*.

2.3.4 Eclipse Platform Debug

The JDT debugger is realized using the Eclipse Platform Debug framework [Ecl09b]. This is a specialized framework for implementing debuggers for arbitrary programming languages. The most important features of the framework are ready-to-use user interfaces which can be connected to the actual debugger implementation by extension points. Thus, all debuggers using this framework provide a consistent user experience while maintaining full flexibility in the actual debugging logic.

2.4 Other Components Relevant to this Thesis

We will conclude the foundations chapter by taking a look at existing software components that are also relevant to this thesis, but which did not fit into the previous sections.

2.4.1 Graphical Modeling Framework

The DMM Player is supposed to display the executed runtime models in their concrete syntax. Editing the models this way shall be supported as well. The "Eclipse way" of creating graphical editors for models is the *Graphical Modeling Framework* (GMF) [GMF09a]. GMF has been specially designed as diagram editing framework for EMF-based models.[18] Thus, it uses the modeling functionality provided by EMF extensively. Besides the actual meta model instance, it also stores the visual structure of the concrete syntax in an EMF-based model.

Furthermore, GMF aims at providing a completely declarative way of implementing diagram editors using models, which are—naturally—also EMF-based models. These models are used to generate Java code which does the actual implementation of the particular editor. Thus, a GMF editor may be also implemented non-declaratively by directly writing Java code that uses the GMF Java API.

The GMF API [GMF09b] makes extensive use of design patterns [GHJV95] in order to reach easily extensible and changeable editors. For example, the *factory method pattern* is used for creating the visual representation objects for model elements. Thus, the visual representation can easily be exchanged. Model editing with undoable and redoable actions is realized using the *command pattern*. The complete behavioural specification of graphical objects, i.e., the way objects react to mouse clicks or to the user trying to create an edge between two nodes, is specified using *decorators*.

A diagram editor realized using GMF benefits from the very rich editing functionality which is provided by the framework. The developer does not need to care about undo/redo support, diagram layout algorithms or export functions to other file formats. All this is automatically provided with each GMF-based editor.

The *UML2Tools* project [U2T09] has used GMF to implement graphical editors for most UML diagram types. This project can be seen as the first endurance test for the concepts

[18] Cf. section 2.2.2 on page 19.

employed by GMF. One of the developers of the UML2Tools summarized the experiences with GMF as follows:

> "Is GMF magic or tragic? It is magic for the people who are sorcerers." [Fes08]

Despite this ambivalent statement, GMF must be regarded as a very powerful framework which is very relevant to all modeling work done in an Eclipse environment.

2.4.2 EProvide

EProvide (Eclipse Plug-in for Prototyping Visual Interpreters and Debuggers) [SW08, SW09] is an Eclipse plug-in that provides a bridge between the program launch and debugging features of Eclipse (partially realized by Eclipse Platform Debug) and further plug-ins that define execution semantics for EMF-based dynamic models. EProvide decouples the actual definition of execution semantics and the method to define the semantics by using two layers:

On the first layer, EProvide allows to configure the *semantics description language*, which provides the base for the actual definition. There are already EProvide plug-ins that are able to define execution semantics with abstract state machines, Prolog, or QVT. DMM would fit right inside this group.

The second layer defines the actual *execution semantics* for a language using one of the semantics description languages defined before. The execution semantics for UML activities would be defined at this layer.

After the user launched the execution of a model, EProvide will repeatedly call the respective plug-ins to perform a single execution step and to update the model accordingly. If the EMF model has been opened with a GMF diagram editor, the editor will automatically visualize the changes. After each step EProvide will wait for a configurable duration; thus, the user will see the changes in the model evolving spread over a period of time.

Recently, EProvide was extended by a generic debugging infrastructure for dynamic models [Blu09, BFS09]. However, this infrastructure makes a couple of assumptions limiting its application area: The debugger assumes that the current execution state can be represented by a single model element from the executed model. Thus, breakpoints of EProvide are always associated with such a single model element. In order to use the EProvide debugger, the execution semantics needs to define a function which returns this model element. During the execution, EProvide will compare the model elements of the breakpoints with the model element returned by the function defined by the execution semantics. If the elements are identical, EProvide will suspend the execution.

Thus, EProvide could facilitate the realization of a simple interpreter for DMM semantics; however, the EProvide debugger cannot be used for a number of reasons: The state representation by one single model element is not transferable to arbitrary languages; already UML activity diagrams represent their state by a number of elements that are distributed over the model. Furthermore, the debugging logic of EProvide is realized at the execution semantics layer, i.e., independent and agnostic of the semantics description layer. However, it is an explicit goal of the DMM Player to realize a generic debugger that can work on DMM rules.

Requirements Analysis 3

The goal of the requirements analysis is to define functional and non-functional requirements to the planned tool from the viewpoint of the user. First, we will look at the processes related to DMM and identify points where the DMM Player can aid these processes. These information will be used in section 3.2 to identify the stakeholders of the DMM Player and analyse their knowledge and skills. Afterwards, section 3.3 will identify requirements that emerge from the DMM approach independently from the used process.

The requirements will then be summed up as use cases in section 3.4. Section 3.5 will list the non-functional requirements of the tool. The specification of actual product functions takes place in the final section 3.6.

3.1 Analysis of Process-Induced Requirements

In order to be able to analyze the requirements for the planned tool, we have to analyze the existing processes which the tool is supposed to aid. Thus, we can identify the concrete points the DMM Player may come into action. This will yield use cases that will be collected later.

3.1.1 Specification of DMM Semantics

We already got to know the specification process for DMM semantics in section 2.1.4. The obvious application area of the planned debugger lies in the action Fix semantics which is triggered when a test case produces an error. However, the paper [SE09] does not provide greater details on the procedures that are due in this case. Thus, we will now try to build up a more detailed scenario on how to cope with test failures.

3.1.1.1 Handling Test Failures

While being represented by the single action Fix semantics in the activity diagram in figure 2.7, the actual task of fixing the reason for a failed test imposes a number of demanding problems on the developer.

1. First of all, the developer should try to find out how the counter-example produced by the failed test violates the CTL expression. Due to the extensive expressiveness of CTL, there is no straight-forward way of finding out how the counter-example violates the CTL expression. However, there are some special cases that can be easily assessed.

   If the CTL expression only consists of a simple sequence of events,[1] one only needs to search for the first event that is present in the sequence, but not present in the counter-example.

   If the CTL expression specifically forbids the occurrence of certain events,[2] the first occurrence of such an event in the counter-example is of course a violation.

   The other special case is the *DMM specification failure*[3] which indicates that a DMM rule tried to perform an invocation that could not be handled by any other existing rule. This failure can be recognized by a special GROOVE rule and a trivial CTL expression checking for that rule. Thus, no further analysis on the reason for the CTL violation is needed in this case.

   If the violation of the CTL expression cannot be found quickly, this step may be skipped aspiring that the actual cause of the error will be found in the next step. Skipping this step is of course also reasonable if the cause for the failure is already obvious by other evidence.

2. Even if the point that violated the CTL expression could be located in the counter-example, this is in most cases not the root cause of the failure. The only case in which this point is indeed the root cause is a transformation rule whose left-hand side is too general, i.e., it matches in cases it is not supposed to.

   Otherwise, the reason for the failure can be found in the counter-example at some place before the located point. Thus, the developer should manually check the states and transitions between that point and the beginning of the counter-example. If that root cause is not immediately obvious, it is advisable that the developer reviews the states from the beginning in their original execution order. Reviewing the states in the reverse order from the located violation point may seem to be quicker at the first sight; however, comprehending the transitions in reverse order is significantly harder than doing it in the "natural" order. Furthermore, one needs to ensure that any located error is not caused by a preceding error.

   This review of the counter-example could be aided by the planned tool. For instance, the linear sequence of states the counter example consists of could be visualised in the concrete syntax of the language. This may reduce the semantic gap and thus help the developer in finding the source of the error.

3. When the root cause has been found, the next step is, obviously, the correction of the erroneous rule.

[1] In CTL terms, this is a chain of **EF** expressions.
[2] This can be reached by an **AG** CTL expression with a negated predicate.
[3] Cf. section 2.1.3 on page 11.

If the root cause of the failure did not directly cause the test to fail, but started a chain of sub-failures that finally led to the violation of the CTL expression, the developer should also consider creating a new test case that is able to discover the actual root cause.

The Fix semantics action is concluded by this step. The semantics specification process now requires a re-run of all tests to ensure that the modifications that have been performed are indeed correct.

3.1.1.2 Manual Review

It should be noted that the test-driven process is not able to guarantee the development of absolutely correct DMM semantics. The process just ensures that the DMM semantics is correct with respect to the test cases and the CTL expressions which specify the expected behaviour. Errors in the semantics may go unnoticed because of two primary reasons:

- A certain scenario that is handled incorrectly by the semantics is not covered by the example models.
- There is an example model covering the erroneous part of the semantics; however, the corresponding CTL expression is insufficient and does not recognize the wrong behaviour.

The first reason can only be handled by specialized measures during the development process; this is not the scope of this thesis and will not be covered any further.

Errors that go unnoticed for the second reason during the automated tests might be however noticed by a manual inspection of the actual behaviour exposed by the execution of the example models using the developed DMM semantics. Thus, the developer should perform such a manual review at least once for each example model. Currently, the only tool support for this task is GROOVE which allows the developer to manually step through the transition system of an example model and view the respective states in the GROOVE representation of the abstract syntax of the example model.

An important characteristic of such manual reviews is that the developer needs to keep a comprehensive overview over the states as he does not actually know what he is looking for. Due to the verbose visualisation of the model states performed by GROOVE this is very hard if not even impossible. The language's concrete syntax is more suited for such an overview; thus, the review of the model states in the language's concrete syntax can be regarded as a further use case for the planned tool.

When an error has been found that was not noticed by the tests, the erroneous rules must be corrected. However, before performing the actual correction, it is advisable to create a new test case that actually discovers the error. In some cases refining an existing test case may also be sufficient. The new or extended test case may now be used in order to verify that the modifications that have been performed actually do correct the semantics. Obviously, regressions that may be caused by subsequent modifications to the semantics will be also discovered this way.

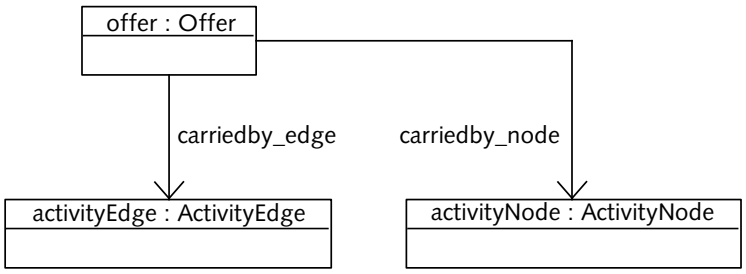

Figure 3.1: Example of an semantically invalid runtime meta model instance

3.1.1.3 Limitations of Test Cases

The given tooling for the test-driven development of DMM specifications has one important limitation which should be noted at this place. The CTL expressions can only check the sequence of events, i.e., the application of DMM rules. However, it is not possible to examine the states of the transition system for validity or other properties.

Still, checking the validity of a state is a valid requirement, as the example model state in figure 3.1 demonstrates. It is an instance of the runtime meta model of the DMM UML activity semantics. The instance is semantically invalid, because a token may only be linked to either an edge or a node. Still, without further measures, the meta model allows such instances. When the instance is represented using EMF, tools such as the EMF validation framework may be used to impose further constraints on the meta model in order to discover such invalid states. However, those constraints are currently lost when the EMF-represented model instances are transformed to GROOVE states. Thus, as long as the instances are represented as GROOVE states, they may become semantically invalid without being noticed.

There are some special cases in which the GROOVE state is examined to a limited extent: The *DMM specification failures*[4] are checked using a special GROOVE rule with equal left hand side and right hand side and a CTL expression watching matches of that rule. Because its left and right hand side are equal, the execution semantics is not influenced by the rule. The DMM2GROOVE transformation tool automatically adds the rule watching for DMM specification failures to each GROOVE grammar transformed from a DMM rule set. The rule itself is not transformed from a DMM representation, but is directly written in GROOVE.

Furthermore, using so-called *emphasized attributes* inside the DMM rules, attributes in the states of the transition system can be bound to parameters inside the CTL expressions. Thus, the CTL expression can check for the transformations in which objects with certain names, IDs or other attributes are involved. However, there is no other way to assess the states for validity.

In order to cope with this problem, DMM could introduce support for user defined rules which check the model states for certain structures. We will call such rules with an equal left and right hand side checking for state properties *property rules*. Section 4.1 will provide a deeper introduction into property rules later.

Property rules may be used both in the test cases and in the manual reviews to highlight properties of interest.

[4]Cf. section 2.1.3 on page 11.

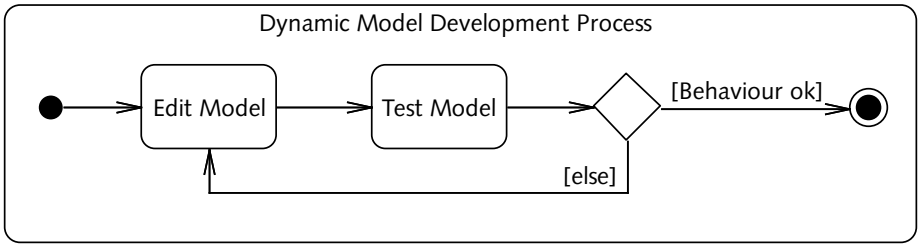

Figure 3.2: A very basic development process for dynamic models

3.1.2 Development of Dynamic Models

Besides the development of DMM semantics, the second application area of the DMM Player is the development of dynamic models using modeling languages for which DMM-specified execution semantics are already available. An example for such a language is, of course, UML with its activity diagrams.

As the DMM Player is supposed to be generic, i.e., usable with any language with DMM-specified semantics, we cannot identify all potential processes in which the tool will be later employed. Even if we limit this process analysis to UML activities, a multitude of processes is applicable, because UML activities have been designed to be a generic concept usable for arbitrary tasks.

Thus, for now, we will just consider a very basic and generic development process, a simple iterative process shown in figure 3.2. Consequently, the resulting tool will provide a generic functionality, which matches the approach taken by DMM.[5]

3.2 Stakeholder Analysis

The stakeholder analysis is meant to find out what people are concerned by the introduction of the new software. Furthermore, some background information useful for requirements analysis about those people should be retrieved. Knowledge and skills of the users are of particular interest here.

The specification process for DMM semantics[6] does not provide any information about cooperative work during the specification of DMM semantics or about any roles involved within of it. Thus, we will assume one single role, the *DMM semantics developer*.

People assuming this role can be expected to have knowledge about the concepts of DMM, particularly about DMM rules and runtime meta models, as this knowledge is essential for the task. Because the current tool support uses Ecore models to specify the runtime meta models, a basic knowledge about EMF can be expected as well. A further essential prerequisite is knowledge about the language whose semantics is to be specified using DMM. The formalization of the execution events for the test cases requires knowledge in CTL.

[5]Still, it should be taken into account that specialized functionality might become necessary in the future. So, special care should be taken to develop an easily extensible software product.

[6]Cf. section 2.1.4 on page 14.

People might have experience in using "classic" debuggers for text-based programming languages or other tools of integrated development environments.

For the development of dynamic models, we will also assume the most simple case for the identification of the stakeholders: The process involves a single role, the *model developer*.

People assuming this role will have knowledge about the particular language they are going to use. However, they cannot be expected to have any knowledge about DMM. Knowledge about the meta model of the particular language may be present, but cannot be expected to be full-fledged.

Furthermore, those people might also have experience in using "classic" debuggers or other tools of integrated development environments.

3.3 Analysis of Method-Induced Requirements

The methods employed by the processes and by the planned tool may induce further requirements. In the following sections, we will look at those requirements.

3.3.1 Execution Paths

We have already seen in the UML activity example in section 2.1.5 that the execution of a model does not need to be deterministic. In a transition system, this is indicated by multiple transitions going out from a node. Such a constellation is also called a fork.[7] Transition systems notated as graphs use spatial dimensions for denoting transitions. Representing a fork in a graph is no significant problem. However, the DMM Player will perform the transitions along the time, a dimension in which forks cannot be regarded as user-friendly. Thus, it is necessary to choose a single path through a transition system for execution.

An obvious solution is to let the user choose the path to be executed. As decision nodes in a UML activity diagram may represent decisions external to the model which cannot be expressed in the model state, a user-controllable choice of the execution path seems to be very sensible.

However, external decisions are not the only concept that may cause forks in a transition system. One example for another cause of forks is the use of concurrency in UML activity diagrams—which may be caused by fork nodes in the diagram. Figure 3.3 shows a simple example of a UML activity diagram using concurrency. Applying the DMM semantics for UML activities on this diagram results in the transition system shown in figure 3.4.[8]

As the activity diagram allows the actions a, b and c to be executed in any order, the transition system contains paths for each possible order. Further forks are caused by the the offers

[7] A fork in a transition system should not be confused with a fork in a modeling language, such as UML activity diagrams. While forks in activity diagrams may cause a fork in a transition system, this is neither required nor sufficient.

[8] The transition system has been shortened to show only states with either multiple incoming or outgoing transitions. The transitions which thus comprise more than one DMM rule have been marked with the label [...].

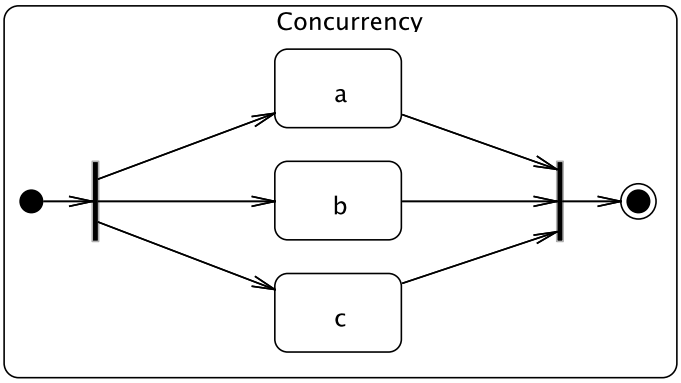

Figure 3.3: Simple UML activity diagram using concurrency

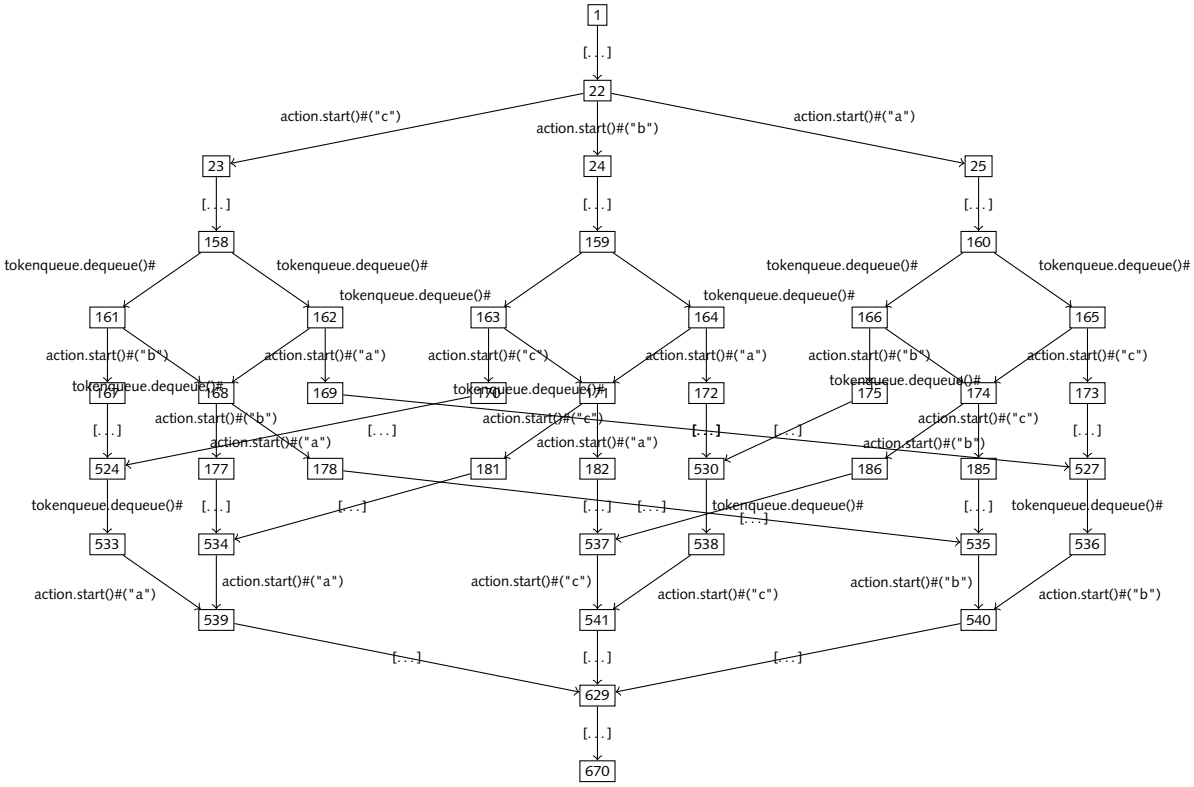

Figure 3.4: The transition system of the UML activity in figure 3.3

flowing concurrently towards the actions before the actions are actually executed. The tokenqueue.dequeue()# rules are responsible for this behaviour.

Thus, when executing this model, up to four decisions need to be made on the path to be used. This may still be manageable for the user; however, with growing models, this may quickly get out of hand. Here, an automatically performed choice may be better.

It would be desirable to control in some way whether the user is requested to choose the path to be used. Optimally, this will be contextually dependent on the kind of fork; with such a solution, it would be for example possible to request a path choice at forks caused by choice nodes and to automatically choose a path at other forks.

3.3.2 Debugger Concepts for DMM

In section 2.3, we got to know the concepts of debuggers for classical programming languages. We saw that breakpoints and—to some extent—watchpoints are the key concepts of debuggers, as they provide the means to find the execution states of a program the user is interested in. When a break- or watchpoint has suspended the execution of a program, the user can examine the particular execution state using the further functionality provided by the debugger.

In order to be able to define the requirements for a debugger for DMM, we need to transfer these concepts from the world of classical programming languages to DMM. Actually, we need to do a transfer to two different domains, as the DMM Player is supposed to both serve as a debugger for DMM semantics and for models developed in languages defined with a DMM semantics specification.

3.3.2.1 Breakpoints for DMM Semantics

We will begin with the transfer to a debugger for DMM semantics. Classic debuggers put breakpoints on instructions; when the execution of the particular program reaches that instruction, the debugger will suspend the execution before the instruction has been executed. What could be the DMM equivalent of an instruction?

An obvious analogy to an instruction from a classical programming language is a DMM rule. It may modify the state and invoke other rules. Thus, setting breakpoints on rules seems to be a very reasonable feature of a DMM debugger. Just like classic debuggers, the DMM debugger could suspend the execution before the rule marked with a breakpoint will be executed. However, in contrast to classical programming languages, the rule that will be executed next is not deterministically defined by the program state. Rather, several rules may be applicable, i.e., executable; due to different matches, a rule may be even applicable in different ways at the same time.

As described before, the DMM Player will choose only one rule that is going to be executed, though. Thus, breakpoints which suspend the execution before a particular rule is going to be executed cannot handle the indeterministic character of DMM rules. A further kind of breakpoint can solve this problem: A breakpoint that suspends when its attached rule becomes applicable, i.e., its left hand side matches the current state. We will call this kind of breakpoint

a *match breakpoint*,[9] opposed to the *before application breakpoint* described before.

Conditions limiting the applicability of breakpoints may be useful for DMM debugging as well. A scenario in which a rule breakpoint with a condition is feasible could be the execution of a certain action in an activity diagram. In this scenario, the user could create a rule breakpoint for the rule action.start()#. A further condition may check that the name attribute of the action node in the rule has a certain value.

3.3.2.2 Watchpoints for DMM Semantics

Watchpoints have been described in section 2.3.2 as boolean conditions using variables from the program state that cause the program execution to be suspended when they evaluate to true. In contrast to conditions of breakpoints, watchpoints are not tied to a certain execution event. Rather, watchpoints monitor the program state constantly. For classic programs, the program state is represented by variables and complex data structures stored on the stack or in the heap. The DMM representation of that is just the state of the runtime model.

The ideal way of checking the runtime model for certain conditions or properties are the already mentioned property rules.[10] As explained, the use of property rules is to recognise states with certain properties that are specified in their respective patterns. Thus property rules can be used—just like classic watchpoints—for monitoring parts of the program state.

In order to define a watchpoint for a DMM runtime model, the user may formulate the property to be watched using a property rule and use a match breakpoint on that rule to suspend the execution when a state with the particular property occurs. Thus, for debugging DMM semantics, watchpoints are a special case of rule breakpoints using property rules.

A more comprehensive introduction in property rules will be later given in section 4.1.

3.3.2.3 Advanced Features of a Debugger for DMM Semantics

We already learned about match breakpoints. Supporting this kind of breakpoints makes further functionality desirable, though. A match breakpoint enables the user to see that a particular rule is applicable. In this situation, he may also want to make sure that the rule is actually applied to the current state rather than another rule that is also applicable. The execution path configuration which was already alluded to in section 3.3.1 is not suitable for this purpose, as it is supposed to be usable without the knowledge of rules.

Rather, a user interface similar to the UI employed by GROOVE for the manual exploration of a graph grammar could be used to give the user control over the next rule to be executed. Figure 3.5 shows a screen shot of this UI; applicable rules are action.start_3 and decisionnode.flow_1[11] which can be applied for two different matches. Clicking on one of the match entries would visualise the matched elements of the graph; double clicking would apply the particular rule and activate the next state. This, however, leaves the scope of this thesis, as the

[9] Cf. appendix E for the complete tennis scoring rules adapted to DMM.

[10] Cf. section 3.1.1.3 on page 30.

[11] The numerical suffixes of the rule names which can be seen here arise from the fact that GROOVE requires rule names to be unique. Thus, the DMM2GROOVE transformation adds those suffixes to the rule names.

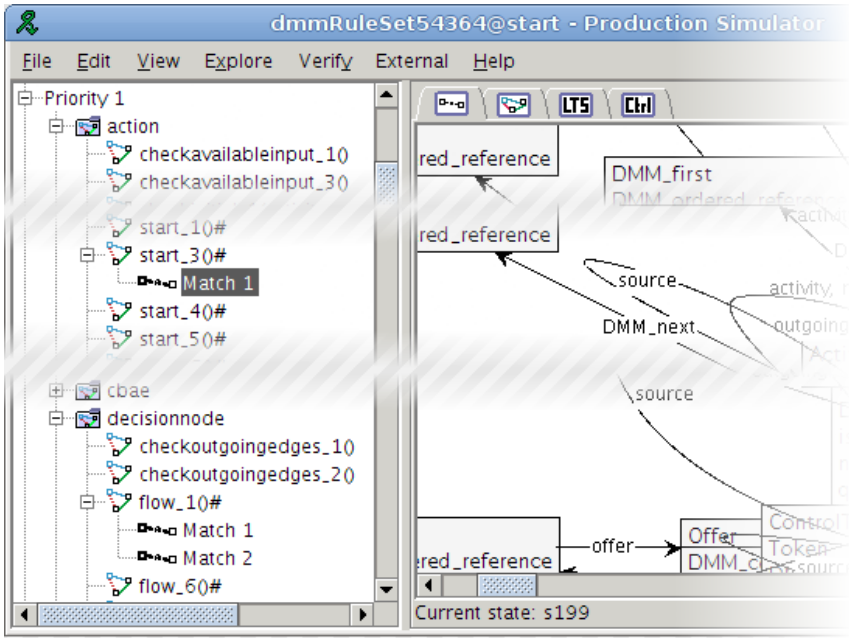

Figure 3.5: Details of a screen shot of the program GROOVE showing several applicable matches

planned tool would thus converge to a universal DMM front-end for GROOVE. The functionality is still a reasonable desire. Thus, a future extension of the DMM Player by such a feature may be sensible.

A further problem that arises in the development of specifications using graph transformation rules is that rules that are supposed to match a certain state, do not actually match. The concepts we have seen so far do not help in this case, as they all depend on existing matches. It has been proposed in [SK08] to aid the debugging of graph transformation rules by visualising *partial matched rules*, i.e., rules that match the current state with a substantial number of nodes, but not with all nodes that are present in the left hand side of the rule. Such a visualisation would be both applicable and helpful to the debugging of DMM rules. This is however also beyond the scope of this thesis and will not be further explored at this place. Furthermore, the current DMM implementation depends on GROOVE to perform the graph transformation. At least the current version of GROOVE does not support finding partial matches.

3.3.2.4 Breakpoints for Models with DMM Semantics

We will now move on to transferring debugger concepts to the development of actual models for which a complete DMM semantics specification already exists. The key differences of this domain to the domain of debugging DMM semantics are the knowledge of the user and his goals.

The stakeholder analysis[12] showed that a model developer does not necessarily need knowl-

[12]Cf. section 3.2 on page 31.

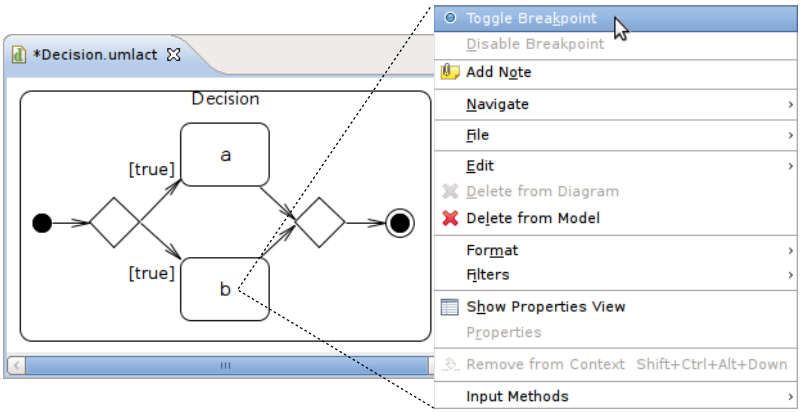

Figure 3.6: Setting a concrete syntax breakpoint on an action of an activity diagram

edge about the underlying DMM rules and the runtime meta model. Thus, a user without such knowledge cannot use the rule breakpoints as described before. A desirable solution would be to let the user specify breakpoints in the concrete syntax of a model. Thus, we will call this kind of breakpoints *concrete syntax breakpoints*.

Figure 3.6 shows an example of how that could look like. In this example, the user wants to set a breakpoint in the action b. However, it is not yet clear, when that breakpoint should suspend the execution. Feasible moments would be when an offer or a token arrives at the action. Generally, the moments depend on the particular language, as UML activity diagrams in this case. Thus, in order to support such breakpoints, a DMM semantics specification needs to define breakpointable situations besides the actual operational semantics.

These situations may be defined with property rules with a node that acts as a kind of template parameter. When the user sets a breakpoint on a model element supported by a property rule defined for this purpose, the particular model element could be bound to the template parameter. Thus, a new, more specific property rule arises which could be internally used as rule breakpoint. For UML activities, such a property rule could use the a node with class ActivityNode as template parameter. This node would be linked to a node representing a token. Thus, concrete syntax breakpoints could be defined for the situation in which a token is attached to a certain activity node.

However, this should be considered as a low priority feature as this is only required if the user has no knowledge about the meta model underlying the language he is using. If he has, he may use normal property rules for this purpose. In order to keep the scale of this thesis on a reasonable level, this feature will not be followed further.

3.4 Use Cases

After having gathered information about the present processes and structures, we can now move on to identifying ways of supporting these processes in the planned tool. This section formulates these ways as use cases [JCJO92]. Each use case describes a single goal of a user and how it can be achieved using the tool. However, use cases do not yet make any stipulations regarding the user interface of the tool.

For clarity, the use cases are presented here in two categories: Essential use cases, i.e., use cases that result directly from the goals of this thesis or from the processes described before. Administrative use cases comprise goals that emerge from the planned tool itself, i.e., those use cases would not exist without the tool.

Inside the sections representing the categories, the use cases are ordered roughly from general to specific use cases. Note that the following use case specifications will omit the documentation of a scenario and/or trigger if these are trivial; an example of a trivial trigger would be the trigger *The user wants to edit a model* for the use case Edit runtime model in concrete syntax. Likewise, the scenario of this use case would be *The user edits the runtime model until he is finished*.

The actors in the use case models correspond to the stakeholders identified in section 3.2. There is one additional actor, the *DMM Player user* which is a generalization of all other actors. This actor is required in order to describe use cases that are common to the other actors.

3.4.1 Essential Use Cases

Figure 3.7 shows the use case diagram of the essential use cases. The single use cases are described in the following sections.

3.4.1.1 View Execution

This is the central use case of the DMM Player. The user watches the stepwise execution of a model in its concrete syntax. Possibly, there is more than one way of executing the model, i.e., there are forks in the transition system of the model. In that case, the user should be able to configure the execution path to be used.[13]

This is a general use case, that can occur in various contexts. There are a couple of other use cases that include the use case View execution in specific contexts.

User: DMM Player user

Preconditions: The user has chosen a model to be executed. A DMM specification for that model type must be available.

Postconditions: None.

Triggers: The user initiated model execution using the UI.

[13] Cf. section 3.3.1 on page 32.

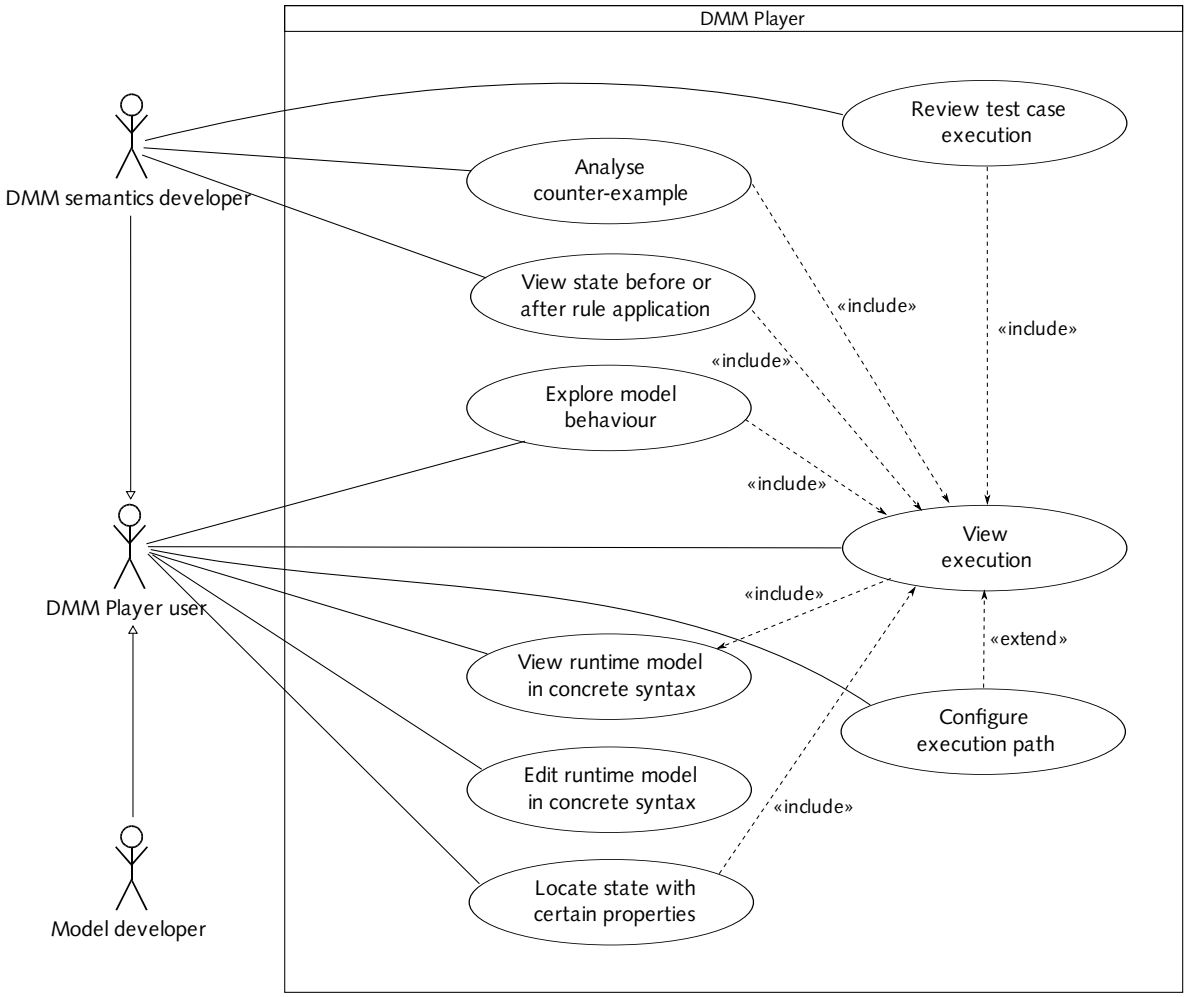

Figure 3.7: Use case diagram describing the essential use cases of the DMM Player

Primary scenario:[14] Execute model transformations until no further transformations are possible.

Alternative case; multiple execution paths: Execution is currently at a fork in the transition system. The user will be prompted to choose the path to be used. Afterwards, continue normal execution.

Alternative case; execution suspended: The execution was suspended. The DMM Player waits for the user to continue the execution by a certain UI interaction. The execution may be suspended for these reasons:

- The condition of a breakpoint was met.
- The DMM Player is configured to suspend execution after each step.
- The execution was manually suspended by the user.

3.4.1.2 Configure Execution Path

In this use case, the user configures the behaviour of the model execution at forks in the transition system. The user needs to choose which path of a fork will be used during the execution. This use case can occur during the execution of a model (represented by the use case View execution) when the execution reaches a fork. However, the use case can also occur independently in order to prepare the execution path configuration for any future model executions.

User: DMM Player user

Preconditions: The user has chosen a dynamic model.

Postconditions: The execution path configuration for the model has been stored.

Triggers: The user initiated execution path configuration using the UI; *Or:* During a model execution a fork in the transition system was found.

Notes: The selection of execution path choices requires some further conceptual considerations which will be made in section 4.2.

3.4.1.3 View Runtime Model in Concrete Syntax

This is a very basic, yet essential use case. The goal of the user is to view the state of a runtime model in its concrete syntax in order to get a better overview of the model than it would be possible when viewing it in other syntaxes. The model may stem from arbitrary sources; it may have been generated or given to the user by a third person. The use case View execution includes this use case, as the tool will visualize the current model state after each execution step. The goal of that use case implies, that the user will be interested in viewing at least one of the model states reached during the execution.

[14] For this use case, I chose not to use the common scenario format of a list of numbered steps because it would significantly complicate the scenario structure without providing information relevant for requirements analysis.

User: DMM Player user

Preconditions: The user has chosen a dynamic model. A specification for the concrete syntax of the corresponding model type must be available.

Postconditions: None.

Triggers: The user opens a dynamic model; *Or:* The use case View execution yielded a new model state.

3.4.1.4 Edit Runtime Model in Concrete Syntax

Besides viewing a runtime model in its concrete syntax, editing it is of course also a valid use case. As a stand-alone use case, the user may want to edit the model at any time for various reasons. This use case can extend the View execution use case when the execution is suspended; the user may chose to edit the model in this case to correct obviously incorrect model states in order to continue an execution without needing to correct the reason for the incorrect state before. Another reason may be exploring the behaviour of a model under special conditions.

User: DMM Player user

Preconditions: The user has chosen a dynamic model. A specification for the concrete syntax of the corresponding model type must be available.

Postconditions: The model has been altered according to the user's actions.

Notes: The concrete syntax of a dynamic model integrates static elements representing the static behaviour specification of the model and dynamic elements representing the current state of the model. This use case only comprises modifications to dynamic elements. Modifications to the static elements should be already possible using the software components the DMM Player will be based on. Technically, such modifications are also possible during a suspended model execution. The advisability of such is questionable, though.

3.4.1.5 Explore Model Behaviour

This is a general use case representing the user goal of "getting to know a model's behaviour better". It may comprise one or more model executions, possibly with intermittent modifications to the model or the execution path configuration.

User: DMM Player user

Preconditions: The user has chosen a dynamic model to be explored. A DMM specification for that model type must be available.

Postconditions: None.

3.4.1.6 Locate State with Certain Properties

During model execution, the user may want to view the current model state when it has certain properties. Thus, when the model is executed continuously, the DMM Player should be able to suspend the execution in such cases.

User: DMM Player user

Preconditions: The user has chosen a model to be executed. A DMM specification for that model type must be available.

Postconditions: The execution was suspended at the designated point.

Primary scenario:

1. Specify the properties to watch for using a property rule or choose an already existing property rule.
2. Suspend execution when a state with these properties occurs.

3.4.1.7 View State Before or After Rule Application

In order to analyse the behaviour of a certain DMM rule, the user may want to take an extensive look at the model state before and/or after the rule has been applied to it. In the case of continuous model execution, the DMM Player should be able to suspend the execution at such points.

User: DMM semantics developer

Preconditions: The user has chosen a model to be executed. A DMM specification for that model type must be available.

Postconditions: The execution was suspended at the designated point.

Primary scenario:

1. Choose the rules and whether the suspension should occur before or after the rule has been applied.
2. Suspend execution when the condition is fulfilled.

3.4.1.8 Analyse Counter-Example

When a CTL-based test for a DMM specification fails and provides a counter-example,[15] the user will want to analyse that counter-example in order to find the reason for the test failure. In order to achieve this, he may want to view the stream of model states that forms the counter-example just like a normal model execution. Thus, this use case includes the use case View execution. Because a counter-example is a linear sequence of states, there will not be any execution forks and thus the use case Configure execution path will never be triggered.

[15] Cf. section 3.1.1.1 on page 27.

User: DMM semantics developer

Preconditions: A counter-example of a failed test is available.

Postconditions: None.

Triggers: A CTL-based test for a DMM specification fails.

Notes: This use case requires integration of the DMM Player with a test framework for DMM. As no generic implementation of such a framework is available yet, the first version of the DMM Player will not support this feature.

3.4.1.9 Review Test Case Execution

This use case corresponds to the manual review of test cases described before in section 3.1.1.2. The user's goal is to observe the model states during the execution of the test case and watch for errors that are not noticed by the CTL expressions accompanying the test case.

This use case includes the use case View execution. If there are forks in the transition system, the user may want to run several executions in order to cover all possible paths.

User: DMM semantics developer

Preconditions: A test case for a DMM semantics has been chosen.

Postconditions: None.

Primary scenario:

1. Watch model execution.
2. Change the execution path configuration[16] and restart until all—or all interesting—execution paths of the model have been covered.

Alternative case; error discovered:

1a. If the user discovered an error:
   a) Extend tests in order to be able to catch the error.
   b) Fix the error.
   c) Start again with the manual review of the test case.

Notes: For this thesis, no automated support for covering all possible paths in the transition system is planned. Thus, the user has to manually choose the paths to execute. Possibilities for automatically iterating through the possible paths maybe desirable, though.

[16] Cf. section 3.4.1.2 on page 40.

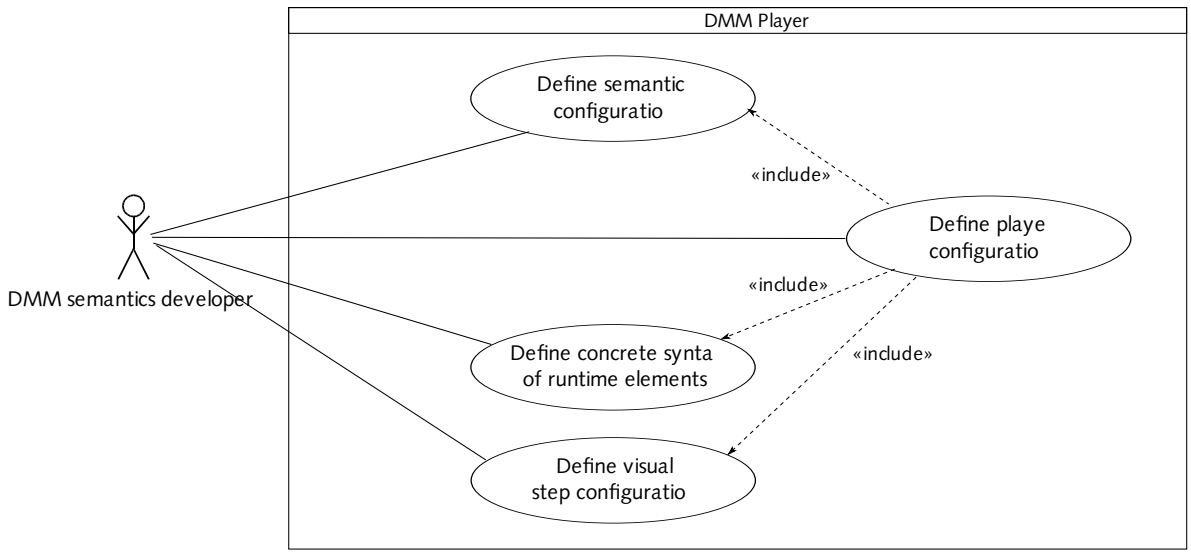

Figure 3.8: Use case diagram describing the administrative use cases of the DMM Player

3.4.2 Administrative Use Cases

Administrative use cases comprise goals that emerge from the existence of the DMM Player. Without the tool, these use cases would not exist. The use case diagram of the administrative use cases is depicted in figure 3.8.

3.4.2.1 Define Player Configuration

This is the general use case for defining the configuration of the DMM Player for a model type with DMM-specified semantics. It includes all other administrative use cases which represent the definition of independent parts of the configuration.

User: DMM semantics developer

Preconditions: A DMM semantics is available.

Postconditions: The configuration has been created; The DMM Player can now be used with the model type.

3.4.2.2 Define Semantic Configuration

This is the most important administrative use case, as this configuration enables the DMM Player to actually execute a model. This configuration associates DMM rule sets and runtime model initialization code with model types. Thus, if there is such a configuration for the type of a certain model, the DMM Player will be able to execute it.

User: DMM semantics developer

Preconditions: A DMM semantics is available.

Postconditions: The configuration has been created; The DMM Player can now be used with the model type.

3.4.2.3 Define Concrete Syntax of Runtime Elements

Just like the runtime meta model extends the static meta model of an existing modeling language, the concept of the DMM Player also stipulates that it should be possible to extend existing visual editors for the concrete syntax of the modeling language by a concrete syntax for the runtime data.

The user goal represented by this use case is to define concrete syntax representations for data in a runtime model.

User: DMM semantics developer

Preconditions: A DMM semantics is available; a visual editor based on the static meta model is available.

Postconditions: The existing editor is extended in such a way that elements from the runtime meta model can be displayed and edited in a concrete syntax.

3.4.2.4 Define Visual Steps

Normally, the model state resulting from the application of a DMM rule on the previous model state will immediately be made visible to the user by the DMM Player. However, there are some cases in which certain model states should not be visible to the user. Rules—especially small step rules—may cause temporal inconsistencies in the model state that get fixed by rules that are applied afterwards. A user of the DMM Player that is not supposed to know the internals of a particular semantics specification should not be bothered by those temporal inconsistencies which are purely implementation dependant. Thus, it should be possible to define which rules produce model states that may be viewed by the user.

User: DMM semantics developer

Preconditions: A DMM semantics is available

Postconditions: The DMM Player only displays states according to the step unit configuration.

Notes: The conceptual analysis of this feature is provided in section 4.3.

3.5 Non-functional Requirements

The use cases—or more general: functional requirements—are complemented by non-functional requirements. Non-functional requirements leave behind the question for "*what needs* to be done" and ask "*how* it *needs* to be done".[17] Such requirements often refer to qualities relevant to the end-user or the software developer. Technical conventions such as components or algorithms to be used may also be stated as non-functional requirements. Generally such a requirement is made in order to improve again a certain quality of the software.

The following sections describe the non-functional requirements of the DMM Player.

3.5.1 Use of Existing Components

As already stated in the goals section of this thesis, the use of existing software components should be strongly preferred over creating new components. By using well matured components, the implementation and maintenance efforts for the software can be reduced, even if some adaption to the existing components is necessary. Furthermore, modularisation and complexity of the code will improve because of strict component boundaries.

The DMM Player shall use these components for its tasks:

- The *Eclipse Platform*[18] as its base implementation platform, as the whole existing DMM implementation bases on it.
- The *Eclipse Modeling Framework* (EMF)[19] shall be used for management and representation of the models that the DMM Player will use. This is essential, as the existing DMM tooling already uses EMF for storing rule sets and model instances. Furthermore, EMF should be also used when defining new model types in support of the DMM Player. An example for those model types may be models storing the definition of the concrete syntax for a runtime meta model.
- The *Graphical Modeling Framework* (GMF)[20] should be used for rendering the concrete syntax of the dynamic models.
- *EProvide*[21] can be used to associate EMF models with their respective DMM execution semantics and to actually launch code that interprets the model with the execution semantics on user interactions in the Eclipse UI. Its debugger implementation cannot be used, though. Recently, EProvide was
- The Eclipse *Platform Debug* framework[22] for the realization of the debugging features.

[17] Note the emphasis on *"needs"*. This question is limited to explicit requirements. If the way of doing something is not stated in the requirements, it will be defined during a later phase, such as analysis and design.
[18] Cf. section 2.2.1 on page 19.
[19] Cf. section 2.2.2 on page 19.
[20] Cf. section 2.4.1 on page 25.
[21] Cf. section 2.4.2 on page 26.
[22] Cf. section 2.3.4 on page 25.

- GROOVE[23] shall be used for performing the actual graph transformations required for the DMM execution. Contrary to the current usage of GROOVE in DMM, it should be completely hidden to the user; the user should neither care about transformation of rule sets or models into GROOVE graphs, nor any of those transformed models should be visible to the user.

3.5.2 Conformance to Known Debugging Tools

The UI of the DMM Player should be as similar as possible to the debugging user interfaces of common integrated development environments. This will facilitate the user's familiarisation with the new tool. By using the Eclipse Platform Debug framework (as stated in the previous section), this requirement is already partially fulfilled.

3.5.3 Execution Speed of Dynamic Models

The execution of dynamic models will be performed stepwise with a short pause between the steps in order to allow the user to recognize the model changes caused by the previous step. The execution of such a step should never take longer than that predefined length of the pause. Otherwise the visualization of the execution will appear jerky. Pauses that take too long may impair the concentration and patience of the user.

3.6 Functional Specification

After having gathered the knowledge of what the user needs, the functional specification now defines how a product that satisfies the user's needs will behave and look like. This view is still limited to the user interface, questions that dig deeper will be handled later.

Most of the user interfaces are not defined by the DMM Player itself. Instead it uses UIs of the frameworks or components described before.

3.6.1 Run or Debug a Model

Eclipse provides a standard user interface for running or debugging executable resources. At the topmost level, the UI consists of functions grouped into one *run* and one *debug* category. These functions are provided in the Run pull-down menu (see figure 3.9), in the context menu of runnable resources and as toolbar elements. The run and debug categories provide the same set of menu items which perform their functions either in normal or in debug mode.[24] *Normal mode* means that the resource is simply executed without providing a debugging functionality during the execution. The *debug mode* enables features such as breakpoints or manual stepping through the execution.

[23] Cf. section 2.2.3 on page 20.

[24] The basic functions that are available in the menu are documented in the JDT user guide [JDT09b] at http://help.eclipse.org/galileo/topic/org.eclipse.jdt.doc.user/reference/menus/run/ref-menu-run.htm

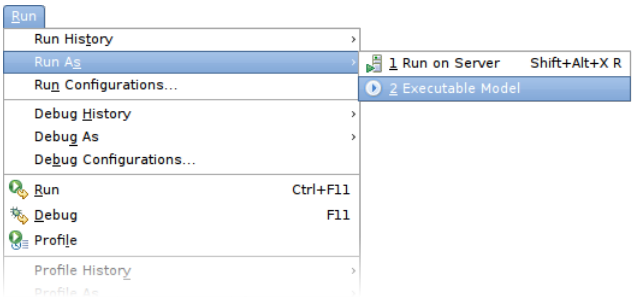

Figure 3.9: Detail of a screen shot showing the upper part of the Run pull-down menu of Eclipse

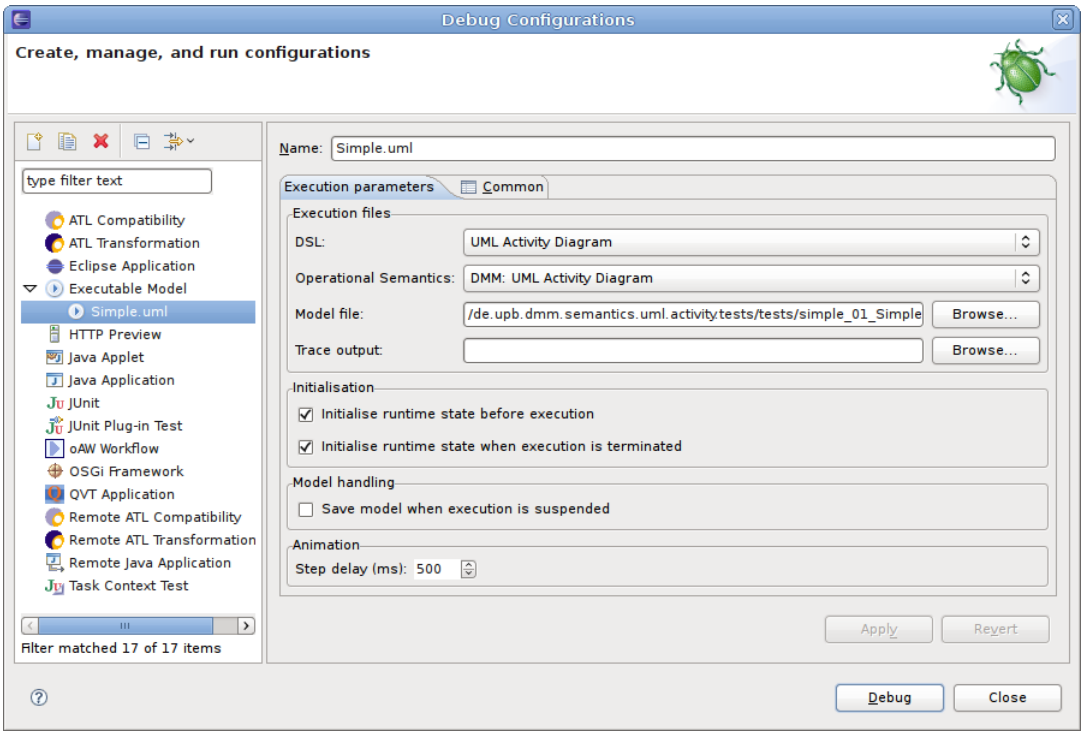

Figure 3.10: Screen shot of the Launch Configurations dialog of Eclipse

Furthermore, the EProvide plug-in extends these menus by the functions Run As Executable Model and Debug As Executable Model. These functions initiate the execution of the model that is currently selected within the Eclipse workbench. This is achieved using further plug-ins and semantics that have been configured with EProvide. The DMM Player will provide such a plug-in in order to use this UI to launch a model execution.

Using the Run As Executable Model and Debug As Executable Model menu items, the execution will be performed with a default configuration. The menu items Run Configurations and Debug Configurations provide means to specify a custom execution configuration. Using one of these menu items will open the Launch configurations dialog which is shown in figure 3.10. The dialog is provided by the Eclipse platform and customized by the EProvide plug-in for providing the model execution parameters properties page, which can be seen in the right part of the screen shot.

In this dialog, the user may create a new launch configuration with the easily overlooked button in the upper left showing an empty page with a plus symbol. Then, he can choose the semantics to be used. As EProvide is designed to allow several semantics description languages for one language, this requires two configurations. The field labelled with DSL allows the definition of the actual language of the model to be executed; even though DSL stands for domain specific language, EProvide can also be used for general purpose languages, as there is no difference in handling the execution of those languages. The field Operational Semantics depends on the selection in the DSL field and shows the actual semantics definitions in the different available semantics description languages. When EProvide is only used for DMM, this field will always contain exactly one entry for the particular language. The field Model file is the last required input field. It specifies the XMI file which contains the model to be executed.

In order to let the user visually follow the execution, there will be a delay between each step. The field Step delay can be used to specify this delay. The remaining fields are special features of EProvide which will be for now not supported by the DMM Player.

When the execution is started with one of the Run As Executable Model and Debug As Executable Model menu items or the Run or Debug button in the Launch configurations dialog, the execution of the selected model will start; an opened model editor will display the stepwise execution of the model.

3.6.2 Execution Path Selection

In order to provide support for the use case Configure Execution Path (cf. section 3.4.1.2), the DMM Player will suspend the execution of a model as soon as a transition system fork occurs that is supposed to require path selection.[25] The tool will then visualize the possible path choices by annotating model elements identify the path with gray colored "straight forward" traffic signs, as can be see in the upper half of figure 3.11. The user can then open the context menu on one of these elements. The menu will contain the menu item Execution Path Selection which provides a number of configuration possibilities for the respective model element. In

[25]Cf. section 3.3.1 on page 32.

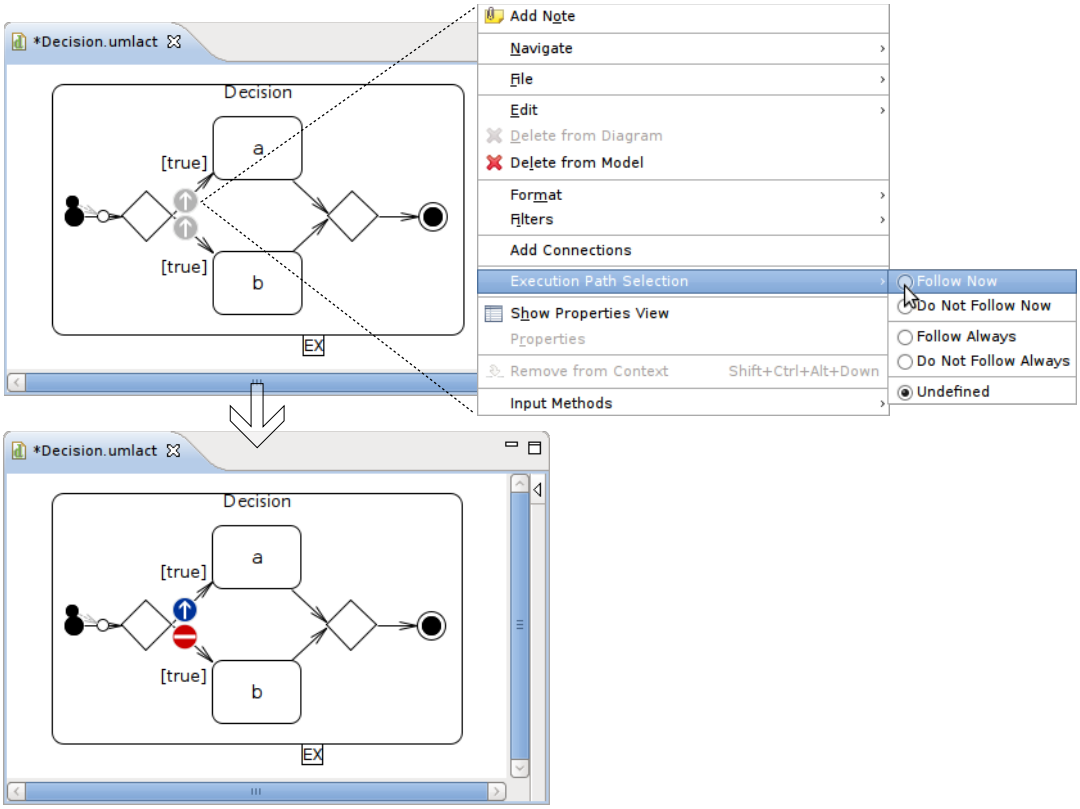

Figure 3.11: Configuration of the execution path to be used with the DMM Player

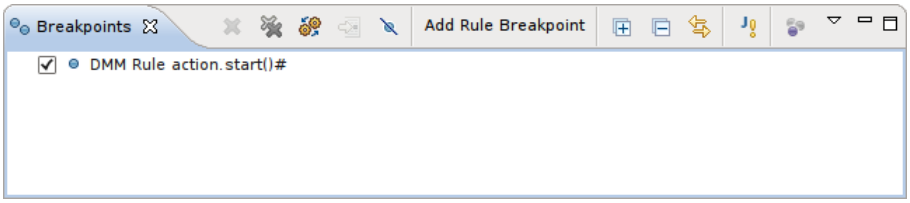

Figure 3.12: Detail of a screen shot showing the Breakpoints view of Eclipse extended by the Add Rule Breakpoint button

order to instruct the DMM Player to follow the particular model element, the user may choose either the sub-item Follow Now or Follow Always. The former item restricts the choice to this single occasion; if the model contains loops or other constructs that cause the particular position in the model to be reached again later, the user will be again asked for a choice. The latter item, Follow Always, makes a choice that is persistent.[26] Afterwards, the DMM Player will visualize the choice by coloring the traffic sign on the selected element blue; the other elements will be annotated by a "no entry" sign. See the lower half of figure 3.11 for a depiction of this.

Persistent choices can be made and changed any time, i.e., also when no execution takes place or the execution was not suspended for the selection of an execution path. Thus, the DMM Player will always provide the Execution Path Selection context menu item for applicable model elements. If the menu is opened when the execution is not suspended for a path choice, the menu items Follow Now and Do Not Follow Now will be not available, of course.

3.6.3 Manage Breakpoints

As the DMM Player will use the Eclipse Platform Debug component, its functions and UIs will be employed for operations related to debugging. Among others, that component provides the Breakpoints view (see figure 3.12) that lists all breakpoints that are currently defined in the Eclipse workspace and provides functions for manipulating them.[27]

The DMM Player extends the view by an Add Rule Breakpoint button. Clicking that button will open a dialog that allows the user to choose a rule from all rule sets that are available. After a rule has been selected, a new rule breakpoint will show up in the list of breakpoints.

By selecting the menu item Breakpoint Properties from the context menu of a DMM rule breakpoint, a property editor for that breakpoint can be opened (see figure 3.13). The editor allows to choose whether the execution should be suspended before or after rule application or already when the rule has just matched.[28]

[26] Besides the sub-items Follow Now/Always, the menu Execution Path Selection contains the sub-items Do Not Follow Now/Always and Undefined which alter the execution path configuration of the particular element accordingly.

[27] All the functionality provided by that view is documented in the JDT user guide [JDT09b] at http://help.eclipse.org/ganymede/topic/org.eclipse.jdt.doc.user/reference/views/breakpoints/ref-breakpoints_view.htm

[28] To differentiate the meaning of Before Application and When Matched: A matching rule does not imply that it will be immediately applied. Other rules that are also matching may be applied instead. Furthermore,

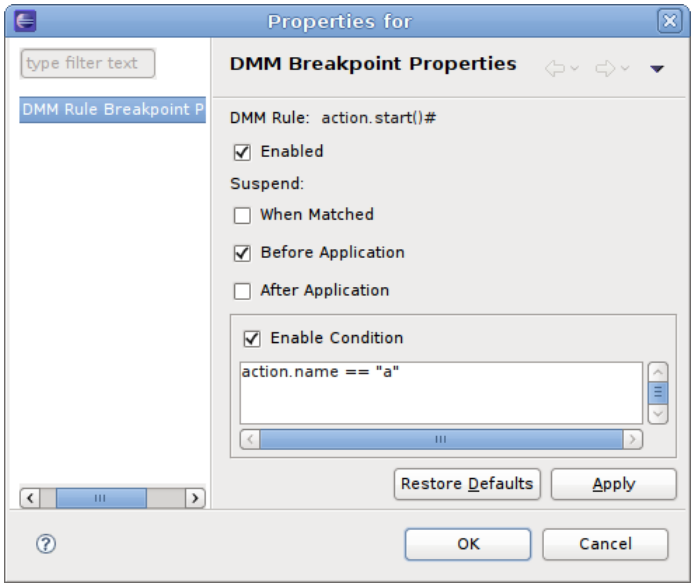

Figure 3.13: Screen shot of the properties editor for DMM rule breakpoints

The Add Rule Breakpoint button together with the properties editor provides the functionality required for the first step of the use case View State Before or After Rule Application (section 3.4.1.7). The same partially holds for the use case Locate state with certain properties (section 3.4.1.6). The properties to be located must however be defined before as property rules using the DMM rule editor.

3.6.4 View and Edit State in Concrete Syntax

The viewing and editing of the runtime model elements (described by the use cases in the sections 3.4.1.3 and 3.4.1.4) will be integrated seamlessly into the existing GMF-based diagram editors. The user will be able to use the extended diagram editors just like the basic editors.

property rules are never applied. Thus, the options Before Application and After Application are disabled for property rules.

4

Conceptual Analysis and Design

Having finished the requirements analysis, we now have a catalog of features the DMM Player should support in order to be sensibly usable. These requirements raise a number of conceptual problems to be solved; this chapter will now present these problems and—of course—their respective solutions. This involves both analysis and design activities; this chapter will restrict itself to a more conceptual design, though, such as the design of models and data structures. The more technical part of the design is subject of chapter 5.

First, we will discuss the concept of property rules which was already mentioned a number of times, but not yet completely introduced. The next topic that is dealt with is how to choose the execution path the DMM Player shall use when there are multiple options. This includes the clarification what "multiple options" actually means in this context. Section 4.2 covers that. The subject of the subsequent section 4.3 is the definition of visual steps, i.e., the selection of model states that the user of the DMM Player is supposed to see when a model is executed. Section 4.4 deals with the conceptual issues of breakpoints for DMM. The two final sections perform an integration of the afore identified solutions: Section 4.5 describes the concept of rule events which can be used to provide a unified way of identifying the situations that are relevant to the handling of execution paths, visual steps and breakpoints. Section 4.6 eventually presents a conceptual view of the whole process required to execute a model in the DMM Player.

4.1 Property Rules

The concept of property rules stems from the domain of graph transformations [BRRS08, KR06]. Property rules are not a very challenging concept, however, they prove to be useful in various contexts. We will now introduce property rules and discuss how they are applicable with the specialised DMM graph transformations.

We have already seen a special kind of a property rule in section 3.1.1.3: the rule that checks

for DMM specification failures. It indicates the case that a DMM rule tried to perform an invocation, which could however not be satisfied by any other rule.

As already explained, a property rule does not modify the state; to put it in graph transformation terms: its left hand side and right hand side are equal. Therefore, it does not influence the semantics of a rule set.[1] Its purpose is to constantly watch the state and to signal the cases when the rule matches the current state. This makes property rules very useful for testing and debugging purposes; the occurrence of forbidden states may be monitored with them, as it is the case with the rule for DMM specification failures. Property rules may be also used for finding valid states in which the user is interested in for some reason.

However, the given DMM implementation does not support property rules directly. The rule checking for specification failures is hard-coded as a GROOVE rule and is automatically added to all GROOVE rule sets that are transformed from DMM rule sets. It is also not possible to "misuse" the existing DMM rule types as property rules. Even though one might create DMM rules with an equal left hand and right hand side, the existing DMM rule types impose further limitations on the applicability of rules. Smallstep and premise rules are never directly applicable; bigstep rules are only applicable when no invocations are pending. A true property rule should give its creator complete freedom in the choice when it will match, though. Thus, it appears sensible to introduce support for property rules directly in DMM.

This can be reached quite easily: The DMM rule set meta model needs to be extended by a new class representing property rules. The graphical editor for DMM rules needs to be extended by a tool for creating property rules as well; we will not go into these details, though.

Finally, the transformation from DMM rules to GROOVE rules must be extended to handle the transformation of property rules. Fortunately, this is also quite easy. The special applicability limitations of the other DMM rule types are reached by extending the rules by constructs that depend on the state of the invocation stack which is contained in each GROOVE state for DMM. Thus, bigstep rules are extended by a construct that checks that the invocation stack is empty. As property rules are supposed to be applicable at any time, the trick is simply to leave out the constructs that check the invocation stack for property rules.

With these modifications, the user is able to define property rules. When GROOVE is used to calculate a transition system for a model with DMM semantics, property rules show up as labels on the states of the transition system. Further support for property rules will be realised with the rule events which are documented later in section 4.5.

[1] Even though adding property rules to a rule set causes changes to its transition systems, the semantics can be considered as equivalent. The changes to the transition system only consist of transitions with identical source and target states. The new transition system is therefore *stuttering equivalent* [Lam83] to the original transition system.

4.2 Controlling Multiple Execution Paths

Let us quickly recapitulate the requirements for the configuration of execution paths:[2]

- Differentiation between transition system forks that shall require the user to choose a path and other forks that do not so.
- When the user is required to choose a path, the model execution is suspended and the possible choices are displayed in the concrete syntax of the model.
- In the other case, when a fork is present, but the user is not required to choose a path, the DMM Player shall choose a path arbitrarily.
- The user shall be able to persist the choices, i.e., the choices should be automatically re-applied during the next execution of the model if the user chooses to do make a persistent choice.

The following sections will discuss how these goals can be reached. We will again use the UML activity semantics as a reference case; still, the resulting concept shall be generic, i.e., independent of any concrete semantics specification.

4.2.1 Terminology

In order to have optimal means of discussing the matter, we will coin a few terms for frequently needed concepts.

A fitting name for transition system forks on which the DMM Player is supposed to ask the user for a path is *execution path switches*. The name is borrowing from railway switches; both concepts provide means to determine the direction of the "traffic" at the fork of a network.

However, as transition systems are derived from the model and the rule set, execution path switches cannot be simply installed on the transition system. Rather, we need a way of describing constellations in a transition system that can be regarded as an execution path switch. This means, we need further differentiation:

On the general side, we have *execution path switch descriptions*—or shorter: *switch descriptions*—that describe such constellations. The methods for the description will be defined later. In a specific transition system, there may be *execution path switch instances*, or *switch instances*. These are specific occurrences of transition system forks that fit the particular switch description.

4.2.2 Execution Path Switches in UML Activities

To get a clearer picture of the properties of switch instances and switch definitions, we will use again the UML activity semantics as case study. We will first define the cases in which a switch instance should occur for UML activities. As we have no technique for defining switch descriptions, yet, we will do this in an informal way. Afterwards, we will explore some example transition systems of UML activity examples to find characteristics of switch instances that shall help us finding suitable means for switch definitions.

[2] Cf. section 3.3.1 and section 3.4.1.2

4.2.2.1 Informal Switch Description

To complete the reference case for further exploration, we will now provide an intuitive and informal switch description for the UML activity semantics. We have seen that forks in the transition systems of UML activities may be caused by decision nodes[3] and by concurrency.[4] In UML activities, concurrency may be caused by various constructs; fork nodes may cause it, however, it may be also caused by more than one outgoing edges from an activity node or by more than one activity node that will initially get a token.[5]

As concurrency may cause quite a lot forks, we define that those are not regarded as switch instances. Only forks caused by decision nodes should be switch instances and thus require user interaction.

4.2.2.2 Characteristics of Execution Path Switches

We have already seen a transition system for an activity with a decision node in figure 2.9. The outgoing transitions from the node at which the fork occurs are caused by the rule decisionNode.flow()#. So, let us first have a look at that particular rule in the DMM semantics. In fact, the DMM UML activity semantics contains two rules with the name decisionNode.flow()# which are pictured in the figures 4.1 and 4.2. The rules differ only in the way the guard is checked which is linked to the activity edge with the name out.

The first variant matches when no guard is attached to the outgoing edge, as the link with the name guard linked to an object with an negative application condition indicates. The other variant uses the premise rule activityEdge.P_checkGuard() to restrict the cases in which it matches to those in which a guard exists and it evaluates to true.[6] As the difference between the rules lies only in the configuration of an outgoing edge, the matches that result from a single decision node that is about to forward an offer, may include both rules.

If one of the rules matches, the actual change of the state is performed by the invocation of the rule decisionNode.validateOffer(). This rule (which is not pictured here) moves an offer from an incoming edge of the decision node to the outgoing edge identified by the node out.[7]

Thus, based on this example, an approach for recognizing switch instances could look like this: The switch description is a set of rules that trigger switch instances. A fork is a switch instance when the rules that are applicable on the current state are contained this set. For the DMM UML activity semantics, this set would consist of the rule decisionNode.flow()#.

In the considered example, all transitions going out from the state at which the fork started were caused by the rule decisionNode.flow()#. However, this is a very special case; there might

[3] Cf. section 2.1.5 and the transition system in figure 2.9.

[4] Cf. section 3.3.1 on page 32.

[5] Nodes that are initialized with tokens are initial nodes and actions without incoming edges; cf. section 2.1.5.

[6] The rule activityEdge.P_checkGuard() is shown in figure 4.3. As there is no DMM implementation of the Object Constraint Language [Obj06b], it is not complete and only works for guards consisting just of the boolean literal value true.

[7] This can be also seen in the transition system in figure 2.9; in the state before the rule decisionnode.flow()# is applied, the offer is still located on the incoming edge of the decision node. The rule which is applied next is the smallstep rule decisionNode.validateOffer(). In the resulting state, the offer is located on an edge outgoing from the decision node.

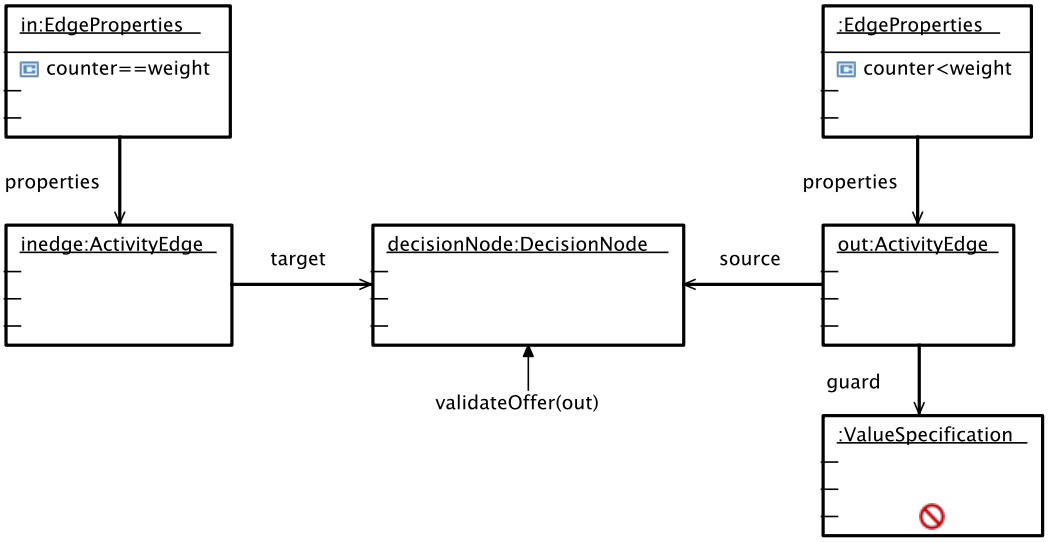

Figure 4.1: First variant of decisionNode.flow()# from the DMM UML activity semantics [Hor09]

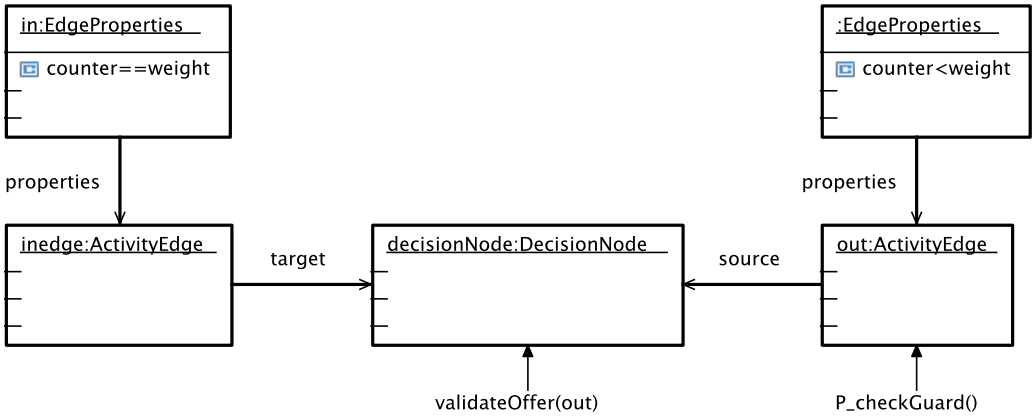

Figure 4.2: Second variant of decisionNode.flow()# from the DMM UML activity semantics [Hor09]

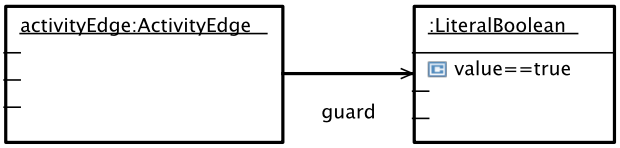

Figure 4.3: The premise rule activityEdge.P_checkGuard() from the DMM UML activity semantics [Hor09]

be other cases in which the situation is not that clear. Thus, we will now try to construct a complexer case.

More complex cases are formed when decisions and concurrency are combined in the model. Figure 4.4 shows such an activity diagram. The important part of the transition system induced by that activity is shown in figure 4.5. In order to be able to distinguish the control flow edges, names were given to the important edges. Furthermore, the rule decisionNode.flow()# was modified to include the name of the edge out as emphasised attribute. Thus, the transitions caused decisionNode.flow()# are now labeled with the name of the edge the offer flows to. The figure also visualizes selected morphisms between the nodes of the rule causing a transition and the elements of the activity diagram by dotted lines.[8]

State 23 is the first interesting state: The offer of the control token attached to the initial node has already passed the fork node. Thus, the offer was split into two offers; each offer is now attached to one control flow edge going out of the fork node. In the transition system, state 23 has four outgoing transitions, all of them induced by decisionNode.flow()#. However, the concurrency causes that both decision nodes are referenced each by two transitions. The two transitions in the left half of the transition system are caused by the decision node in the lower half of the activity diagram; likewise, the two transitions in the right half are tied to the upper decision node.

How should such a case be handled by the DMM Player? Basically, there are two options:

- As all transitions are caused by rules that shall trigger switch instances, the DMM Player could offer the user all four paths as choices. When the user selects one choice, he explicitly chooses one of the alternative edges going from a decision node. But implicitly, he also chooses one of the two concurrent branches in which the next execution step will be performed.

  Some steps later, the execution will hit again a fork; that is the case in the states 32, 33, 34 and 35. Two of the three transitions going out of each state are caused by the rule decisionNode.flow()#. The other transition is caused by action.start()# on the action that was reached concurrently to the decision node. The two decisionNode.flow()# transitions have the same morphism as two of the four choices that were offered to the user before.

  It is debatable whether all three transitions, or only the two decisionNode.flow()# transitions should be given as choices. We will not make a decision, as it is not relevant to the following considerations.

- The other option is to let the DMM Player automatically choose one of the concurrent branches before the user is asked to choose between the possible paths. This reduces the number of choices that are given to the user to two. These two choices will now refer to the same decision node in the diagram.

[8] No morphism is visualized for the nodes of the class EdgeProperties because these are not represented in the concrete syntax. The class EdgeProperties is an addition of the runtime meta model and has a 1-to-1 association to an ActivityEdge. The node with the type ValueSpecification is actually not contained in the morphism, as it is a negative application condition which forbids the existence of such an object.

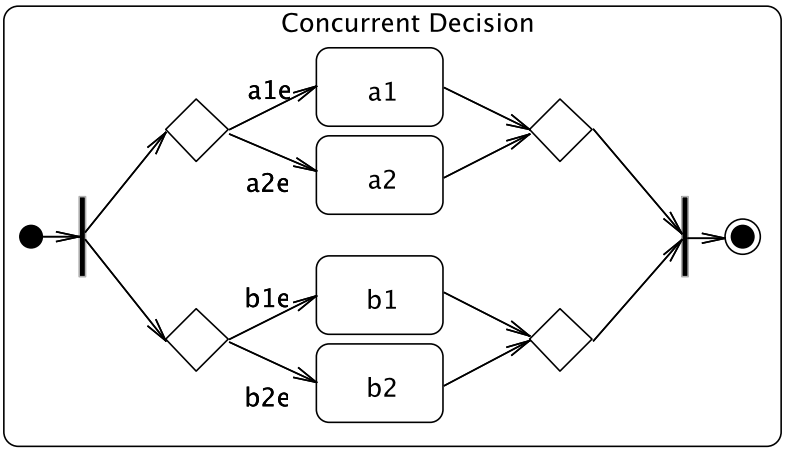

Figure 4.4: An activity diagram using both decision nodes and concurrency

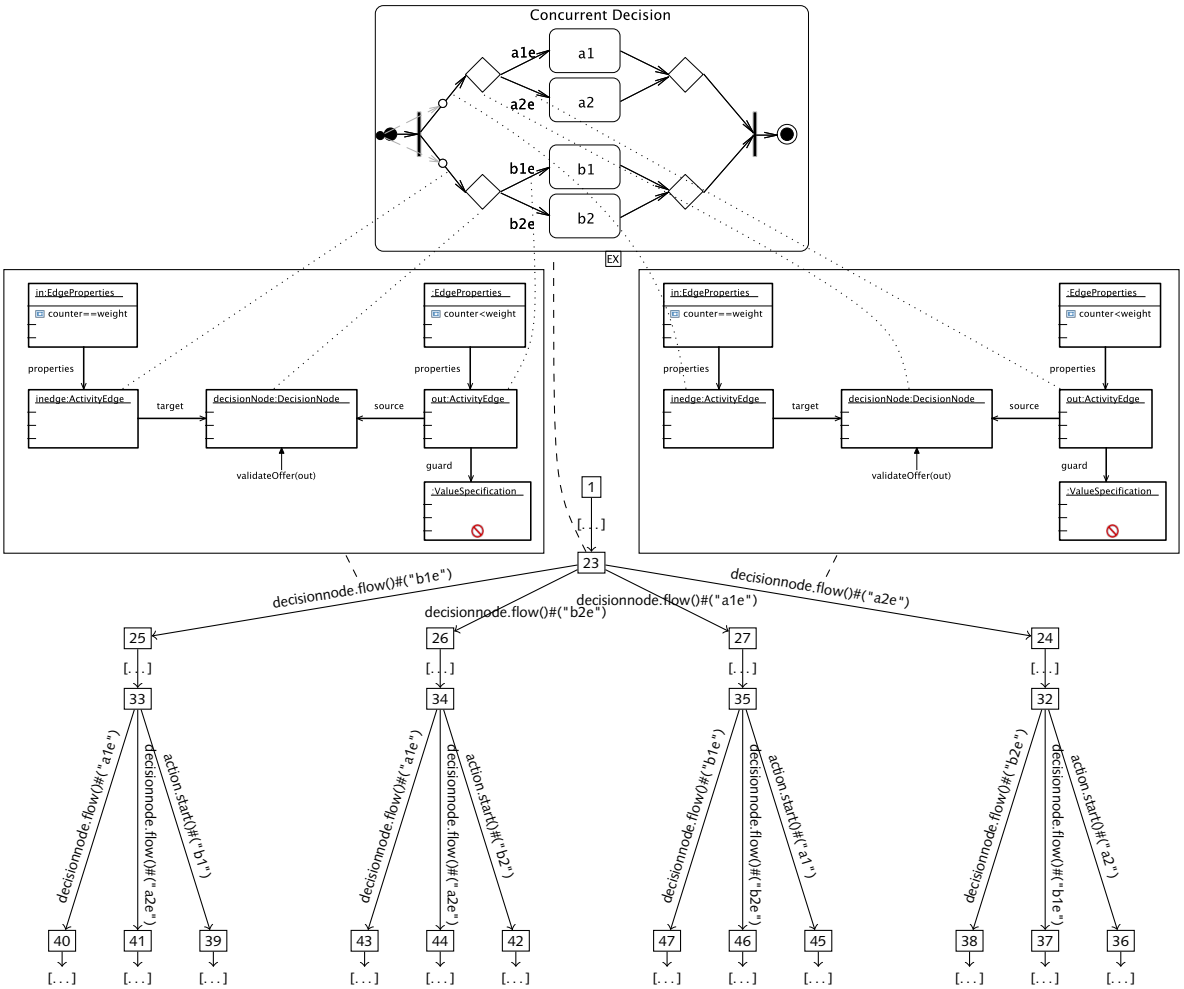

Figure 4.5: A part of the transition system induced by the model from figure 4.4

The next fork also has two groups of transitions: The transitions caused by decisionNode.flow()# which operate in one concurrent branch and the transition caused by action.start()# which operates in the other branch. Thus, the DMM Player will again automatically choose one of those branches. If action.start()# is chosen, only one transition remains and no choice by the user is necessary. If the other group is chosen, two transitions remain; thus, the DMM Player will request the user to choose one.

The first option gives more control to the user; beneath the choices induced by a decision node, he may also choose one of the concurrent branches for the next execution step. However, we have defined before that the user is actually not supposed to make choices induced by concurrency. Thus, the second option seems to be the correct way of determining the choices. If the user wants to control forks caused by concurrency, he may define switch descriptions for the remaining rules as well. Furthermore, the second option provides a more consistent user experience by giving distinct choices. One should also keep in mind that the decision nodes which incidentally become active in the same state are not necessarily located in the diagram as close as in this example.

4.2.2.3 Grouping Transitions by Common Elements

We have seen that the decisionNode.flow()# transitions going out of one fork do not need to be caused by a single decision node. Due to concurrency, several decision nodes may be involved here. However, in order to provide correct choices for a switch instance—as just defined—, we need to separate the transitions into groups with identical decision nodes. Then, a single group represents the correct choice set for a switch instance.

But how to find the transitions with common decision nodes? Fortunately, the necessary information can be extracted from the DMM rule that triggers switch instances and its matches, i.e., the morphisms between the rule and the current model state.

In the example in figure 4.5, the morphism of the transition between the states 23 and 25 is completely disjoint with the morphism of the transition between 23 and 24. However, the (unpictured) morphism of the transition between 23 and 26 differs from the morphism of 23–25 only by the element mapped to the node out. In the former morphism, out maps to the transition b2e, while in the latter morphism, it maps to b1e. Likewise, the morphisms of 23–24 and 23–27 also only differ in the mapping of out, which maps to a1e or a2e, respectively.

It can be easily seen that the morphism of the node decisionNode ties the particular match to the concurrent branch the decision node is located in. That is, grouping the transitions by the object that is mapped by the decisionNode object will yield groups of transitions that are caused by a single decision node and are not mixed due to concurrency.

It should be noted that the nodes inedge:ActivityEdge and in:EdgeProperties are uniquely determined by the node decisionNode.[9] Thus, grouping the matches additionally by those nodes would result in the same groups.

[9]The UML specification constitutes that a “decision node has one incoming edge” [Obj09].

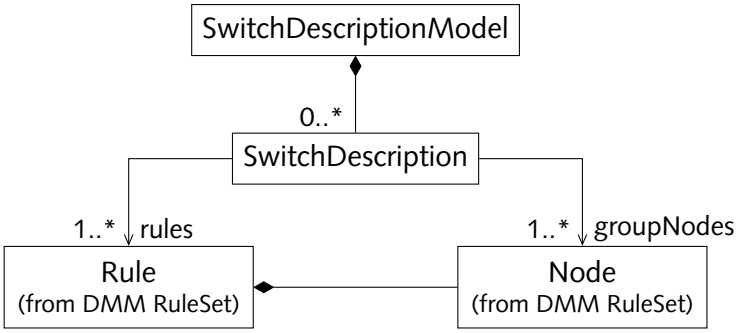

Figure 4.6: Class diagram of a possible switch description structure

4.2.2.4 Summary

We will now summarise the necessary aspects of a switch description for UML activities that have been found in the previous sections. There are two DMM rules that may trigger switch instances; the rules are widely identical and share the name decisionNode.flow()#. As the difference is limited to the outgoing edge, both rules may be involved in a single switch instance.

When a state has more than one outgoing transition caused by one of the rules, the matches need to be grouped by the node decisionNode to find out if a switch instance actually occurred. Both rules have such a node; even if the nodes have the same name, it should be emphasised that the nodes are not identical, as they are part of different rules.

We conclude the study of execution path switches in UML activities here and move on by putting these results on a generic level in the following sections.

4.2.3 Switch Description Model

We will start the generalisation with modeling the structures needed for switch descriptions. Figure 4.6 shows a class diagram of such a switch description structure.Switch descriptions, which are—obviously—represented by instances of the class SwitchDescription, can reference one or more Rule objects from the DMM rule set meta model. These are the rules that may be involved in one switch instance. The possibility to define more than one rule is necessary because all rules constructed for case distinctions—such as both decisionNode.flow()# rules from the previous example—might take part in a single switch instance. Furthermore the nodes that define the grouping of the matches are identified by the reference groupNodes pointing to the class Node. The reference must contain at least one Node for each Rule taking part in the other reference.

All switch descriptions for a particular DMM semantics specification are aggregated by the class SwitchDescriptionModel.

This switch description structure is not yet complete, though. One further feature will be added in the upcoming section 4.2.5.

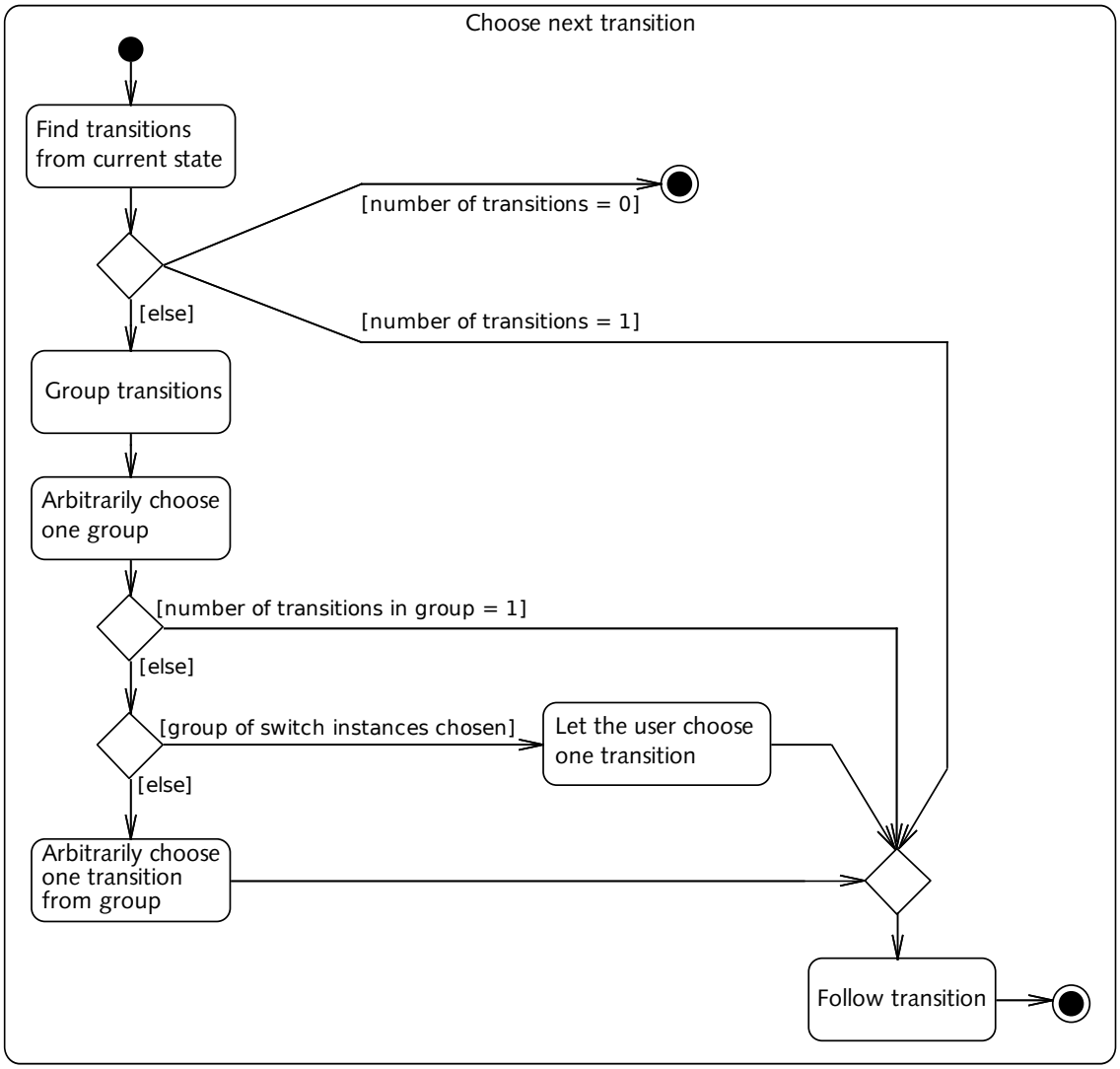

Figure 4.7: Algorithm that chooses the next transition and handles switch instances

4.2.4 Recognising Switch Instances

In a transition system, each state has a—possibly empty—set of outgoing transitions. The DMM Player will always follow only one transition, thus it needs to choose one, or—for switch instances—let the user choose one. We have already seen in section 4.2.2 that this is a quite involved task, as concurrent and non-concurrent transitions need to be separated. We will now formulate a generic algorithm that chooses the transition to be used for the next execution step and thereby recognises switch instances. An activity diagram modeling the algorithm can be seen in figure 4.7; it will be discussed by the following paragraphs.

- We start with the action Find transitions from current state. The complete set of possible transitions and the underlying matches is essential for this algorithm.

- The first decision considers the number of transitions. If there is only one or none outgoing transition from a state, there is no choice. Follow the transition or end the execution, respectively.

- Otherwise, the algorithm continues with the action Group transitions. This requires to look up the switch descriptions referencing the rules that triggered the respective transitions. If there is no switch description for a rule, the transitions caused by it will be put into the *group of non-switch instances*. Otherwise, the transitions will be grouped by the switch descriptions and by the model elements that are bound to the group nodes specified in the switch descriptions. These are the *groups of switch instances*.

- The next step is to Arbitrarily choose one group out of the groups produced by the previous action.

- If the group only contains one transition, there is no choice. The activity is concluded by Follow transition.

- Otherwise, if the chosen group is a group of switch instances, Let the user choose one transition contained in that group. The chosen transition will be the one used by the action Follow transition.

- Otherwise, the arbitrarily chosen group is the group of non-switch instances. The DMM Player will arbitrarily choose one transition from that group and follow it.

When a transition has been followed, a new state becomes active. Then, this algorithm can be applied on the new state. The whole process ends when a state is reached which yields no transitions that lead to a new state.

4.2.5 Visual Identification of Choices

An issue which is still open is the way the user shall be able to distinguish the choices at a switch instance. The challenge is to visualise the information that is encoded in the matches corresponding to the choices in such a way that the user is able to quickly distinguish the choices and to recognize the effect a choice will have.

We already saw an example of the anticipated user interface in the functional definition section (see figure 3.11 on page 50). The figure shows a UML activity diagram in the DMM Player at the moment the execution reached a decision node and thus triggered an execution path switch. The decision node has two outgoing edges which are each annotated by an icon that represents the choice.

The outgoing edges can be also found in the rules that trigger execution path switches for UML activities, i.e., the rules with the name decisionNode.flow()#.[10] In both rules, an outgoing edge is represented by the node out:ActivityEdge. As the node is not a universally quantified

[10] Cf. figures 4.1 and 4.2 on page 57.

node and thus binds only one model element, there will be a match for each followable outgoing edge. Due to the grouping of matches described before in section 4.2.2.3, each match in the finally chosen group will have the same decision node bound to it. Furthermore, all other nodes are either uniquely determined by the model element bound to decisionNode or by the model element bound to out:ActivityEdge. This means that the chosen group of matches will not contain two or more matches with the same model element bound to out:ActivityEdge. Thus, this node can be used for identifying and distinguishing the matches of a group.

Fortunately, the outgoing activity edge is also a very suitable model element for identifying the direction an offer will take after the rule decisionNode.flow()# has been applied with the particular match, as it will appear in the concrete syntax as an arrow pointing away from the decision node.

Thus, the DMM Player will just have to find the concrete syntax representations of the model elements bound to the node out:ActivityEdge and annotate them with the respective "traffic sign icons". When the user picked one of those model elements, the transition to be used can be again identified by the selected element.

Generalising these findings, the switch description needs to be extended by a further node from the associated rule that can be used to uniquely identify each match of a group. Thus, a switch description now consists of a rule, a set of nodes from that rule that are used to partition all matches into groups and a further node that can be used to identify a match in its group. The remaining nodes of the rule need to be uniquely determined by either node or node set, respectively. If this is not the case, the matches may not be uniquely identifiable which means that a choice the user made may be ambiguous for the DMM Player. The DMM Player would then choose an arbitrary match out of the matches that are bound the particular model element that identified the user's choice.

Furthermore, the node that is used to identify the match needs to have a suitable representation in the concrete syntax of the language. If this is not the case, one might check whether there are further model elements which are uniquely determined by the original node and which do have a suitable concrete syntax representation. If this is the case, the rules can be safely extended by nodes representing those model elements. As they are supposed to be uniquely determined by an existing node, the applicability and effect of the rule will not change.

In order to be able to configure the node that identifies a match, we will extend the switch description from section 4.2.3 by a further association between the class SwitchDescription and the class Node. This association represents the node used for identifying a match of the particular rule. The association role name of the node is identificationNode. Figure 4.8 shows the extended and thus completed switch description model.

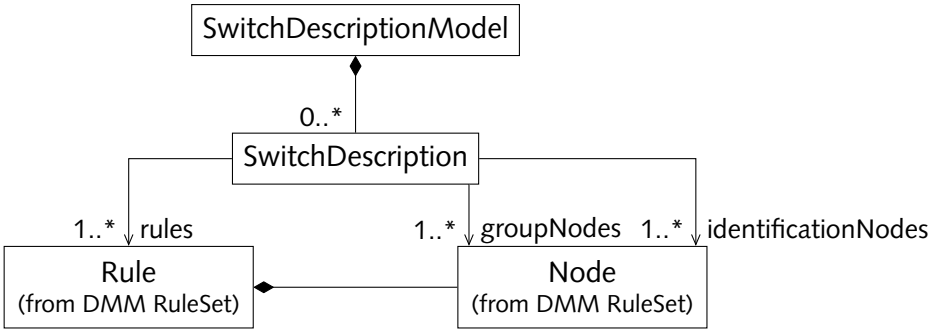

Figure 4.8: Class diagram of the final switch description structure

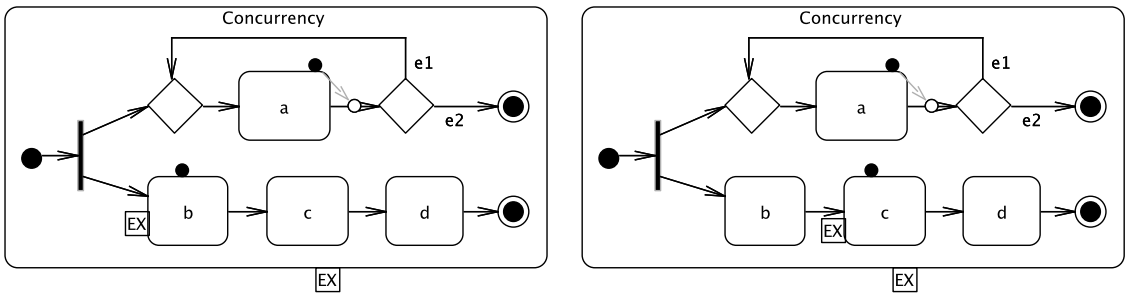

Figure 4.9: Two states of an activity using concurrency

4.2.6 Persistent Choices

A requirement for the DMM Player is that execution path choices made by the user may be persistent. This means that the user makes a choice once and the DMM Player will remember that choice and will automatically make that particular choice in the future.

The choices we have seen so far are references to transitions which are parts of switch instances. Thus, the transitions—and their source and target states—were known before the choice was made. When choices are supposed to be persistent, it is however not feasible to bind the choices to transitions and thus to whole states. Again, concurrent ongoings in the states cause problems.

Figure 4.9 shows two states of an activity using concurrency. The upper concurrent branch uses a loop constructed using a merge and a decision node. The user may want that the execution in the upper branch loops until the execution in the lower branch reaches the final node and thus ends the whole activity execution. Thus, he could make a persistent execution path choice which directs the offer onto the edge e1. When he makes that choice in the left state, the offer and its token will make a loop; in the meanwhile the execution in the lower branch will also continue. Thus, the state shown in the right of the figure is a state that may be reached from the left state. Due to the persistent choice, the offer in the upper branch is supposed to take again the edge e1.

However, the state is not identical to the state in which the persistent choice was originally made. The control token in the lower concurrent branch moved from action b to c. Thus, the

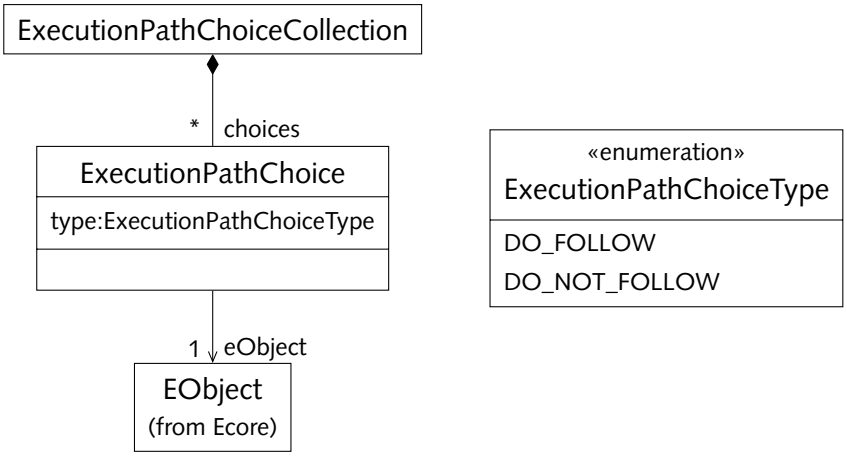

Figure 4.10: Class diagram of the structure that stores persistent path choices

re-application of a persistent choice cannot be decided by simply comparing the states.

Instead, the DMM Player will utilise just the identification node of switch descriptions,[11] i.e., in the case of UML activities the edges going out of decision nodes. A persistent choice will just mark the model elements that are bound to identification nodes either as elements to be followed or as elements not to be followed. These choices are saved in a global map from model elements to either the setting "follow" or "do not follow". Figure 4.10 visualises that data structure as a class diagram. The class EObject is the Ecore base class that is the super class of all other classes. Thus, it can represent every possible element of an Ecore model.

When the model execution hits a switch instance, that map will be checked for entries for the model elements that are bound to the identification nodes of the new switch instance. This check must distinguish two cases:

- There is an entry for each model element bound to an identification node of the current switch instance and exactly one is set to "follow". In this case, follow the transition identified by that model element without prompting for a user interaction.
- Otherwise, the stored choices cannot be used in the current situation. In this case, the user should be prompted to choose a path, just like there would be no persistent path choice. This situation may arise when an additional outgoing edge of the particular decision node becomes available. It may have been unavailable before because it was blocked by an offer or its guard evaluated to false.

4.3 Definition of Visual Steps

As specified in section 3.6.1, the DMM Player will perform the model execution step-wise; a delay between each step serves for the purpose that the user may visually follow the current state of the execution.

[11] Cf. section 4.2.3 on page 61.

4.3.1 Problems With The Natural Step Measure

Of course, each programming language has a "natural" step measure, which is defined by its architecture; for classic, instructional programming languages, this is the execution of one instruction. For a DMM-specified language, the natural step is represented by the execution of one DMM rule.

Using this step measure for the visualisation of the execution progress of a model is not ideal, though. This is due to a number of reasons:

Not every rule needs to change the state of runtime model. Many rules just perform invocations on other rules which perform the actual modifications on the model. Let us look again at the transition system of our running example in figure 2.9. The states in the transition system are GROOVE states, which contain beneath the actual model state also the invocation stack. Thus, each invocation will cause also a change to the GROOVE state. The conversion back to the EMF representation of the model will however discard the information in the invocation stack. The information contained in the invocation stack is not representable in the runtime meta model or the concrete syntax—which is what the user will see—anyway. So, only a subset of the transitions seen in the transition system will cause a change the actual model.

For each branch of the transition system in figure 2.9, the actual model is modified by only 18 transitions.[12] Totally, one branch contains 44 transitions. So, when using the application of each rule as a measure for a visual step, the user will see phases in which the model does not change whose durations are a multiple of the configured step delay. Furthermore, the relation between rules that cause a change and rules that do not is not constant; we can see in the example that there are a number of consecutive transitions that cause changes at the end of the transition system, while the middle contains some bigger gaps between the transitions that cause changes.

Another problem of using all rule executions as step measure are *temporal inconsistencies*. An example for this can be again found in the transition system in figure 2.9. In the concrete syntax diagram that is attached to state 110, you can see two tokens. However, as the activity diagram does not use any concurrency, there should be only one token. This is a temporal inconsistency which is caused by the fact that creation and deletion of tokens is performed by separate rules. The creation of the new token is performed by the rule offer.accept(), which afterwards invokes token.destroy() which actually deletes the old token.

Such temporal inconsistencies are an absolute natural result of the way DMM uses invocations to structure the rule set. Visualising temporal inconsistencies is also no problem when the user of the DMM Player is a semantics developer, as he is aware of the internal structure of the rules and may be even interested in seeing those intermittent results. A simple model developer, however, is not necessarily aware of those effects and should thus not be confronted with temporal inconsistencies.[13]

[12] On the right branch of the transition system, these are the transitions between the states 8/10, 11/12, 14/16, 20/22, 26/28, 36/38, 46/50, 50/52, 56/60, 66/68, 68/70, 82/84, 88/92, 96/98, 106/110, 110/111, 113/116, and 116/118.

[13] Cf. section 3.2 for the stakeholder analysis.

4.3.2 Approaches to Defining Visualisation Steps

We will now try to find visualisation step measures that fulfill all requirements. We have seen that, depending on the user, we need different step measures; a simple model developer must not see temporal inconsistencies. A DMM semantics developer should see the execution process as detailed as possible; yet, he should not experience longer delays when a rule just performs invocations and does no direct modification on the model.

4.3.2.1 Changes to the Model

Informally expressed, a sensible step measure for a semantics developer is quite simple: Every execution step that changes the model should be visualised. Thus, the developer sees temporal inconsistencies and experiences a model change for each visualised step.

It gets more complicated when the user is a model developer: The central goal is to hide temporal inconsistencies from such users. We will discuss several approaches to that problem in the following sections.

4.3.2.2 Application of Bigstep Rules

A simple approach would be visualising the model changes only after all invocations of bigstep rules have finished. As we saw, temporal inconsistencies arise when the modifications to the model are delegated to other rules by an invocation. Bigstep rules cannot be invoked, thus, after all invocations that were directly or indirectly caused by a bigstep rule have finished, temporal inconsistencies caused by smallstep rules can be ruled out. It is of course also possible to construct rule sets that exhibit temporal inconsistencies caused by constructs other than smallstep rules. However, such rule sets need to respect such constructs in all bigstep rules; this cannot be regarded as good style. Thus, we will not consider such cases here but only temporal inconsistencies caused by smallstep rules.

Besides the states that are left *after* the completion of bigstep rules, consistent model states may also occur *during* the execution of a bigstep rule. Looking again at the transition system from figure 2.9, we will see that the last application of a bigstep rule occurs on the state 86.[14] In the state 86 (pictured in figure 4.11), the token is still attached to the action a; the offer is located on the outgoing edge of the merge node. As this state is directly before the application of a bigstep rule, no invocations are pending and thus this model state will be visualised.

The next model state which will be visualised in this approach will be the final state 118, because all previous transitions were caused by smallstep rules which were directly or indirectly called by the bigstep rule activityfinalnode.accept()#. In the final state, all runtime model elements such as tokens and offers have been deleted. Thus, the visualisation would change from a state with a token on action a and an offer on the edge before the final node directly to a diagram without tokens and offers. Such a behaviour would probably be quite confusing to the user; the reason for the deletion of the offer and token is that the execution was finished because a token reached a final node. This reason would never be shown to the user, though.

[14] And on state 85 in the other branch. For brevity, we will only look at the right branch, though.

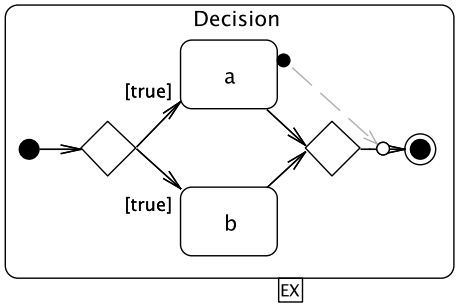

Figure 4.11: State 86 from the transition system in figure 2.9. If only bigstep rules are used for visualising the model state, this will be the last visual state before the execution ends.

Thus, the approach of visualising the model state only after a bigstep rule has finished all invocations is not sufficient.[15] Yet, it may be possibly combined with another approach that causes further states to be visualised.

4.3.2.3 Explicitly Specified Rules

A very straight-forward approach is specifying all rules that leave a state that should be visualised. Thus, after the application of a rule, the DMM Player would check whether the rule is contained in that set of rules, and visualise the state if that is the case.

Actually, two different moments after the application of a rule can be relevant for the visualisation: If the rule performs invocations, the first state after the application of the rule will just contain the modifications the rule did directly using nodes or edges with create or destroy roles. The modifications by the invocations are however not yet done, as the invocations have just been queued into the invocation stack. The following transitions will process the invocation stack and thus the invocations performed by the original rule.

Thus, we can differentiate between the moment *after application* of the rule and the moment *after completion* when all invocations performed by the rule have been processed.

Specifying the visualisation steps using explicit rules is the most flexible approach, but also the most laborious one. As said before, the DMM UML activity semantics comprise 217 rules.[16] Each rule needs to be checked whether its result should be visualised or not. Furthermore, when someone extends or modifies the rule set, he needs to keep in mind to check whether that set of rules needs to be updated as well.

In this context, the way DMM represents case distinctions is a significant problem. In a case

[15] Still, the effect caused by the bigstep rule activityfinalnode.accept()# seems to be quite big. Splitting the bigstep rule into two bigstep rules, where the first puts the token on the final node, and where the second—triggered by a token on a final node—performs the final clean up may be worth considering. This may cause issues with concurrency as other rules may get also applied in between, though. To cope with this problem, an extension to DMM which provides a better control over concurrent rule applications such as rule priorities may be imaginable. All this is beyond the scope of this thesis, though.

[16] Cf. section 2.1.5 on page 15.

distinction, each case is represented by an equally named and similarly structured rule; the distinction of the cases is represented by the differences in the node and edge structure of the rules.[17] Due to the similar structure, it is most likely that all rules can be treated the same for the visualisation. However, the configuration for the visualisation needs to be done separately for each rule.[18] The disadvantage of a large number of rules that need to be evaluated for a step configuration can be slightly alleviated by combining this approach with the approach that triggers the visualisation after the invocations of a bigstep rule have completed.[19] If the visualisation of the model is always updated after the completion of bigstep rules, only the smallstep rules of a DMM semantics need to be checked for the remaining visualisation configuration. Thus, for the DMM UML activity semantics, 172 rules remain. This can be only regarded as minor improvement.

4.3.2.4 Checking the Model for Consistency

Another approach to the step configuration is using property rules to check the current state of the model for properties that allow visualisation. The major motivation of excluding certain states from visualisation were the temporal inconsistencies that should not be shown to certain users. Thus, a property rule could be used to check each model state for consistency. If no inconsistency is found, the state may be visualised.

This approach has the advantage that is completely independent from the rules that implement the semantics. It is not affected by changes in the rule set. However, some inconsistencies are only recognisable if beneath the model state further context information is available.

The temporal consistency of two tokens depicted in the state 110 in figure 2.9 can be only recognised because the preceding and succeeding states show only one token. Viewed isolated from the other states this cannot be recognised easily. The only way to check this would include performing a simulation of the activity, which is outside of the scope of a property rule.

4.3.2.5 Summarising the Approaches

We have seen that each approach has its advantages and disadvantages. Only the approach listing all rules that shall trigger a visualisation can be used universally, but all other approaches may be usable in certain cases.

Thus, the best approach might be allowing all approaches; as DMM is a generic technology, one can choose for each DMM semantics specification the best approach separately.

Figure 4.12 shows the data structures necessary for defining visual rule steps using any of the aforementioned approaches. This model will be later in section 4.5 mapped to the new concept

[17] A nice example for this can be seen in the figures 4.1 and 4.2. The general method of binding invocations to rules has been described in section 2.1.3.

[18] A solution to this problem could be the introduction of a new concept that automatically concentrates the related rules in a group. The visualisation configuration could then reference the group instead of the group's members. However, the definition of such groups is a quite complex matter. For example, due to the effects of polymorphic types, one rule may be in several—possibly nested or overlapping—groups. Thus, this thesis will not follow this topic any further.

[19] Cf. section 4.3.2.2 on page 68.

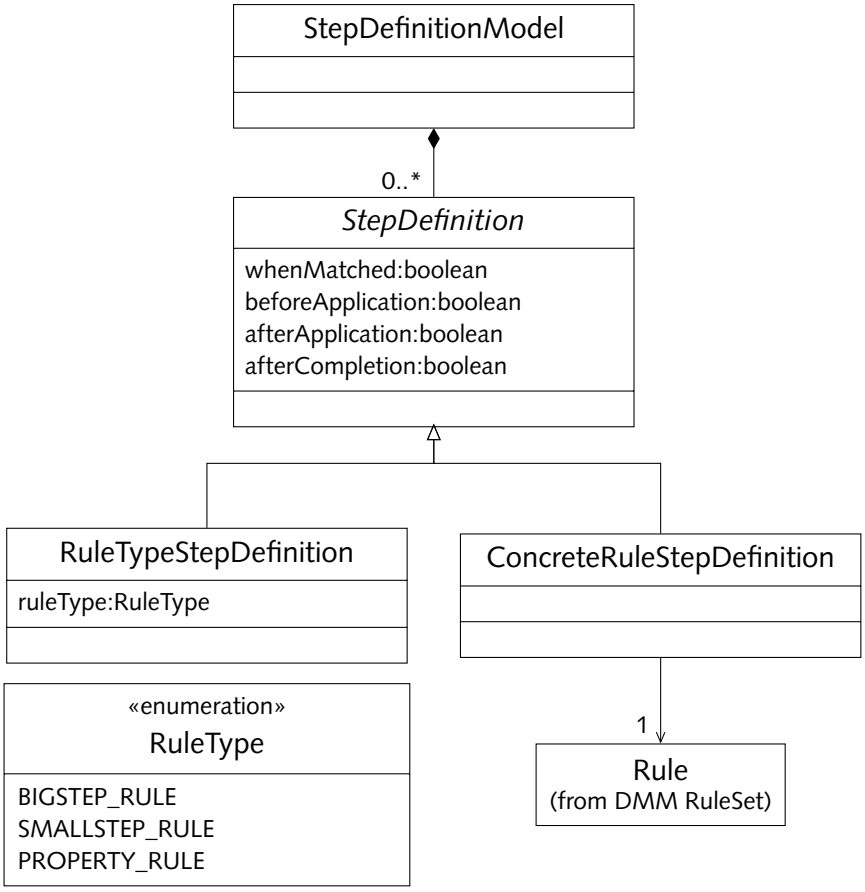

Figure 4.12: The step definition meta model as class diagram

of rule events.

If there is no step definition for a certain semantics specification, the DMM Player will use a default step definition which regards all applications of bigstep and smallstep rules as a step.

4.3.3 Step Definition for UML Activities

The first application of the step definition approach is the definition of visual steps for the UML activity semantics. Due to the size of the UML activity semantics, we will limit ourselves to the rules that occur during the execution of our running UML activity diagram example.

In order to define the visual steps, we need to first determine which states should be shown to the user. Figure 4.13 again shows the transition system of the running example. All states that should be visualised have been marked and are also pictured in their concrete syntax on the right side of the figure. The left side of the figure represents the invocation stack during the execution of the model; whenever a rule has performed an invocation, an edge connects these rules. Furthermore, the edge going straight down from a rule represents the period in which invocations performed by this rule are still pending. The invocations performed on the right

branch of the transition system have been omitted from the figure, as they are identical to the ones from the left branch.

The states to be visualised have been chosen in such a way that no temporal inconsistencies are exhibited. This includes the two tokens appearing in state 110 and also the action execution node appearing in state 28 before the token has actually reached the activity which is going to be executed.

We can see that the states between the execution of bigstep rules are always states to be visualised. Thus, we can start with a step definition that requests a visualisation every time a bigstep rule is going to be applied or has finished all its invocations.[20]

Furthermore, there are the states 51, 52 and 111 which do not occur in between two bigstep rules but should still be visualised. All states represent the case that a token has just arrived on a node; in the case of the states 51 and 52 that node is an action, in the other case it is the final node.

In order to identify circumstances that can be used to identify these states in the step definition, we will need to take a closer look at the invocation stack around the states. The rule that was applied directly before the states is in both cases token.destroy(). This can however not be used for the step description, as it is also used in other cases that should not be visualised. The rules applied afterwards are activityNode.destroyTokens() and activityExecution.terminate() which might be usable for a step definition.

However, taking a broader look yields a better solution: In both cases, the rules offer.getAccepted() and offer.accept() have finished all their invocations before the state going to be visualised occurs. Actually, this is quite self-evident as these rules move the token from its previous place to the new place found by the offer which also gets deleted meanwhile. As offer.accept() is invoked by offer.getAccepted() the use of one rule as step definition suffices. The difference between both rules is that offer.getAccepted() additionally interrupts possible interruptible regions using a further—optional—invocation after offer.accept(). Those regions are not part of the example; nevertheless, it is sensible to wait for the interrupting to be completed as this should be coincide with the acceptance of the offer. Thus, the visualisation of the state should occur after offer.getAccepted() finished all its invocations.

Thus, we can define the step definition for UML activities as follows: First, we use a RuleTypeStepDefinition with the attributes beforeApplication and afterCompletion set to true. The definition is completed by a ConcreteRuleStepDefinition for the rule offer.getAccepted() with the afterCompletion attribute set to true.

[20] Actually, it is sufficient to either watch for bigstep rules going to be applied or for bigstep rules which have finished all invocations. This is due to the fact that each bigstep rule application is preceded and succeeded by another bigstep rule application. The two exceptions, the beginning and the end of the execution will be always visualised.

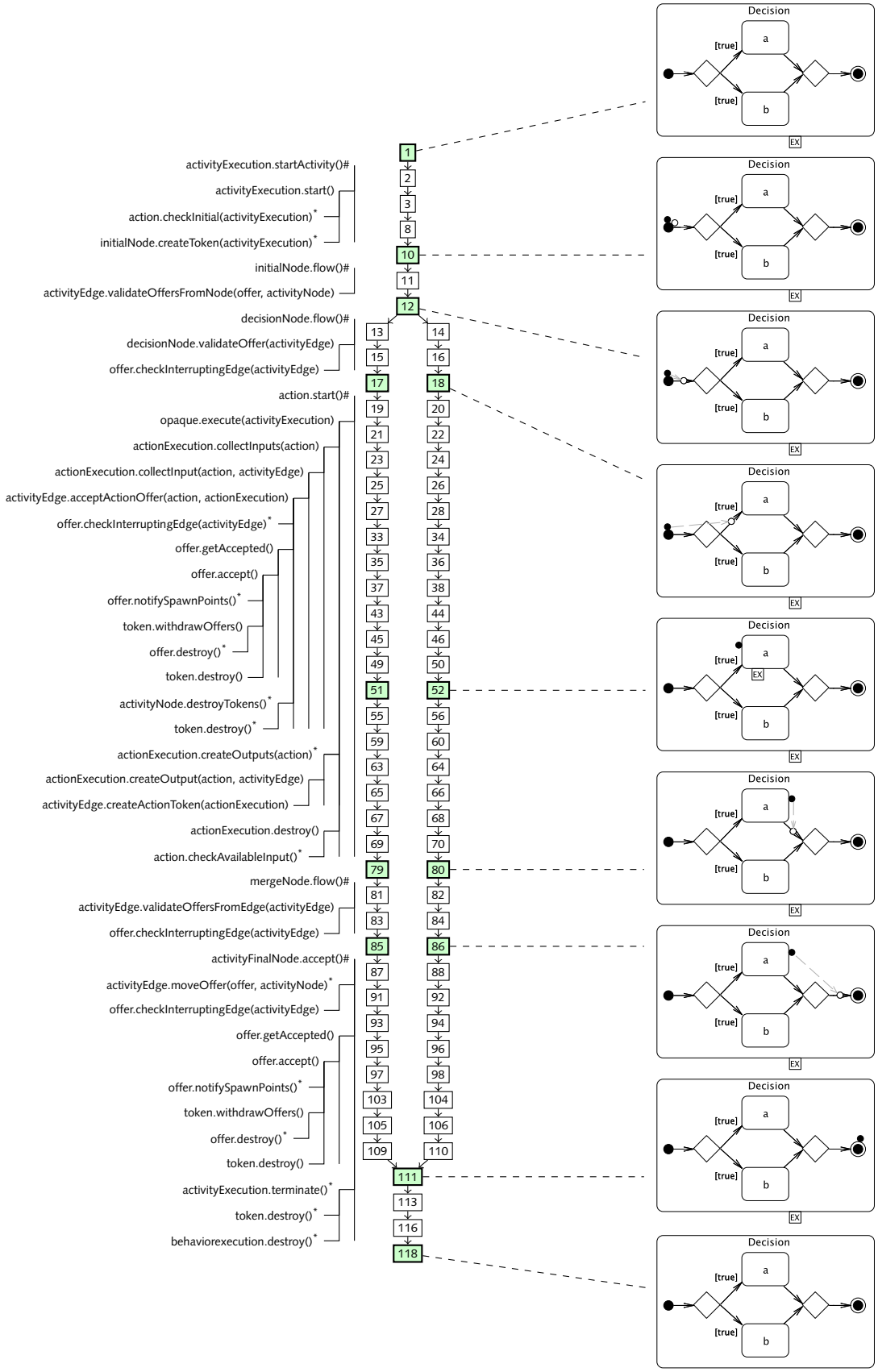

Figure 4.13: States of the UML activity example that should be visualised

4.4 Breakpoints

From a conceptional point of view, breakpoints on DMM rules are fairly simple. The DMM Player will map rule breakpoints to rule events and suspend the model execution when a rule event for a breakpoint occurs. Exceptions are rule breakpoints with additional conditions[21] and concrete syntax breakpoints.[22] Thus, we will examine these topics in the following sections.

4.4.1 Conditions

Conditions are supposed to be a quickly and easily usable way of limiting the applicability of rule breakpoints. An example is a breakpoint on the DMM UML activity rule action.start()# that shall only suspend the execution when an action with a certain name is about to be executed. The name of an action is represented by an attribute in the class Action, which also occurs as a node in the aforementioned rule.

DMM already provides a similar feature: In a DMM rule, conditions may be attached to nodes. These conditions provide a textual expression syntax which allows to formulate boolean expressions on the attributes of the respective node and other nodes in the rule. The syntax borrows from the established expression syntax used by Java or C; details about these expressions can be found in [Bau08]. Such a condition limits the applicability of the whole rule; if the expression evaluates to false, the rule cannot be applied.

It is obvious that conditions for rule breakpoints should try to reuse this existing functionality. However, one fundamental difference between the two condition concepts needs to be resolved: The existing conditions for DMM rules limit the applicability of the whole rule and thus influence the semantics. The conditions for breakpoints however must not influence the semantics; they are only supposed to monitor the existing semantics in some way.

This problem can be solved by duplicating the particular rule and attaching the breakpoint conditions to the duplicate. A duplicate that has been only extended by conditions will be applicable in a subset of the states the original rule is applicable in. Furthermore, conditions do not change the right hand side of a rule. Thus, when such a duplicated rule is added to a DMM rule set, the states of the transition systems induced by the rule set will not change. The only change are further transitions between the states; these transitions are caused by the new rule and only occur between states the original rule already caused a transition before. Thus, such a modification of the rule set does not change the semantics.

As the new rule matches only when the original rule matches and when the condition is true, the transitions caused by the new rule exactly represent the occasions when a rule breakpoint which is restricted by the respective condition should be activated.

The user interface for defining breakpoint conditions provided by the DMM Player will be a simple text input field in the breakpoint properties editor.[23] Here, the user may write the conditions that shall be attached to the nodes of the particular rule. The conditions need to be

[21] Cf. section 3.3.2.1 on page 34.
[22] Cf. section 3.3.2.4 on page 36.
[23] Cf. section 3.6.3 and figure 3.12 on page 51.

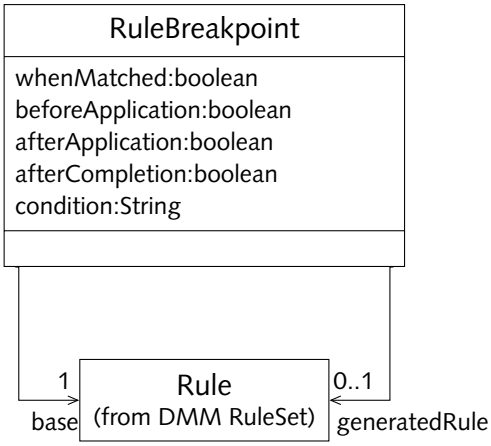

Figure 4.14: Class diagram of the breakpoint model

prefixed with the name of the node they are supposed to be attached to. For example, writing the breakpoint condition `action.name == "a"` will attach to the node with the name `action` the condition `name == "a"`.

When the user has defined such a rule breakpoint condition, the DMM Player will automatically duplicate the referenced rule and attach the condition to the duplicate. During the model execution, only the applicability of the new rule is regarded as a trigger for the particular breakpoint.

As DMM strives to provide solutions with a minimal semantic gap, it might be feasible to allow the user to define the conditions directly in the concrete syntax of the DMM rule. However, for realising such a feature, a number of further problems need to be solved: The concrete syntax of DMM rules must differentiate conditions that belong to the semantics and breakpoint conditions in some way. Furthermore, in order to guarantee that the semantics stays unchanged, it is crucial that the user only modifies the breakpoint conditions. This needs to be ensured in some way by the editor. Thus, for the time being, the simple solution with a textual input will be employed.

4.4.2 Breakpoint Model

Figure 4.14 shows the breakpoint model. Each rule breakpoint is tied to a DMM rule which primarily defines when the breakpoint should suspend the execution; the boolean attributes define the exact moment of the suspension.

Optionally, a condition may be specified for the breakpoint. If this is the case, the DMM Player will automatically generate a further rule which will be used as breakpoint trigger instead of the original rule.

4.5 Rule Events

The goal of rule events is to integrate the common requirements of breakpoints, visualisation steps, and execution path switches in one concept and later also in one software implementation. As we have seen that the requirements of those concepts have a number of similar properties, it is obvious to aim for such an integration. Benefits are a lower implementation effort, less code that requires maintenance, and thus in the long term a more stable product.

Basically, all concepts need to react in some way to the application, or just the applicability of a rule to the current model state. Generalising this, all concepts need to react to an event which always involves a rule. Thus, we call the new, common concept rule events.

We will now introduce the rule event model; afterwards, we will map the aforementioned concepts to that model.

4.5.1 The Rule Event Model

Figure 4.15 shows the rule event model as a class diagram. The abstract class RuleEventDefinition is the core of the rule event model; it allows the specification of the kind of rule events the DMM Player should react on. The reaction is defined by the abstract class Action which is bound to the RuleEventDefinition with a composition. Concrete subclasses of Action, which are not included in the diagram, can be used to bind a RuleEventDefinition to actual event handler implementations defined inside the DMM Player code-base.

The class RuleEventDefinition has five boolean attributes that specify when the actions should be invoked in reaction to a rule event. If whenMatched is true, the actions are executed already when a match is found that fulfills the conditions specified by the rule event and which are described below. The attribute beforeApplication orders to execute actions when a eligible match is found and it is going to be applied. Likewise, afterApplication performs the action after the application. The attribute afterCompletion waits for all invocations that are performed by the particular rule to be completed before performing the specified actions. The attributes may be combined in any way, causing possibly multiple action executions for one event.

The two attributes ending with Multi specify that the actions should only be performed when more than one match is found that fulfills the further conditions.

These conditions are specified by the concrete subclasses of the class RuleEventDefinition. The class RuleTypeEventDefinition can be used to specify rule event definitions relating to the types of DMM rules, i.e., which perform actions whenever a bigstep rule, smallstep rule or property rule matches the current state.[24] The actual types that trigger the particular rule event can be defined in the attribute ruleType.

The ConcreteRuleEventDefinition requires the specification of one or more concrete rules that trigger the event. Optionally, nodes from these rule can be specified that are used to separate the matches into groups. Specifying those multiMatchGroupNodes is only meaningful if one of the Multi attributes in the RuleEventDefinition is set to true.

[24]Premise rules are not included here because these rules are merged into the other rules before the rule set gets interpreted. Therefore, they will never directly cause a match.

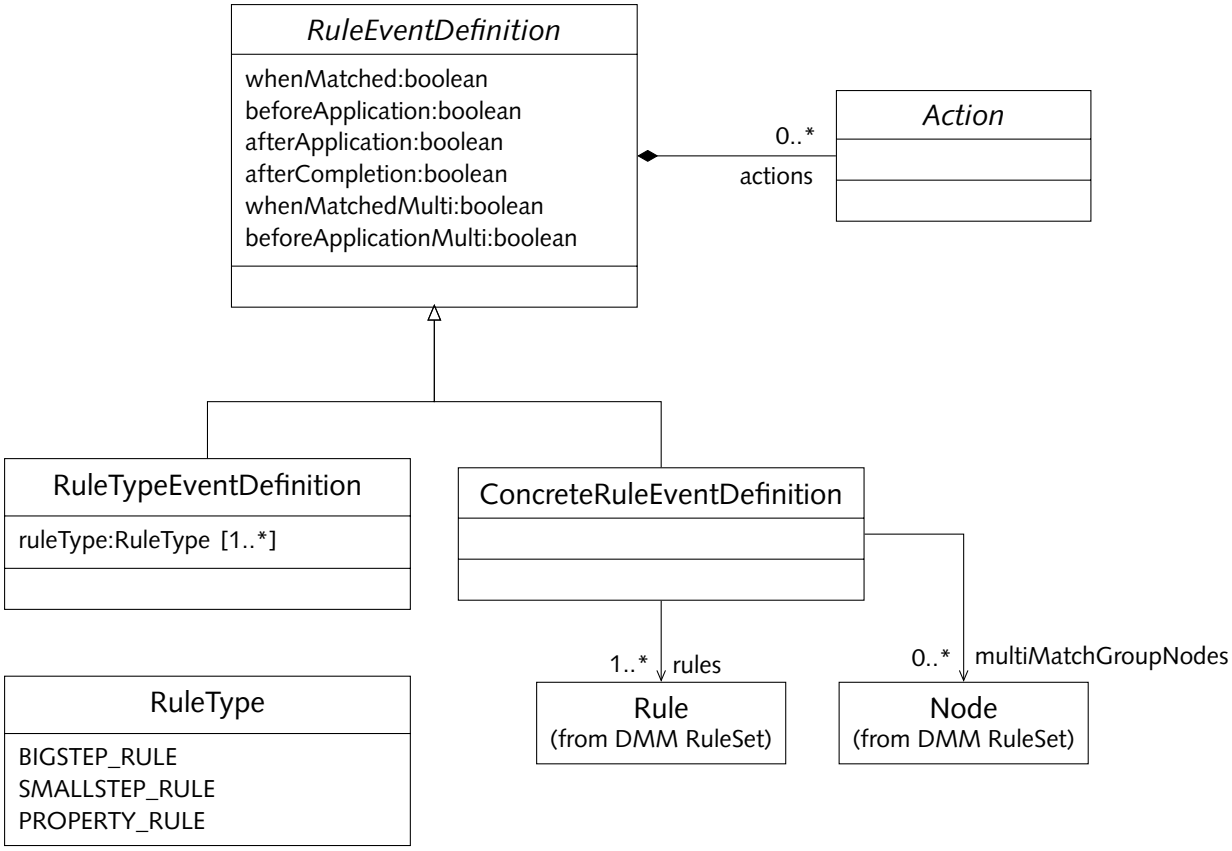

Figure 4.15: The rule event model as class diagram

4.5.2 Mapping other Models to the Rule Event Model

Figure 4.16 shows the mapping of the path switch description model, the step definition model, and the breakpoint model to the new rule event model. The mapping is quite simple, as the step definition model and the breakpoint model are already very similar to the rule event model.

Switch descriptions get mapped to instances of ConcreteRuleEventDefinition with the beforeApplicationMulti attribute set to true. The identificationNodes attribute from the switch definition model is not directly supported in the rule event model. However, it can be specified using a custom action class. When a rule event for a switch instance occurs, the custom action class suspends the execution and uses the stored identification node to find out the model elements for which the "traffic sign" annotation should appear.

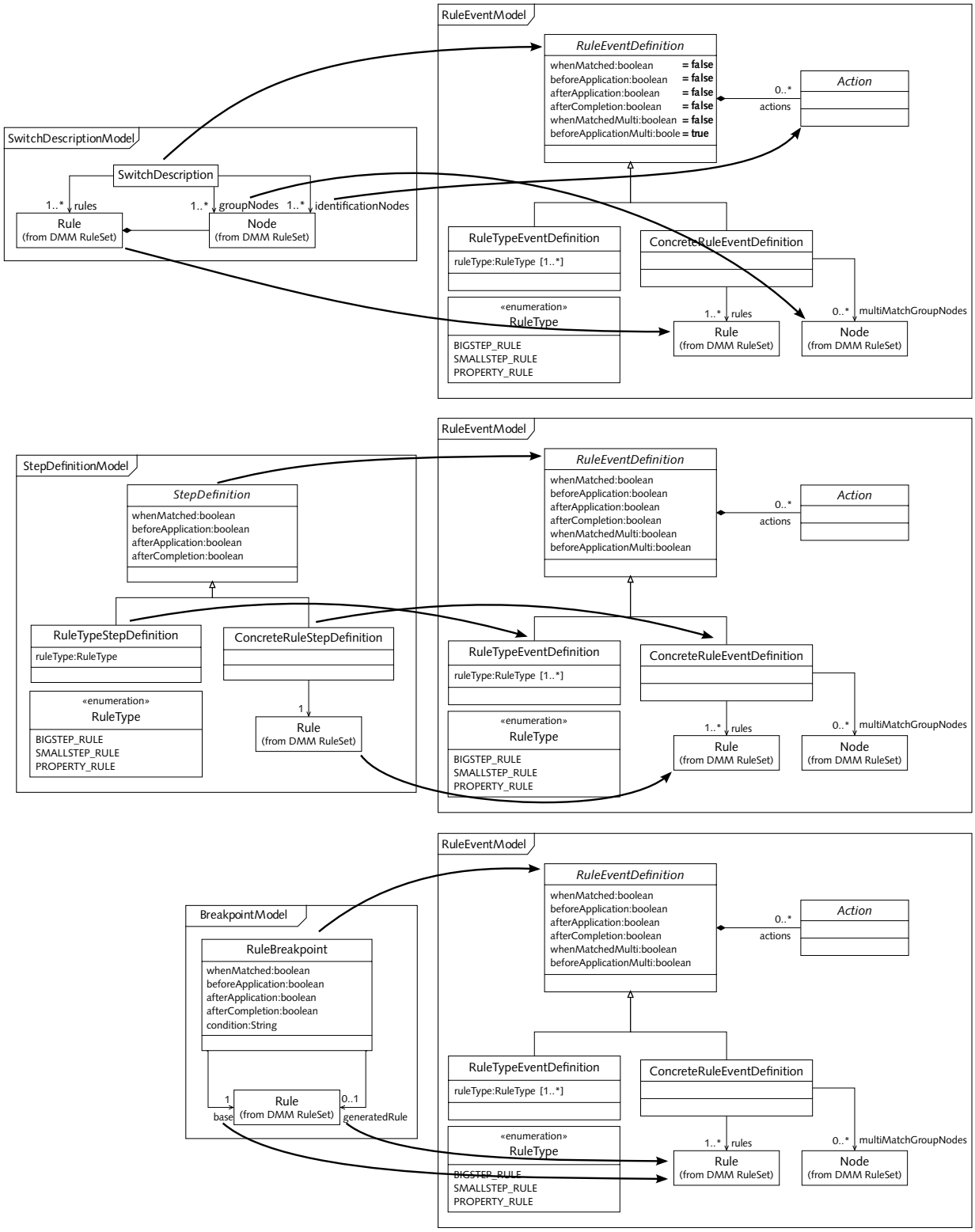

Figure 4.16: Mapping the previously identified models to the rule event model

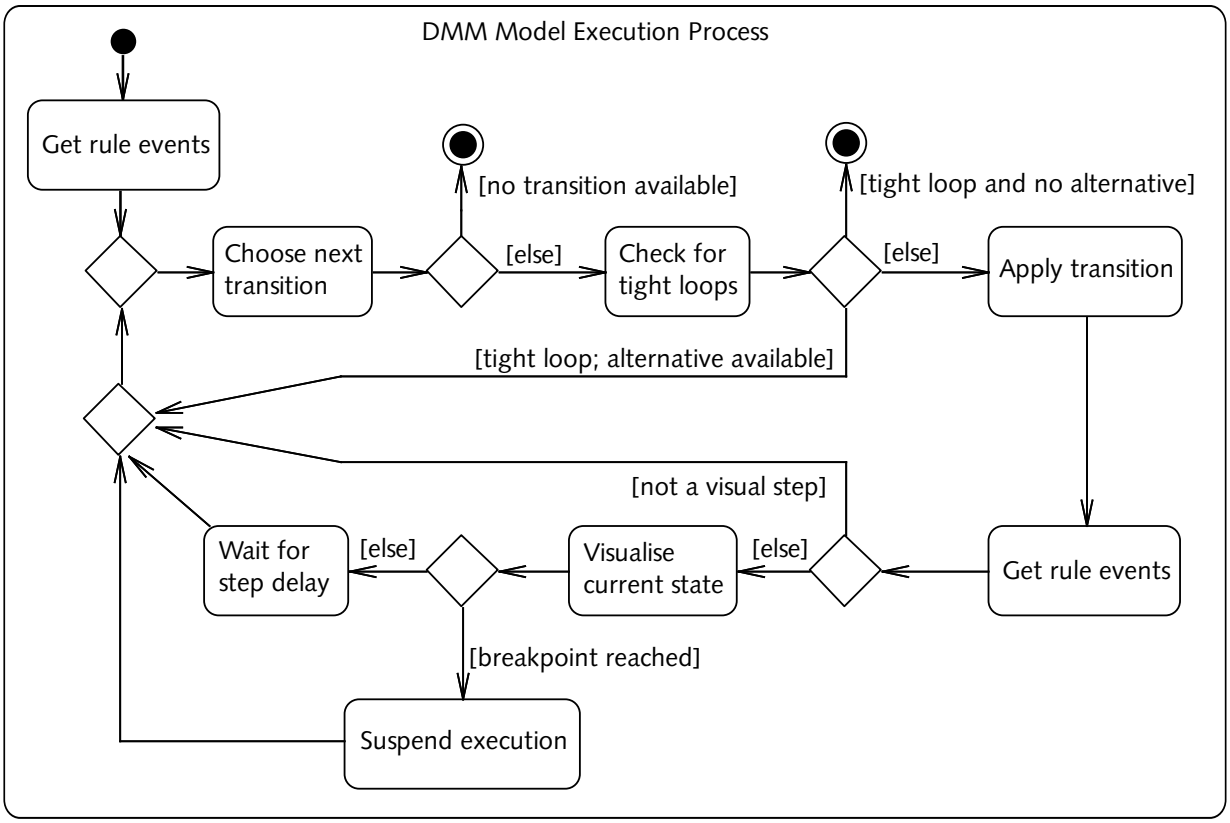

Figure 4.17: The overall model execution process of DMM

4.6 Model Execution Process

We conclude the conceptual analysis and design by composing a comprehensive model execution process using the concepts which have been introduced in the preceding sections. Figure 4.17 shows this process modeled with a UML activity diagram.

Obviously, the process forms a loop which performs a transformation on the model state in each pass. The loop will run until no further transformations are available; this means that either the model will be in a state in which no rule matches or that the matching rules do not modify the state.

Preconditions for this process are the following: There is a DMM semantics specification for the model to be executed. The step definition, switch descriptions, and breakpoints have been converted to rule event descriptions. Furthermore, the model to be executed has already been transformed to its runtime model representation.

We will now go through the actions performed during the process:

The initial action is Get rule events. This action determines the applicable rules on the initial model state and the associated matches. When there are rule event definitions with the whenMatched attribute set, their associated actions will be executed now. Other rule event descriptions will be evaluated later.

The first action in the loop is Choose next transition which employs the algorithm described in section 4.2.4 for choosing the transition to be used and recognising and possibly handling switch instances. For recognising switch instances, rule events with the beforeApplicationMulti flag are used. Thus, these are handled during this action. After a transition was chosen, also the rule events with the beforeApplication flag will be handled. If there is no transformation rule which is applicable on the current state, the execution will be ended.

The next action, Check for tight loops, is necessary because certain combination of DMM semantics and step definitions may cause infinite execution loops which are however not visualised for the user. Normally, executing a model which does not terminate is a valid usage of the DMM Player. The user may watch the looping execution state in the DMM Player and terminate the execution whenever he wants. However, this requires that the visual step definition used during this execution visualises the current model state at least twice in the loop. It may happen—due to defects in the step definition or in the semantics—that this is not the case. Thus, the execution runs in a *tight loop*, while the user only sees a constant model state without knowing why the execution is not finished. In order to avoid such situations, the model execution process needs to watch for tight loops and escape the loop by either choosing another available transition or by terminating the execution. Tight loops may be discovered by memorising all states that have been reached since the last state visualisation. A state that is reached twice signifies the start of a tight loop.

After the check for tight loops, Apply transition finally sets the current model state to the one reached by the transition which was chosen before. Immediately after, the action Get rule events determines—just like the initial action—applicable transitions and rule events resulting from those.

Now it is time to use the new rule events and also the previous afterApplication and afterCompletion rule events to check whether the visual step definition requires the visualisation of the current model state. If this is not the case, this execution pass is finished and the loop continues with the action Choose next transition. Otherwise, the new model state will be visualised in the user interface by the action Visualise current state.

Afterwards, the process checks the rule events for events that represent breakpoints. If a rule event created for a breakpoint is found, the action Suspend execution pauses the process until the user manually continues it. If no breakpoint occurred, the action Wait for step delay will pause the process for the short period of time that is supposed to give the user the chance to follow the execution visually.

This eventually closes the loop. Afterwards the next transition to be used will be picked again in the action Choose next transition.

Software Design 5

We will now advance to the more technical side of the realisation of the DMM Player. This chapter will start with an architectural overview over the DMM Player implementation in section 5.1. Afterwards, some particularly important aspects of the software design and implementation will be covered. Section 5.2 will describe the architecture used for the components that actually enable the DMM Player to execute models. Section 5.3 covers some interesting aspects of the adaption of the external tool GROOVE to the requirements of the DMM Player. Finally, section 5.4 will go into the visualisation of the model states using GMF. All sections will remain on a relatively high abstraction level; more concrete information can be found in the API documentation of the DMM Player which is provided as JavaDoc HTML documents on the disc accompanying this thesis.[1]

5.1 Architectural Overview

We start with the big picture of the software architecture employed for the DMM Player. Figure 5.1 shows a UML component diagram of the components the DMM Player is built of and some related components which are used by the DMM Player.

The architecture plans the creation of seven new components for the realisation of the DMM Player. These components have been marked in the component diagram with a green background; the existing DMM components can be recognized by a yellow background, while other third-party components have white background.

This relatively high number of new components is the result of a strong *separation of concerns* [Dij82] of the components. This decreases the coupling of single components to other components and facilitates the re-usability of components in other software configurations.

An example for optimal decoupling is the GMF Diagram Augmentation component which allows the extension of existing GMF-based diagram editors by the additional information contained in runtime models. This component does not have any dependency on the other components of the DMM Player. The communication between the GMF Diagram Augmentation component and the other DMM Player components occurs exclusively via the EMF models. The interfaces of EMF provide a very central communication hub, as the usages of

[1] Cf. appendix B on page 117.

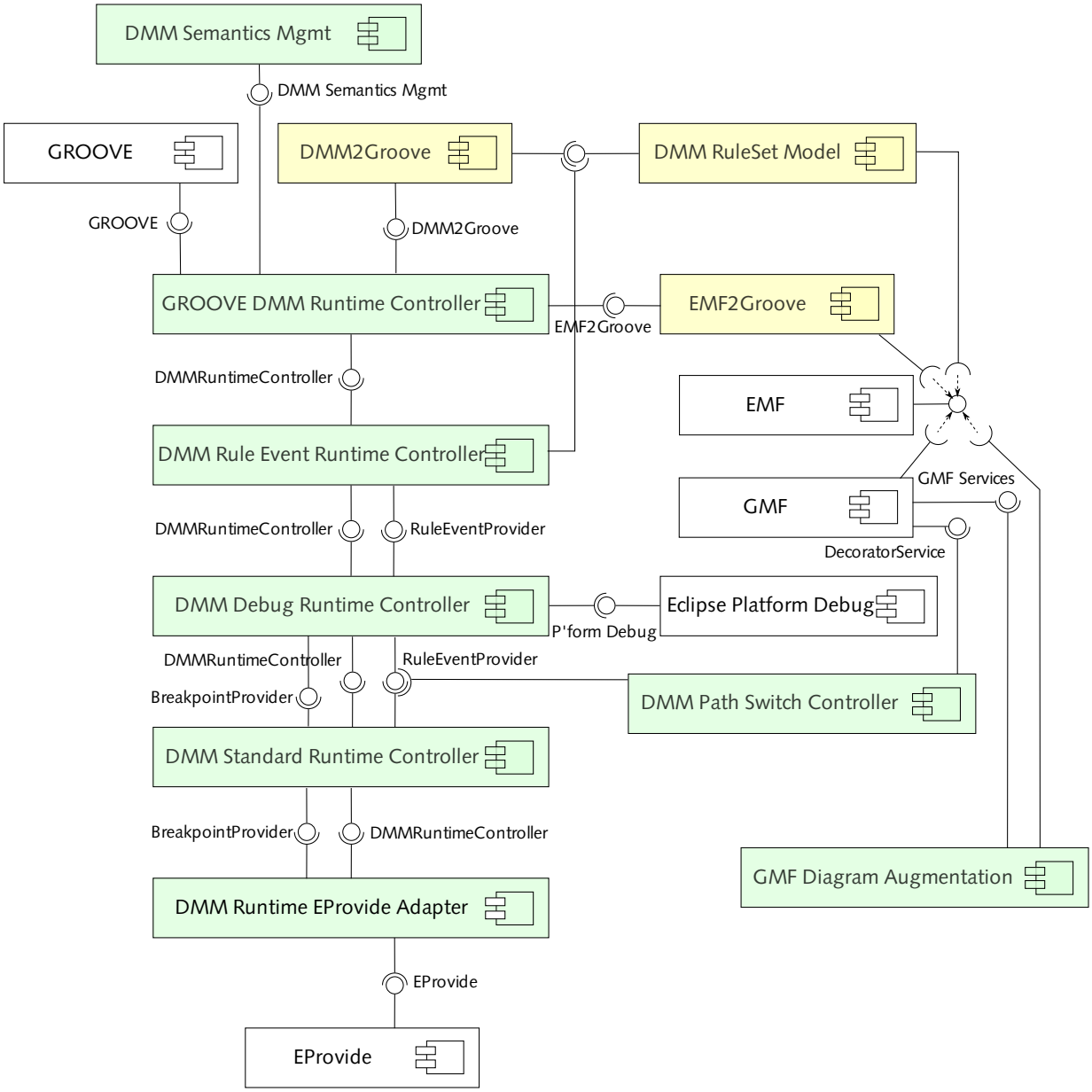

Figure 5.1: UML component diagram of the DMM Player and related components

the Ecore interface in the component diagram illustrate. Section 5.4.2 will provide more information about the diagram augmentation component and its communication to the rest of the DMM Player.

Beneath visualising the runtime model, the other major task of the DMM Player is the actual execution of a model; this involves a wide range of tasks, beginning with interfacing to GROOVE, combining transformations to visual steps, and watching for the occurrence of breakpoints and execution path switches. These tasks are reflected in the architecture by dedicated components. This decomposition helps to keep the dependencies each component has to external components such as GROOVE or the Eclipse Platform Debug framework on the lowest possible level. The decoupling is primarily reached with two design patterns. The now following section will describe the structures and methods employed for realising this group of components, which we will call the *DMM runtime*.

5.2 DMM Runtime

As mentioned before, the tasks of the DMM runtime components are quite heterogeneous; one important and complex task is the communication with GROOVE which performs the actual graph transformations. As GROOVE is a general purpose tool, lots of adaption work needs to be done in this context. Then, there are the tasks that are specific to a UI-based debugger such as breakpoints or the execution path switches. It is desirable that these tasks are performed in separated components; thus, they may be used independently. The Groove DMM Runtime component which interfaces to GROOVE may be also useful in DMM applications other than the DMM Player. Also, it is imaginable that components other than GROOVE could be employed to do the "hard work" of graph transformations for DMM.[2] Thus, it should be possible to exchange the Groove DMM Runtime component by another one without needing to adjust the other components.

This means that there is a need of decoupling the components in both directions. The Groove DMM Runtime component should be usable without the other DMM runtime components which again should be also usable with graph transformation components other than the Groove DMM Runtime.

5.2.1 Streaming the Model State

In order to get an idea of how to decouple these components, we need to look at the kind of information that is exchanged between them. Then, we can define abstract interfaces between the components and choose a suitable architecture that integrates them.

For an easier distinction of the components, we will now call the component that does the actual graph transformations—i.e., currently the Groove DMM Runtime component—the *graph*

[2]Even if the idea of employing components other than GROOVE for the graph transformation seems to be remote at this point of time, it is still good practise to consider as much eventualities as possible. Once in a time, the Y2k also seemed to be remote.

transformation component. Further components processing the results from this component will be called the *client components*.

The main results of graph transformation components are model states; when the graph transformation component has been initialised with a model and the respective DMM semantics, it can directly produce a new model state, and directly afterwards another new model state—of course provided that there are applicable rules. The clients just need to pull the model states. Possibly, they might want to control which transition is going to be followed next by the graph transformation component. Thus, the main information flow between the components can be interpreted as a stream of model states that is pulled from the graph transformation component by the client components.

The client components may want to monitor the states for breakpoints, switch instances or similar concepts. Furthermore, the concept of visualisation steps—which shows only certain states to the user—actually modifies the stream from the graph transformation component.

An architectural solution for such a situation is the *Pipes and Filters* design pattern [BMR+96].[3] The Pipes and Filters pattern is supposed to decouple tasks that are involved in the processing of a data stream. Structurally, the pattern is a chain of a data source connected with a number of filters which is finally connected to a data sink. The Pipes and Filters pattern can be both used with a data sink and filters that pull the stream, i.e., a passive data source, and with an active data source that pushes the stream to the filters and the data sink. In the case of the former variant—which is also called a *pull pipeline*—both the data source and the filters provide the same interface. Thus, from the outside data sources look the same and become interchangeable.

The DMM Player architecture uses this pattern for decoupling the graph transformation component, i.e., the Groove DMM Runtime Controller component, from its clients. This can be seen in the component diagram in figure 5.1; as the Groove DMM Runtime is the data source, it is the start of a chain of components, each linked by the DmmRuntimeController interface. The components in the middle of the chain which both require and provide the DmmRuntime-Controller interface are the filters that may monitor and modify the stream. The sink in this constellation is the DMM Runtime EProvide Adapter which adapts the communication to the interfaces of the external component EProvide.

The connections between the source, the filters, and the sink are—in terms of the Pipes and Filters pattern description—the pipes. The complexity of the pipes may vary from simple method calls to buffering functions; for the purpose of the DMM Player, method calls on the interfaces are sufficient.[4]

[3]Sometimes this pattern is just called the *Filter* pattern [Gra98]. Pipes are regarded as a variant of the Filter in this case. The original pattern description [BMR+96] also describes variants, which are however always called Pipes and Filters.

[4]However, a kind of state pre-fetching and buffering may be be also suitable for the DMM Player. Currently, the DMM Player idles during the delay between steps. A special pipe could asynchronously calculate the reachable states which could be directly used when the step delay is finished. As pipes can be realised as special filters, such a component would be realisable without having to modify the other components.

5.2.2 The GROOVE DMM Runtime Controller

This component adapts the DMM rule set, the EMF model and the commands received by the DmmRuntimeController interface to the interfaces and data structures used by GROOVE. As it only provides the DmmRuntimeController interface, but does not require it, this is the source of the stream as described by the Pipes and Filters pattern.

As adapting DMM to GROOVE is a quite complex matter, the measures required for this will be covered more in depth in section 5.3 below.

5.2.3 The DMM Rule Event Runtime Controller

The most important filter in the chain is the DMM Rule Event Runtime Controller component. The component can be configured using a further interface, the RuleEventProvider interface, to generate rule events which trigger operations defined by its clients. Thus, this interface realises an *Observer* design pattern [GHJV95] which also provides means of decoupling components. This component also provides a functionality for restricting the streamed model states to those that trigger certain configured rule events. Thus, the visual step configuration can be transformed into rule events and used for the configuration of the rule event runtime. The client just needs to visualise the states returned from DMM Rule Event Runtime. As there might be no states that match the criteria for the visual step, the DMM Rule Event Runtime realises the check for tight loops which has been described in section 4.6.

5.2.4 The DMM Debug Runtime Controller

The DMM Debug Runtime is a further filter component that uses and provides the DMMRuntimeController interface. However, it will not modify the model states. Rather, it uses the further RuleEventProvider interface to install rule event descriptions on the DMM Rule Event Runtime component. These rule event descriptions realise the DMM rule breakpoints that are installed in the current Eclipse workbench. DMM Debug Runtime retrieves these from the external Eclipse Platform Debug component.[5] The DMM Debug Runtime provides three interfaces; the DMMRuntimeController and RuleEventProvider interfaces are delegated to the respective required interfaces. The BreakpointProvider interface allows clients to get notified when a breakpoint was reached and thus the execution should be suspended. As the DMM runtime follows a pull pipeline approach, the data sink needs to be actively instructed to stop pulling when a breakpoint was reached. Thus, this additional interface is required.

5.2.5 The DMM Path Switch Controller

This component uses rule events to watch for switch instances and visualises them in the respective GMF diagrams.[6] To be able to define rule events, the component requires the RuleEventProvider interface. As the visualisation is done with GMF diagram decorators, there is of course

[5] Cf. section 2.3.4 on page 25.
[6] Cf. section 3.6.2 on page 49.

also a dependency on the corresponding interface provided by GMF. This component does not realise the filter pattern, as it does not need to modify the stream of states; it becomes only activated by the callbacks caused by the rule event observers.

5.2.6 The DMM Standard Runtime Controller

The task of this component is merely building up the standard component configuration used by the DMM Player. It provides the DMMRuntimeController and BreakpointProvider interfaces which are completely delegated to the respective required interfaces. It thus acts as a simple facade that conceals the actual component structure from its client.

5.2.7 The DMM Runtime EProvide Adapter

This component represents the sink of the filter pattern, as no further filters can be configured behind of it. However, it is not the end of the component chain because it just adapts the DMM runtime interfaces to the interfaces used by EProvide.[7]

EProvide will repeatedly call a method on the adapter component that signals that the next step shall be executed. The adapter component will use the interface of the filter chain to request the next step. Thus, EProvide will indirectly perform the polling of the filter chain.

5.3 Interfacing to GROOVE

In this section we will look at the communication between the DMM Player and GROOVE. Figure 5.2 shows the actions related to GROOVE which are performed during a model execution. It should be noted that the activity diagram shows a conceptual view of the process which matches the actual action sequence, but abstracts from the actual performer of the action. Thus, some actions are actually initiated by calls from outside the Groove DMM Runtime Controller. We will now look briefly through the actions:

Before the actual execution starts, the DMM rule set needs to be transformed into a GROOVE grammar. This is handled by the existing DMM2Groove component, which produces a set of GXL files containing the GROOVE grammar which can be loaded with GROOVE. The EMF2Groove action and equally named component does a similar task for EMF models which are converted to GROOVE states represented in GXL files as well. After GROOVE has been initialised with the transformed models, the matches of the current state are retrieved in the action Get Matches. Of course, these matches are morphisms between the nodes of the matching GROOVE rules and the GROOVE state. For further processing in the DMM Player—which is for example required for recognising switch instances—the Groove DMM Runtime Controller needs to convert these matches into a DMM compatible format. More on this can be found in section 5.3.1.

The next step is to actually perform the model transformation. Depending on the situation, the DMM Player will have arbitrarily chosen a match for the transformation or have let the

[7] Cf. section 2.4.2 on page 26.

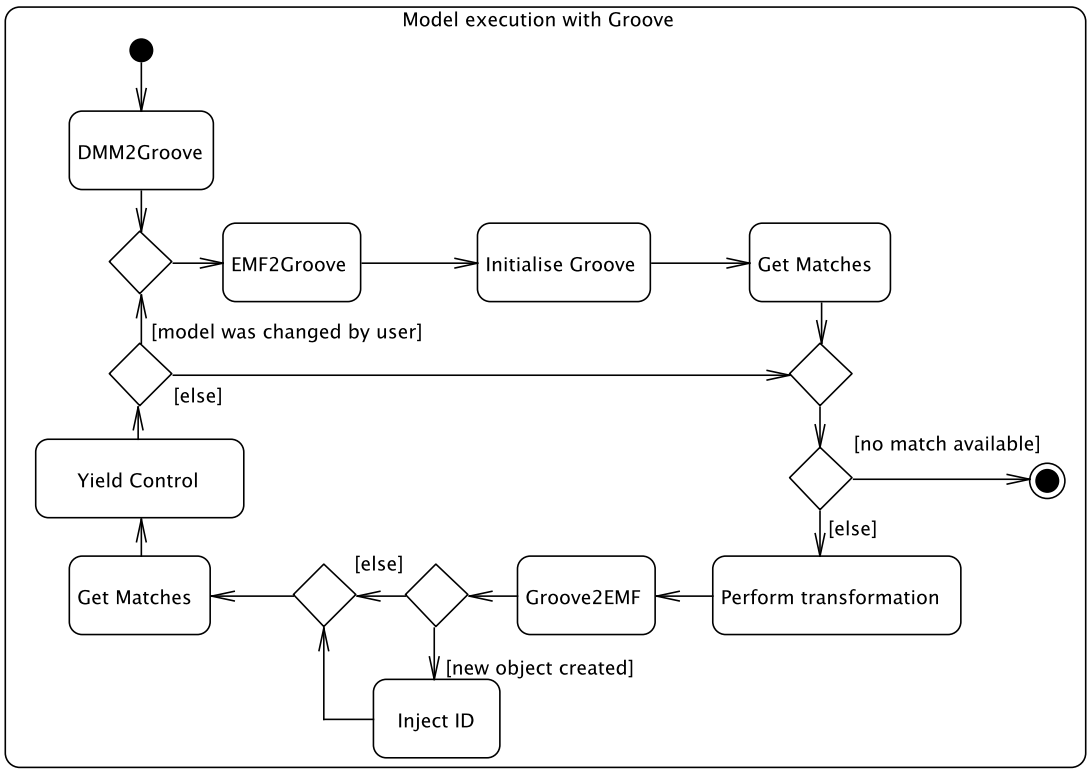

Figure 5.2: Model execution with GROOVE

user choose one. After this, the GROOVE state will be transformed back into the EMF model. When the transformation notices that a new object has been created by GROOVE, an identifier needs to be assigned to the object both on the EMF and the GROOVE side. More information on the transformation and the IDs will be provided in section 5.3.2.

The action Yield Control signifies the point where the DMM Player may suspend the execution for breakpoints or switch instances. During this time, the user may modify the state of the model.[8] If the user did change the model, a transformation from the EMF model to the GROOVE state is necessary again, thus the upper branch of the activity needs to be used. If no change was performed, the DMM Player skips the transformation, initialisation and matching actions.

This closes the loop. The following sections will now look closer on two actions performed in the activity; first, the handling of identities during the EMF model transformations will be discussed. The topic of the subsequent section 5.3.2 will be the adaption of GROOVE matches to DMM matches.

[8]Modifying the program state is a common feature of debuggers; cf. section 2.3.2 on page 22.

5.3.1 Preserving Object Identities in EMF2Groove

In object oriented environments, objects can be typically compared in two ways: Two objects can be tested for equality; if two objects are equal, they have no different properties, but they may actually be two separate objects. If two objects are said to be identical, then one talks actually about only one object.

The EMF2Groove component [Rhe07] provides means for transforming EMF models to GROOVE states and for transforming GROOVE states back again to EMF models. The EMF model resulting from a transformation cycle will be however completely re-built from scratch. All objects will be newly created according to the GROOVE state, even if the objects already existed before the transformation cycle and were not changed.

This behaviour is fine as long the identities do not need to be kept. However, when GMF is used to create a diagram for an EMF-based model, the association between the concrete syntax diagram elements and the underlying model elements will be established by object identity. Thus, if the GMF diagram shall catch the changes performed by GROOVE, the backward transformation to the EMF model needs to keep object identities.

In order to do this, the backward transformation must not re-build the model, but it needs to find out how GROOVE modified the state and then modify the existing EMF model accordingly. This can be reached by comparing the GROOVE state and the EMF model; first, the nodes from the GROOVE state need to be associated with their corresponding objects from the EMF model. EMF objects without a GROOVE counterpart have been deleted by GROOVE and also need to be deleted on the EMF side; likewise, GROOVE nodes without a EMF counterpart indicate a newly created object which also needs to be created in the EMF model. After objects have been handled, a similar algorithm is performed for reference values and attributes.

An important requirement of this algorithm is being able to associate GROOVE nodes with the corresponding EMF objects. However, the original EMF2Groove transformation did not include any identity information in the transformed GROOVE state. Thus, it was impossible to make this association. For the DMM Player, the transformation from EMF to GROOVE states has been extended in such a way, that the EMF2Groove controller class generates numerical IDs that are unique to the particular model and assigns the IDs both to the EMF objects and the GROOVE nodes. For the EMF objects, the association is realised with a Java HashMap data structure in the EMF2Groove class. The GROOVE nodes get a further label that contains the ID value.

Nodes which are created by a GROOVE transformation, have no ID label, though. This is no problem during the first transformation back to EMF. The transformation code will note the missing ID string and thus conclude that the object has been newly created.

However, the DMM Player will not perform clean transformation cycles from EMF to GROOVE to EMF to GROOVE to EMF, etc. Rather, the DMM Player will do only in the beginning a transformation from EMF to GROOVE. Afterwards, it will normally do just GROOVE to EMF transformations, as it can be seen in figure 5.2.

This means that newly created objects will be missing the ID string in later cycles as well if no special measure is taken. Thus, the GROOVE to EMF would recognise these objects as new ones in every cycle and thus create new EMF representations for them. The easiest

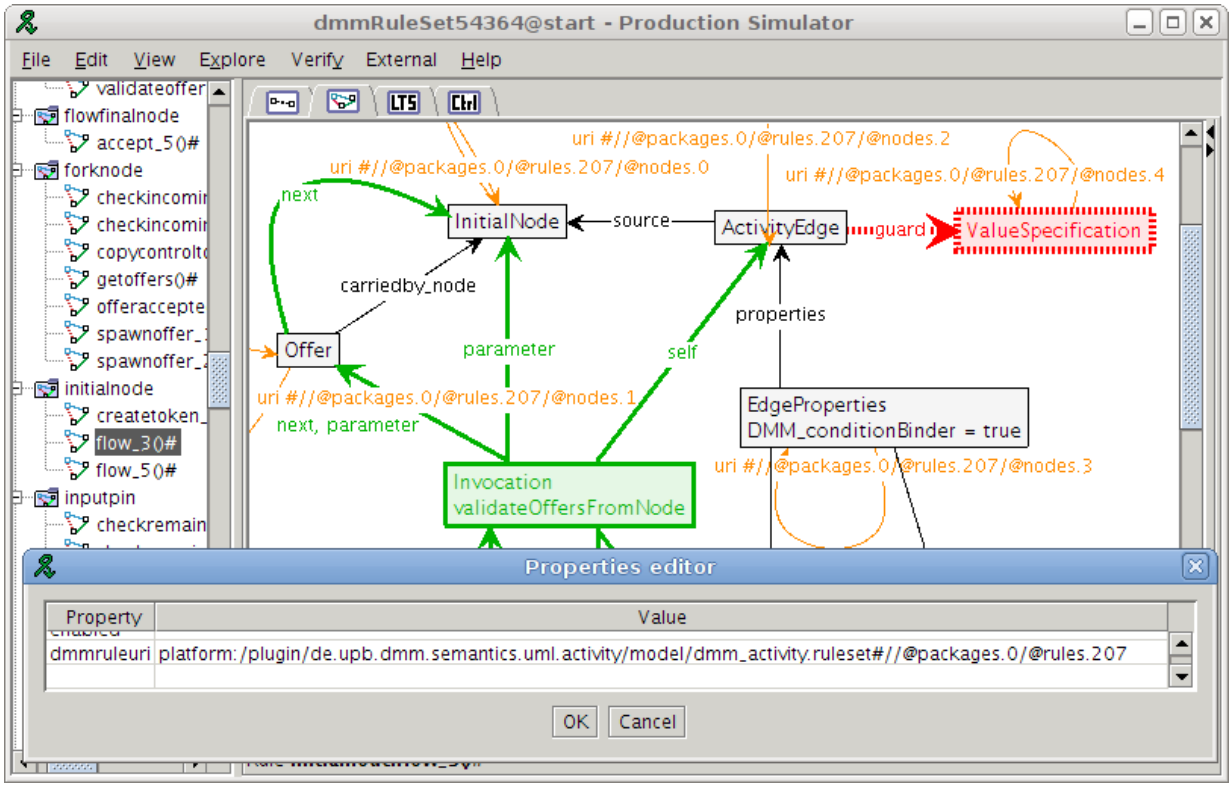

Figure 5.3: A GROOVE rule with identifiers as comment edges

solution to this problem would be to perform an EMF to GROOVE transformation in such cases. However, this transformations require a considerable amount of processing time and thus slow down the execution notably.

Instead, the DMM Player will directly modify the GROOVE runtime representation of its current state and add the new ID to the labels of the respective node.

5.3.2 Adaption of GROOVE Matches to DMM

In GROOVE, matches are a morphism from the nodes of a certain rule to nodes from the current state. For DMM, we have introduced the concept of rule events,[9] which is intended to process matches and morphisms in order to support breakpoints and similar features in the DMM Player. As rule events are a DMM concept, they obviously use DMM rules and rule nodes as criteria. Thus, in this case the DMM Player also needs some kind of translation from GROOVE rules and their nodes to their DMM counterparts.

Both rules and nodes have no existing property which could be used as identifier that associates them on the DMM and GROOVE sides. Thus, we need to find a suitable identifier and attach it to the constructs on both sides.

[9] Cf. section 4.5 on page 76

EMF supports the identification of all models and model elements by special *Uniform Resource Identifiers* (URI). The EMF URIs are based on the URI internet standard [BLFM98]. As DMM rules and nodes are elements of EMF models, the EMF URIs are well suited for identifying DMM elements in their GROOVE counterparts.

Thus, the DMM2Groove component was modified to include these URIs in the GROOVE grammars. The URI of a DMM rule is saved as a GROOVE rule property with the name dmm-ruleuri. The URIs of DMM nodes are attached as comment edges to the GROOVE nodes. For brevity, these URIs are denoted relatively to the URI of the containing rule. These are edges with a label prefixed with the string rem:. Figure 5.3 shows how such a rule set looks when it is opened in the standalone GROOVE application; the property editor in the foreground shows the absolute URI of the DMM rule. The orange edges in the graph are the relative URIs of the particular DMM nodes.

5.4 Model Visualisation with GMF

The previous sections and chapters often mentioned that "the current model state will be visualised" in certain situations. However, it has not been cleared yet how this visualisation is realised. The following section 5.4.1 will describe how GMF diagram editors can visualise the changes automatically performed by the DMM model execution at all. Section 5.4.2 describes the technique that was developed for this thesis in order to augment existing diagram editors—such as the UML activity diagram editor—by the runtime information defined by DMM.

5.4.1 GMF and EMF Integration Basics

As already mentioned, GMF has been specially designed for editing EMF models in graphical diagram editors.[10] Thus, GMF editors can be integrated with further components that operate on EMF models using just EMF as the interface between GMF and the other components. Provided that a GMF diagram editor and another component use the same *editing domain* for loading an EMF model, all changes to the model performed by the further component are immediately visualised in the GMF diagram editor. Likewise, changes in the diagram editor are also directly available in the other component.

Furthermore, EMF offers transaction support for modifications on the model. If a component starts a transaction, all modifications done by this component on a particular model will not be visible to other components until the transaction has been committed.

The DMM Player uses these features to visualise the changes resulting from a model execution in a GMF diagram editor. Before a model execution is started, the DMM Player—or more specifically EProvide which is used by the DMM Player for this task—loads the EMF model in an editing domain which is also used by a diagram editor. Furthermore, EProvide starts an EMF transaction before each execution step and commits it afterwards.

This means that the code that does the actual model execution just needs to perform changes on the EMF model; when the step has completed the changes are visualised in the GMF editors

[10] Cf. section 2.4.1 on page 25.

at once. This deferring of changes is also used by the DMM Player for limiting the visualised states to those that are supposed to be displayed due to the visual step configuration.

5.4.2 GMF Diagram Augmentation

We have seen that due to the tight integration of GMF and EMF, the DMM Player can be used to visualise the execution of a particular language with an existing GMF diagram editor for that language.

A requirement is of course that the diagram editor supports the dynamic elements of the language. A key principle of DMM is however to add dynamic elements to the language by means of the runtime meta model. Thus, the dynamic elements were not part of the original meta model before and thus will not be supported by existing diagram editors.

We can again use the UML activity diagrams as example. The Eclipse UML2Tools project has developed a GMF based diagram editor for UML activities. However, it cannot be used for the DMM Player, as the editor does not know anything about dynamic elements such as tokens or offers.

Fortunately, the GMF architecture provides a number of extension and configuration mechanisms that can be used for injecting the dynamic information from a DMM runtime meta model into existing GMF diagram editors. The goal is to provide a completely declarative way of realising the extended editors. It shall be possible to use a model to define what information from a runtime model will be visualised, where, and how it will be visualised. We will call this approach *diagram augmentation*.

5.4.2.1 GMF Diagram Editor Architecture

Before learning about the actual realisation of the diagram augmentation, we need to look a bit closer at the architecture used by GMF for realising diagram editors.

Each GMF editor uses two model layers and a further UI layer; figure 5.4 shows an illustration of this architecture, figure 5.5 shows an class diagram of the most important classes involved in it. At the lowest layer is the actual model to be edited in its abstract syntax representation. GMF calls this model the *semantic model*. In the class diagram, elements from the semantic model are represented by the class EObject.

The next layer is the *view model* which is an EMF model representing the graphical structure of the concrete syntax of the underlying model. Thus, the most commonly occurring objects in this model will have the type Node and Edge.[11] The elements from both models are mapped to each other; however, there may also be elements from the semantic model without a view representation or view model elements that have no semantic counterpart.

The third level is the actual UI representation of the view model. Each element from the view model is associated with a so-called EditPart object, which defines its appearance and UI

[11] In this case, Node and Edge are classes from the view meta model provided by GMF. Thus, they do not have anything to do with the nodes and edges used by DMM rules or GROOVE.

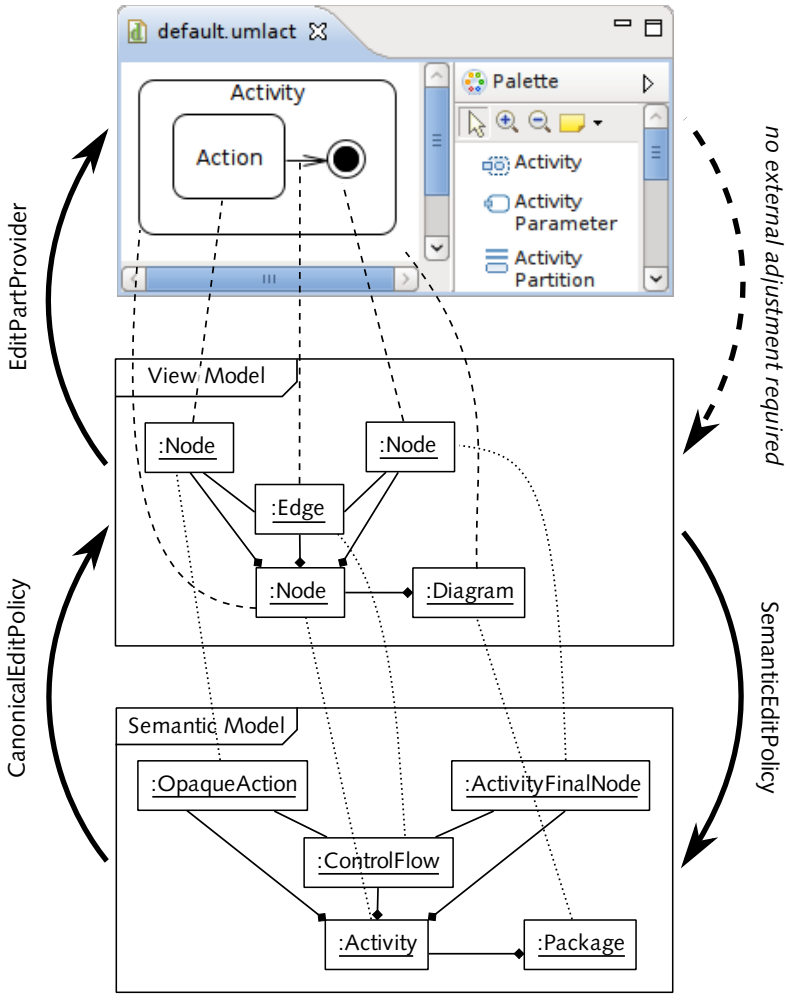

Figure 5.4: Models, policies, and providers used by GMF

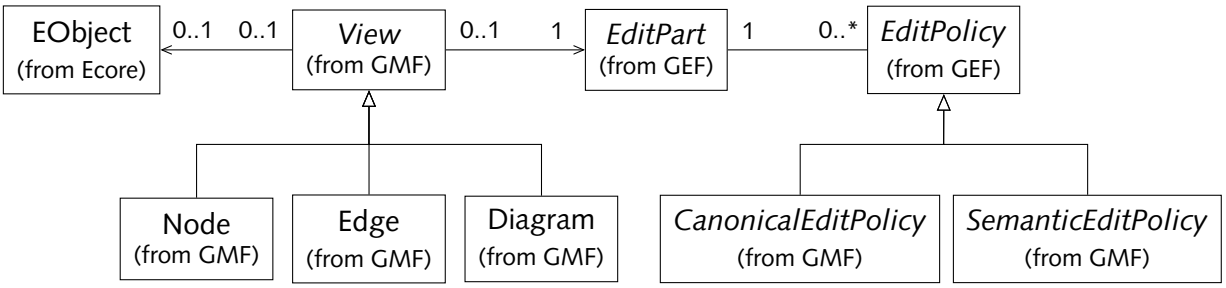

Figure 5.5: Important classes involved in the realisation of GMF diagrams

behaviour.[12]

When elements are created, changed or deleted either in the view model (by means of the edit parts) or in the semantic model (by any other means), the respective other model needs to be updated accordingly. This is managed by edit policy classes. The CanonicalEditPolicy watches for changes in the semantic model and creates, deletes or modifies objects in the view model accordingly. The job of the SemanticEditPolicy is to react on create or delete element commands issued by the user and to perform the requested changes in the semantic model.

Edit policies are attached to edit parts and are organised in a *chain of responsibility* pattern. This means that the edit policies define by themselves whether they support a certain operation or not. If they do not support a requested operation, that operation might get performed by another edit policy in the chain which supports the operation. By default, each edit part defines its own edit policies. However, GMF provides the possibility to attach further edit policies with edit policy providers. These are special classes that are invoked whenever a new edit part is created somewhere in the application. The edit policy provider may then decide whether it wants to attach a further edit policy to the edit part.

The edit parts that are created for the elements of the view model are determined by a further class of providers: the EditPartProvider class. Again, EditPartProviders are organised in a chain of responsibility and receive requests to create edit parts for certain view model elements. Then they may examine these elements and possibly create an edit part for the model elements.[13]

The semantic model and view model are persistently saved; the edit parts however are pure runtime objects which get re-created by the EditPartProviders whenever a diagram is opened. Changes in an opened model are always performed first in the view model and the semantic model and then propagated to the edit part layer. Thus, there is no need for a bidirectional adjustment mechanism between the edit parts and the view model as opposed to the adjustment performed by the CanonicalEditPolicy and the SemanticEditPolicy between the view model and the semantic model.

5.4.2.2 Diagram Augmentation Basics

As seen before, the GMF architecture provides versatile extension mechanisms: EditPolicyProviders may install additional edit policies on any diagram edit part they choose. The providers operate in a chain of responsibility, so that existing behaviour may be exchanged or extended. EditPartProviders can be used to exchange edit parts, and thus the appearance of diagram elements.

These extension mechanisms make the augmentation of existing GMF editors by additional information from DMM runtime models relatively easy. There is however one important

[12] EditParts are provided by the *Graphical Editing Framework* (GEF), which GMF bases on. The main addition by GMF to GEF is the integration of EMF.

[13] It may sound irritating that edit parts are created for existing view model elements while the CanonicalEditPolicy—which is tied to an edit part—creates view model elements. This chicken or egg dilemma is solved by the fact that there is always one root element in the view model, the Diagram element. The CanonicalEditPolicy now creates the view model elements that are going to be the children of the element that belongs to the edit part the policy is tied to.

restriction: The existing diagram elements can only refer to the meta model they have been created for. There is no way to map them to equivalent model elements from a runtime model which was transformed from the syntactic model. Thus, the runtime meta model must be created using the method that does not alter the syntactic meta model; all associations between both meta models will be created as unidirectional references leading from the runtime to the syntactic meta model.[14]

In the visualisation, runtime model elements will be tied to elements from the syntactic model, though. We have already seen this in the various depictions of runtime states of a UML activity in this thesis: Tokens are always attached to a node; offers are attached to either a node or an edge. The same holds for the behaviour execution nodes with the label EX which are attached to activities or actions. Elements from the runtime model may also reference each other or define further sub-elements, though. A simple case of this is the edge that is visualised between a token and an offer which is the representation of the base association between the classes in the runtime meta model.

5.4.2.3 Augmenting the View Model

To realise the diagram augmentation in GMF, we need to represent the elements from the DMM runtime model as additional nodes in the GMF view model. These nodes will be the children of the nodes representing the syntactic model elements they are attached to; this allows the runtime elements to be placed visually into or attached to their parent elements from the syntactic model. The GMF semantic model does not need to be changed. In fact, semantic model is just a name for the meta model instace which is visualised. The combination of two meta models such as the syntactic meta model and the runtime meta model poses no problem, as one GMF diagram may contain elements from different meta models.

As explained before, the mapping from the semantic model to the view model and vice-versa is done by the CanonicalEditPolicy and the SemanticEditPolicy. Each edit part, i.e., each model element in the diagram has its own instances of these policies. The function of the policies is to synchronise the visual children of the edit part they are bound to between the view model and the semantic model. Thus, in order to extend the view model by nodes for runtime model elements referencing an element from the syntactic model, the CanonicalEditPolicy belonging to the element from the syntactic model needs to be altered.

The new CanonicalEditPolicy[15] needs to search the runtime model for the model elements belonging to the respective syntactic model element. Then, it creates view nodes for these elements and adds them as children the view node the policy is attached to. However, it is not sufficient to simply place the new CanonicalEditPolicy in front of the old one in the chain of responsibility managing the edit policies. GMF only uses the first edit policy that signals that it is able to perform the requested task. However, an augmented model element already may have child nodes from the original syntactic model. These must be respected by the new

[14] Cf. section 2.1.2 on page 9 and figure 2.4 on page 10.

[15] In the GMF diagram augmentation implementation, the new edit policy is implemented in the class de.upb.dmm.gmf.augmentation.diagram.edit.policies.AugmentedObjectCanonicalEditPolicy. The package name follows the style used by GMF diagram editors.

CanonicalEditPolicy; otherwise, they would get lost in the diagram. To do so, the DMM Player will search for the existing CanonicalEditPolicy before installing the new one. Then, the new edit policy will be initialised with a reference to the old one; thus, the new edit policy can query the old one for the original view nodes and add the augmenting view nodes to the result.

Thus, with the new CanonicalEditPolicy we are able to build and update augmented GMF view models from DMM runtime models. This is required when a new diagram is newly created for an existing model and when an external component—such as the DMM Player—modifies the model. The other way round becomes important when the user modifies the model using the diagram editor. Then, the SemanticEditPolicy adjusts the GMF semantic model according to the modifications in the view model. A SemanticEditPolicy for an augmented diagram is therefore only required if the user shall be able to manually add runtime information to the model.

Like all edit policies, a SemanticEditPolicy is tied to a view element by an edit part. It handles the creation and deletion of the view's child elements. An augmenting SemanticEditPolicy[16] now needs to intercept creation and deletion requests for runtime model elements. When a user wants to create a runtime element, the new SemanticEditPolicy needs to create the new element, find out its actual parent element in the runtime model, add it to that parent element and set the reference that associates the runtime element with the element from the syntactic model it is attached to. The deletion of a runtime element works analogously to a large extent. The most important difference to normal SemanticEditPolicy implementations is that those usually only need to set the parent element of a newly created element. Other references are set later by further user manipulations.

The actual implementation of the view model augmentation involves a number of further tasks such as listening for change events and processing the runtime model in such a way that the lookups of runtime model elements associated to particular elements from the syntactic model become efficient. We will not further cover these topics at this place, though.

5.4.2.4 Augmenting the Edit Parts

The view model is still an abstract model of the diagram structure. It does not define the shape of its elements or UI operations on them. This is done by the subclasses of EditPart. In order to be able to attach runtime elements to their associated elements from the syntactic model, the EditParts of the latter elements need to be extended in such a way that they include runtime elements such as tokens and layout them on their border. Unfortunately, the extension mechanisms of existing EditParts are quite limited in this respect. EditPartProviders allow exchanging existing edit part implementations by completely new ones. This is however not feasible as the appearance of the augmented element needs to be kept. Thus, the existing edit part implementations need to be extended. This is only possible by creating subclasses of the existing edit part classes. However, the GMF diagram augmentation is supposed to provide a declarative way of

[16] In the GMF diagram augmentation implementation, the new edit policy is implemented in the class de.upb.dmm.gmf.augmentation.diagram.edit.policies.AugmentedObjectSemanticEditPolicy.

```
<?xml version="1.0" encoding="UTF-8"?>
<notation:Diagram [...] type=" UMLActivity " [...]>
    [...]
</notation:Diagram>
```

Figure 5.6: Finding out the diagram type attribute of a particular GMF diagram editor using its diagram files

creating augmented diagrams which involves no programming.[17] To reach this goal, the GMF diagram augmentation implementation needs to employ a trick—which might be regarded by some as slightly dirty. The new augmenting EditPartProvider[18] is initialised with a reference to the original EditPartProvider. When the new provider is then requested to create an edit part for an augmented element, it will first use the original provider to find out the class of the original edit part. Then, the new provider uses the *Java Compiler Interface* library [JCI09] to dynamically create a subclass of the original edit part. The new class will then be able to render runtime information as required.

5.4.2.5 Diagram Augmentation Models

The preceding sections covered internal aspects of the GMF Diagram Augmentation component. Yet, we still need an interface that allows one to use the component by defining a diagram augmentation for a particular GMF diagram editor. Those augmentations can be configured using a diagram augmentation model. The corresponding meta model can be seen in figure 5.7.

Before going through the elements of the meta model, we will define some terms: The model elements representable by an existing model editor to which new model elements are supposed to be attached to are *augmented elements*. The model elements which should be newly supported by the model editor will be called *augmenting elements*. Finally, the elements from the augmentation model that describe how diagram editors should be augmented are *augmentation elements*.

The base of each augmentation model is the DiagramAugmentationModel class. Its only attribute, diagramType serves for identifying the diagram editor this model has been created for. The value of this attribute must correspond to the diagram type string GMF uses for identifying its diagram editors. For any diagram editor, the diagram type can be found out by opening the diagram file in a text or XML editor and looking for the attribute type of the Diagram element as it can be seen in figure 5.6.[19] It should be noted that it is not possible to identify the diagram editors by the underlying meta model, as the same meta model can be used for different diagram types.

[17] Manually creating the subclasses would not be feasible either, as a GMF diagram might consist of dozens of edit parts which all need to be checked whether they may be parents of runtime elements.

[18] The implementation of this provider is located in the class de.upb.dmm.gmf.augmentation.diagram.providers.-AugmentingEditPartProvider.

[19] This value is the so-called *semantic hint* the diagram editor defined for its diagram view model element. GMF uses this semantic hint is to associate the editor-independent view model elements with actual diagram editor implementations.

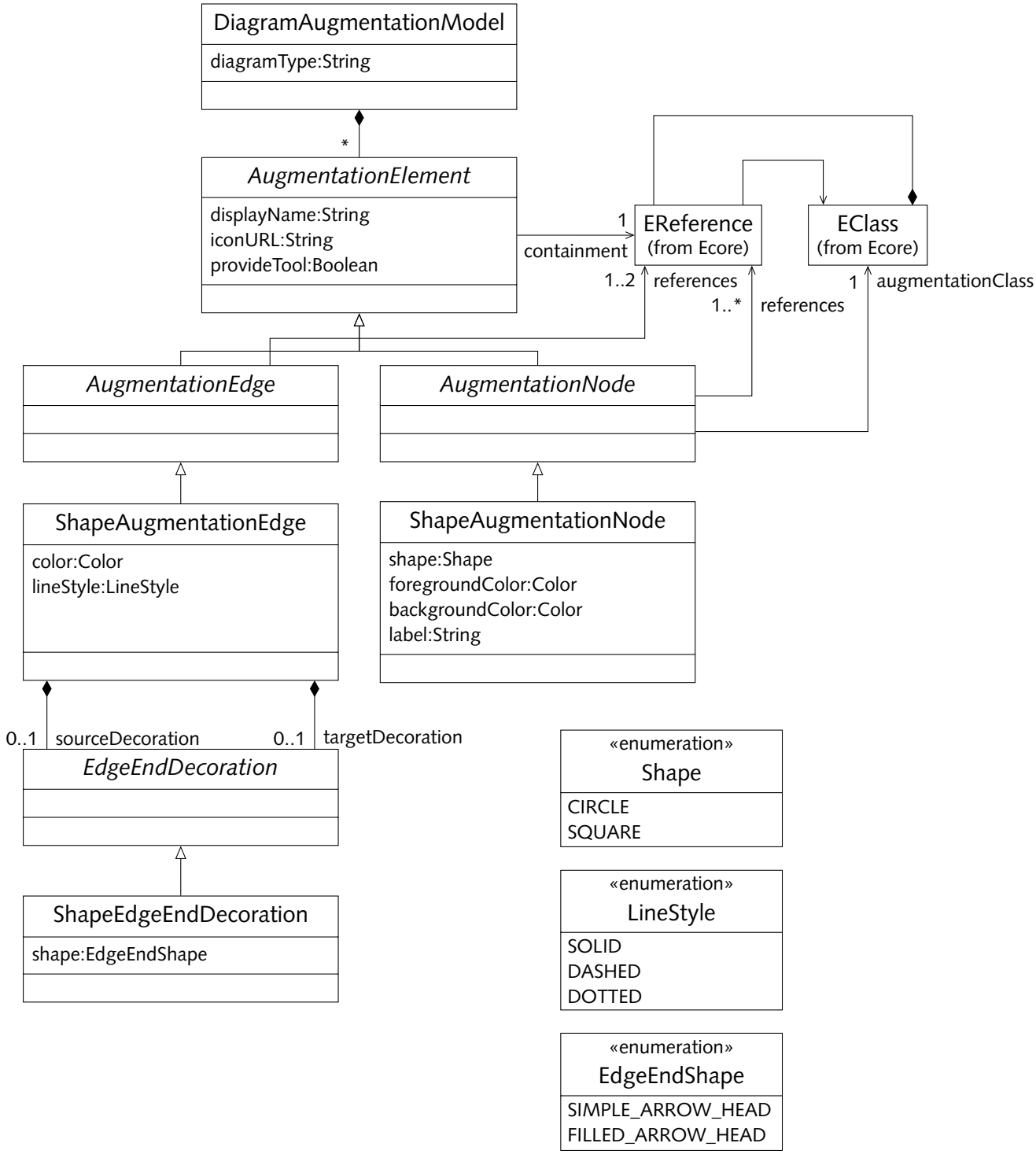

Figure 5.7: Meta model for GMF diagram augmentation models

The *AugmentationElement* is the abstract super class for the classes that specify a model element that is supposed to augment another model element that is already visualised in the diagram editor. All three attributes of it are used for providing a UI entry in the diagram editor palette for creating a new instance of the augmenting model element.

Furthermore, the class has a reference called containment to the EReference class from the Ecore meta model, i.e., a meta-meta model. This is required as the diagram augmentation is supposed to be a generic method, independent of the meta models visualised in a diagram editor. Thus the diagram augmentation model needs to use the next meta level when referencing other models.

The containment reference specifies the containment aggregation to which newly created instances of this class should be added.[20]

The subclasses of *AugmentationElement* differentiate between *AugmentationEdge*s and *AugmentationNode*s. The latter class represents a new node type that is supposed to be added to the diagram editor. The reference augmentationClass of the class *AugmentationNode* specifies the class that represents the new nodes in the meta model. The reference with the name references referencing the class EReference specifies the existing diagram elements to be augmented. The EReference must be a member of the class specified by the augmentationClass reference. Actually, more than one EReference can be used by an *AugmentationNode*; thus, the new node can be attached to different elements from the existing model.

The class *AugmentationEdge* represents an edge between either an augmenting and an augmented element or between two augmenting elements. The references reference specifies the ends of the edges. Beneath the obvious case of two ends, the multiplicity of the reference also allows the specification of only one end. This is useful when the model element representing the edge is also represented as a node in the same diagram; the edge will then connect its corresponding node with the model element to which the single reference points.

Both the *AugmentationEdge* and the *AugmentationNode* are abstract classes as they do not yet define the further visualisation of the augmenting elements. For now, for each class there is only one concrete subclass which allows the specification using visual attributes such as color and shape. However, further subclasses that use more sophisticated methods for defining the appearance of the augmenting elements are thinkable.

5.4.2.6 Diagram Augmentation Model for UML Activities

Now we can use a diagram augmentation model to define the concrete syntax of DMM runtime elements in UML activity diagrams. Figure 5.8 shows the model in a tabular form.

As you can see, there are two tokens classes; the ControlToken class is the one that could be seen in all example activity diagrams this thesis provided. The ObjectToken which models object flow is visualised by a small black square. Both tokens have the same base class, Token, which also defines the reference that is used in the augmentation model.

[20] As the EReference implies a meta model class, but not an actual instance to which new objects could be added, the GMF Diagram Augmentation component will search for objects of this class in the augmenting model. If there is exactly one object, the newly created object will be added to that. If there is more than one, a dialog will appear and ask for the object to be used.

Type[21]	augmentationClass	references	Visual Properties
Node	ControlToken	Token.contained_in	circle, black background
Node	ObjectToken	Token.contained_in	square, black background
Node	Offer	Offer.carriedby_node, Offer.carriedby_edge	circle, white background
Edge	Offer	Offer.base	dashed
Node	BehaviorExecution	BehaviorExecution.executes	square, label EX
Node	ActionExecution	ActionExecution.executes_action	square, label EX

Figure 5.8: Diagram augmentation model for the DMM runtime model for UML activities

The class Offer has two separate references for referring to nodes and edges it is attached to. Thus, both need to be specified in the configuration. The second entry for the class Offer realises the dashed edges between offers and tokens. Only one reference is specified here, as the class Offer directly references the class Token with the reference named base.

The remaining two entries realise the EX nodes displayed when an activity or action is being executed.

[21] In this column, Node signifies an instance of the class ShapeAugmentationNode. Edge signifies a ShapeAugmentationEdge.

Conclusion 6

This chapter concludes this thesis. We will briefly summarise the topics of it in section 6.1. Section 6.2 will provide an outlook over further possible features or applications of the integrated development environment for DMM whose base has now been built in form of the DMM Player.

6.1 Summary

A declared goal of DMM is to facilitate semantics specification by providing an approach which is specially tailored to the problem domain. While the conceptual development of DMM is quite advanced, the tool support has still some weaknesses which obstruct this goal. An approach to improve this situation is the idea of an integrated development environment for DMM. The goal of this thesis was to make the first step towards such an IDE by conceiving and implementing a visual debugger and interpreter for DMM.

The base of the concept was laid by the extensive requirements analysis in chapter 3. This included a review of the development process for DMM semantics; the manual review of the execution and the search for an error source after failed test cases have been identified as central use cases induced by this process. We also learned that semantics developers and later "end users" of the semantics have quite different requirements for an IDE. Thus, two dedicated breakpoint concepts were created. The breakpoint concept for semantics developers bases on the applicability on rules, including the newly introduced property rules. A breakpoint concept based on the concrete syntax has been proposed for model developers without knowledge of DMM.

The analytical work continued in chapter 4, now with the focus on the conceptual realisation of the requirements identified before. Thus, a method for recognising execution path switch instances has been conceived. Then, we compared several methods for the definition of visual execution steps that do not exhibit temporal inconsistencies. Afterwards, the concept of rule events was introduced to provide a common base for execution path switches, visual steps, and rule breakpoints.

Chapter 5 described the design and the implementation of the new software. Two groups of components play a crucial role in the DMM Player: The DMM runtime component group

provides a set of re-usable and re-configurable components for the execution of models with DMM specifications. Furthermore, the GMF Diagram Augmentation component enables GMF model editors provided by third parties to display runtime information in a configurable way. A special feature of this component is the complete independence from the other DMM components. Thus, it can be also used in other contexts.

The DMM Player itself could already be practically employed during the final phase of the development of the DMM semantics for UML activities [Hor09]. The feedback from the user can be regarded as favourable. A screen shot of the DMM Player showing various of its features can be seen in figure 6.1.

The goals set in section 1.1 have been met; Still, the actual support for model developers without specific knowledge of DMM is quite limited; yet, we have identified a number of approaches towards a better tool support in this area.

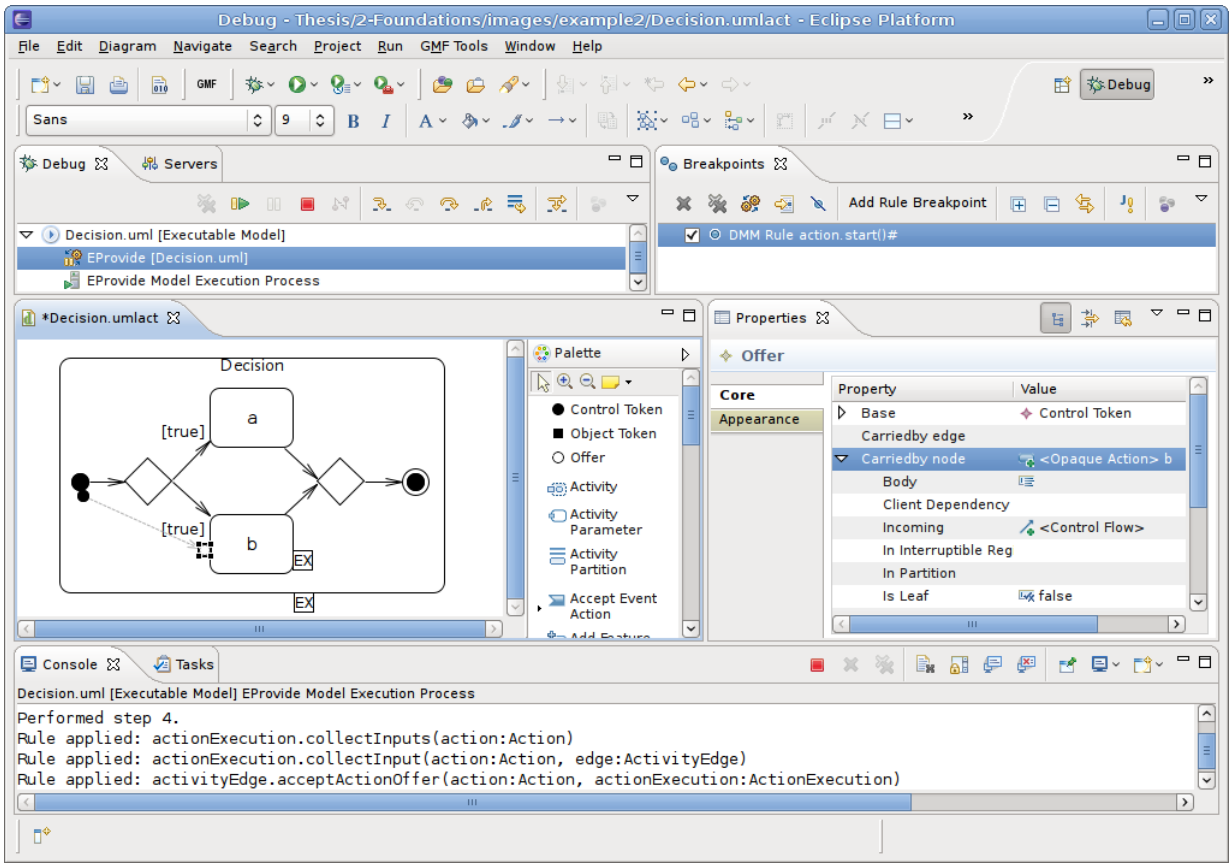

Figure 6.1: Screen shot of the DMM Player in debug mode with the debug controls, the breakpoints view, the properties editor for the selected offer and the execution log in the console

6.2 Outlook

As the DMM Player represents only the beginning of the integrated development environment for DMM, there is quite a significant potential for improvements and further research and development.

Featurewise, the DMM Player is closely related to classical debuggers. However, the special characteristics of the graph transformations employed by DMM may make further features desirable which do not stem from the world of debuggers. In order to be better able to explore the behaviour of a model with DMM-specified semantics, a manual choice of the matches to be applied on the model might be useful—opposed to the mostly automatic choice performed now by the DMM Player. The user interface could be related to the manual exploration user interface of GROOVE.[1] Likewise, a visualisation of the selected match in the concrete syntax of a displayed diagram could contribute to the understanding of the behaviour of the particular rule. During the execution of a model, property rules could be used in a similar manner to highlight certain constellations in the concrete syntax.

A further problem while developing rule based semantics is finding out why a rule that is supposed to match does not match in a particular situation.[2] A mechanism that shows the rules that just partially match the particular state and that also shows the reasons why they do not completely match could be helpful. It should however be said that such a feature depends on the support of a corresponding feature by GROOVE which is presently not available.

Besides debugging features, further support for the test driven specification process for DMM semantics[3] is desirable. We have seen that the tests used in this process provide a counter-example if they fail. This counter-example is a path through the transition system induced by the model; thus, it could be "played" in concrete syntax with the DMM Player to facilitate the understanding of the cause for the error. Furthermore, a way of automatically creating further test cases for a semantics using the current model state might be also useful.

In the course of this thesis, we have also identified some weaknesses of the semantics specification approach by DMM which should be further explored. An obvious way of defining the "natural" steps of a DMM semantics that exhibit no temporal inconsistencies are bigstep rules; however, due to concurrency it is sometimes necessary to create more widespread bigstep rules, as this is the only way to avoid concurrency. Possibly, further ways of avoiding concurrency should be searched for.

Apart from enhancement possibilities, some actual applications of the DMM Player should be also named; right now, two thesises with the goal of realising further DMM semantics specifications are imminent. With these thesises, semantics for UML statemachines and sequence diagrams are going to be realised. Obviously, the DMM Player can be helpful for these tasks.

An application not that near might be the realisation of a workflow execution engine. Such an engine could execute an actual workflow specified with a UML activity [Esh02]. Using the Groove DMM Runtime Controller and the DMM Rule Event Controller components, code realising actual actions—such as mailing notifications to involved employees—could be attached to the UML actions of a workflow specification.

[1] Cf. figure 3.5 on page 36

[2] Cf. section 3.3.2.3 on page 35.

[3] Cf. section 2.1.4 on page 14.

Bibliography

[Apa04] Apache Software Foundation. Apache License, Version 2.0. http://www.apache.org/licenses/LICENSE-2.0, 2004.

[Bau08] Eduard Bauer. Enhancing the Dynamic Meta Modeling Formalism and its Eclipse-based Tool Support with Attributes. Bachelor's thesis, University of Paderborn, 2008.

[BFS09] Andreas Blunk, Joachim Fischer, and Daniel A. Sadilek. Modelling a Debugger for an Imperative Voice Control Language. In *14th System Design Languages Forum (SDL Forum 2009), 22-24 Sep 2009, Ruhr-University of Bochum, Germany*. Springer-Verlag, 2009.

[BLFM98] Tim Berners-Lee, Roy T. Fielding, and Larry Masinter. Uniform Resource Identifiers (URI): Generic Syntax. RFC 2396 (Draft Standard), http://www.ietf.org/rfc/rfc2396.txt, August 1998.

[Blu03] Bill Blunden. *Software Exorcism: A Handbook for Debugging and Optimizing Legacy Code*. Springer-Verlag, New York, 2003.

[Blu09] Andreas Blunk. MODEF - Ein generisches Debugging-Framework für domänenspezifische Sprachen mit metamodellbasierter Sprachdefinition auf der Basis von Eclipse, EMF und EProvide. Diploma thesis, Humboldt-Universität zu Berlin, 2009.

[BMR+96] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stal. *Pattern-Oriented Software Architecture, Volume 1: A System of Patterns*. John Wiley & Sons, August 1996.

[BRRS08] Luciano Baresi, Vahid Rafe, Adel T. Rahmani, and Paola Spoletini. An efficient solution for model checking graph transformation systems. *Electronic Notes in Theoretical Computer Science*, 213(1):3–21, May 2008.

[CES86] Edmund Melson Clarke, Ernest Allen Emerson, and Aravinda Prasad Sistla. Automatic Verification of Finite-State Concurrent Systems using Temporal Logic Specifications. *ACM Trans. Program. Lang. Syst.*, 8(2):244–263, 1986.

[CIG+89] Graham Chapman, Eric Idle, Terry Gilliam, Terry Jones, John Cleese, and Michael Palin. *Monty Python's Flying Circus; Just the Words*, volume 1. Methuen, London, 1989.

[Dai04] John Daintith, editor. *A Dictionary of Computing*. Oxford University Press, Oxford; New York, 5th edition, 2004.

[Dij82] Edsger Wybe Dijkstra. On the role of scientific thought (EWD447). In *Selected Writings on Computing: A Personal Perspective*, pages 60–66. Springer-Verlag, New York, 1982.

[Ecl09a] Eclipse Foundation. Eclipse Platform website. http://www.eclipse.org/platform/, 2009.

[Ecl09b] Eclipse Foundation. Platform Debug website. http://www.eclipse.org/eclipse/debug/platform/, 2009.

[EFS09] Gregor Engels, Daniela Fisseler, and Christian Soltenborn. Improving reusability of dynamic meta modeling specifications with rule overriding. In *Proceedings of the 2009 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC 2009), Corvallis, Oregon (USA) (to be published)*, 2009.

[EMF09] Eclipse Modeling Framework website. http://www.eclipse.org/modeling/emf/, 2009.

[Esh02] H. Eshuis. *Semantics and Verification of UML Activity Diagrams for Workflow Modelling*. PhD thesis, University of Twente, November 2002.

[Fes08] Tatiana Fesenko. Bitter GMF or How We did UML with GMF. http://www.eclipsecon.org/2008/?page=sub/&id=387, 2008.

[Fis07] Daniela Fisseler. Improving the inheritance concept of Dynamic Meta Modelling. Master's thesis, University of Paderborn, 2007.

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley Professional, 1995.

[GMF09a] Eclipse Graphical Modeling Framework website. http://www.eclipse.org/modeling/gmf/, 2009.

[GMF09b] Eclipse Graphical Modeling Framework Runtime API Specification. http://help.eclipse.org/ganymede/topic/org.eclipse.gmf.doc/reference/api/runtime/overview-summary.html, 2009.

[Gra98] Mark Grand. *Patterns in Java: A Catalog of Reusable Design Patterns Illustrated with UML*, volume 1. Wiley, 1998.

[Hau05] Jan Hendrik Hausmann. *Dynamic Meta Modeling: A Semantics Description Technique for Visual Modeling Languages*. PhD thesis, University of Paderborn, 2005.

[Hor09] Markus Hornkamp. A Formal, Graph-Based Semantics for UML Activities. Diploma thesis, University of Paderborn, September 2009.

[HR04] David Harel and Bernhard Rumpe. Meaningful Modeling: What's the Semantics of "Semantics"? *Computer*, 37(10):64–72, 2004.

[HSSW09] Ric Holt, Andy Schürr, Susan Elliott Sim, and Andreas Winter. Graph Exchange Language. http://www.gupro.de/GXL/, 2009.

[JCI09] Apache Commons JCI website. http://commons.apache.org/jci/, 2009.

[JCJO92] Ivar Jacobson, Magnus Christerson, Patrik Jonsson, and Gunnar Overgaard. *Object-oriented software engineering. A Use Case Driven Approach*. Addison-Wesley, 1992.

[JDT09a] Eclipse Java Development Tools website. http://www.eclipse.org/jdt/, 2009.

[JDT09b] Eclipse Java Development user guide. http://help.eclipse.org/galileo/index.jsp?nav=/1, 2009.

[KR06] H. Kastenberg and A. Rensink. Model Checking Dynamic States in GROOVE. In A. Valmari, editor, *Model Checking Software (SPIN), Vienna, Austria*, volume 3925 of *Lecture Notes in Computer Science*, pages 299–305, Berlin, 2006. Springer-Verlag.

[Lam83] Leslie Lamport. What good is temporal logic? In R. E. A. Mason, editor, *Information Processing 83 – Proceedings of the 9th IFIP World Computer Congress*, pages 657–668. North-Holland/IFIP, September 1983.

[Obj06a] Object Management Group. Meta Object Facility (MOF) Core Specification – OMG Available Specification, Version 2.0. http://www.omg.org/docs/formal/06-01-01.pdf, 1 2006.

[Obj06b] Object Management Group. Object Constraint Language – OMG Available Specification, Version 2.0. http://www.omg.org/docs/formal/06-05-01.pdf, 5 2006.

[Obj07] Object Management Group. MOF 2.0/XMI Mapping – OMG Available Specification, Version 2.1.1. http://www.omg.org/docs/formal/07-12-01.pdf, 12 2007.

[Obj09] Object Management Group. OMG Unified Modeling Language (OMG UML) – Superstructure, Version 2.2. http://www.omg.org/docs/formal/09-02-02.pdf, 2 2009.

[OSG09] OSGi Alliance. OSGi Service Platform—Core Specification. http://www.osgi.org/Specifications/HomePage, 2009.

[Ren04] Arend Rensink. The GROOVE Simulator: A Tool for State Space Generation. In John L. Pfaltz, Manfred Nagl, and Boris Böhlen, editors, *AGTIVE 2003 – Revised Selected and Invited Papers*, volume 3062 of *Lecture Notes in Computer Science*, pages 479–485, Berlin, 2004. Springer Verlag.

[Ren06] Arend Rensink. Nested quantification in graph transformation rules. In A. Corradini, H. Ehrig, U. Montanari, L. Ribeiro, and G. Rozenberg, editors, *Graph Transformations (ICGT)*, volume 4178 of *Lecture Notes in Computer Science*, pages 1–13, Berlin, September 2006. Springer Verlag.

[Ren09] Arend Rensink. Groove website. http://groove.cs.utwente.nl/, 2009.

[Rhe07] Thomas Rheker. A Bidirectional Transformation between EMF Models and Typed Graphs. Bachelor's thesis, University of Paderborn, 2007.

[Röh08] Malte Röhs. A Visual Editor for Semantics Specifications Using the Eclipse Graphical Modeling Framework. Bachelor's thesis, University of Paderborn, 2008.

[Roz97] Grzegorz Rozenberg, editor. *Handbook of Graph Grammars and Computing by Graph Transformation, Vol. 1: Foundations*. World Scientific Publishing Co., Inc., River Edge, NJ, USA, 1997.

[Sau16] Ferdinand D. Saussure. Nature of the Linguistics Sign. In Charles Bally and Albert Sechehaye, editors, *Course In General Linguistics*. Bayot, Paris, 1916.

[SBPM09] David Steinberg, Frank Budinsky, Marcelo Paternostro, and Ed Merks. *EMF: Eclipse Modeling Framework 2.0*. Addison-Wesley Professional, 2009.

[SE09] Christian Soltenborn and Gregor Engels. Towards test-driven semantics specification. In B. Selic A. Schürr, editor, *Proceedings of the 12th International Conference on Model Driven Engineering Languages and Systems (MODELS 2009), Denver, Colorado (USA) (to be published)*, Berlin/Heidelberg, 2009. Springer.

[SK08] Mirko Seifert and Stefan Katscher. Debugging Triple Graph Grammar-based Model Transformations. In Uwe Aßmann, Jendrik Johannes, and Albert Zündorf, editors, *Fujaba Days*, 2008.

[Sol06] Christian Soltenborn. Analysis of UML Workflow diagrams with Dynamic Meta Modeling techniques. Diploma thesis, University of Paderborn, June 2006.

[Spi03] Diomidis Spinellis. *Code Reading: The Open Source Perspective, Vol. 1.* Pearson Education, 2003.

[SW08] Daniel A. Sadilek and Guido Wachsmuth. Prototyping visual interpreters and debuggers for domain-specific modelling languages. In Ina Schieferdecker and Alan Hartman, editors, *4th European Conference on Model Driven Architecture - Foundations and Applications (ECMDA-FA'08)*, volume 5095 of *Lecture Notes in Computer Science*, pages 63–78. Springer-Verlag, 2008.

[SW09] Daniel A. Sadilek and Guido Wachsmuth. Eprovide website. http://eprovide.sourceforge.net/, 2009.

[U2T09] Eclipse UML2 Tools website. http://wiki.eclipse.org/MDT-UML2Tools, 2009.

List of Figures

Appendix

Feature Traces

A

Trace dependencies between UML model elements can be used for denoting a causal relationship between those elements. Figure A.1 shows traces between the use cases identified in section 3.8 and the architecture defined in section 5.1.

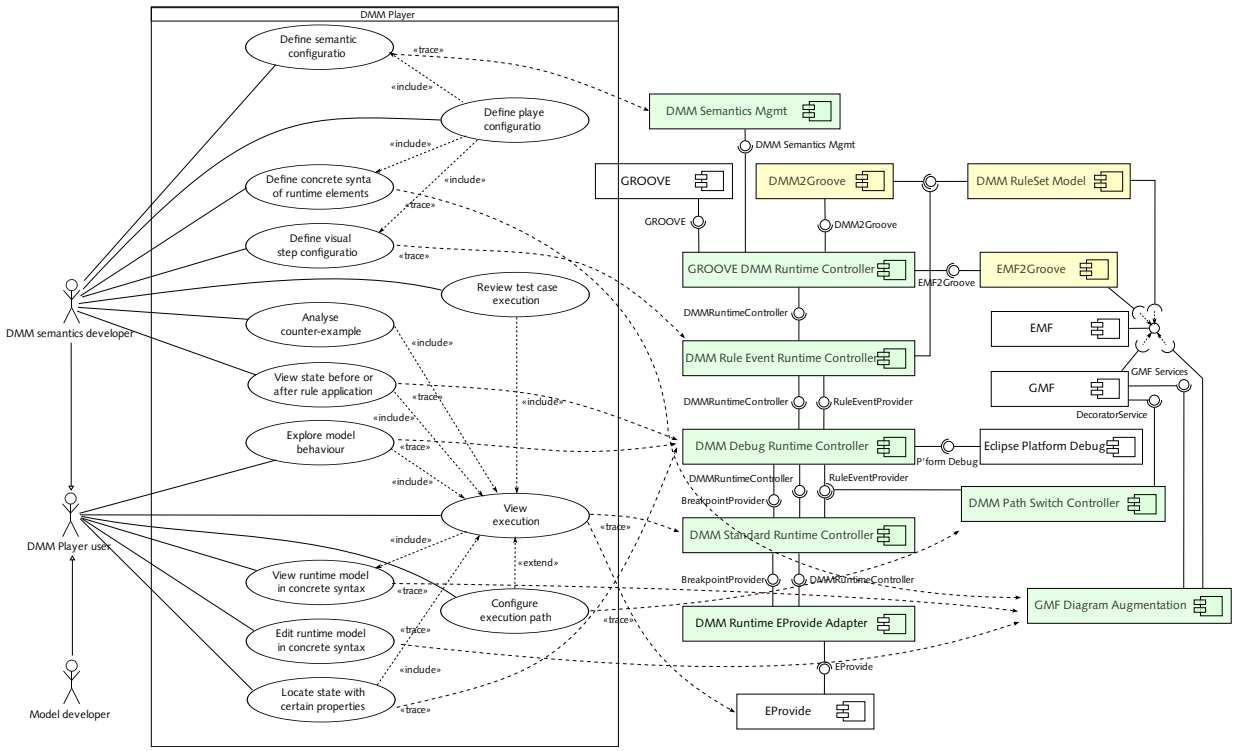

Figure A.1: Feature traces of the DMM Player

B Software

The disc accompanying this thesis contains complete Eclipse installations for Windows and Linux including the DMM Player. The installations can be directly started from the disc. Loading times will improve though, when the installations are copied to a hard disk. Besides either a Windows or Linux operating system, an installed Java JRE 5 or newer is required for running eclipse.

To start Eclipse, open—depending on your system—either the folder eclipse-linux or eclipse-windows. Eclipse may be started using the program eclipse or eclipse.exe, respectively.

The disc also comes with a demo workspace containing an activity diagram that can be executed with the DMM Player. As Eclipse workspaces need to be on read/write filesystems, this workspace has to be copied to another medium, though.

The workspace can be chosen either directly during the program start or using the menu item File > Switch workspace.

When eclipse has loaded the demo workspace, you should see in the project navigator a single project named DMM Player Demo. It contains two files: Decision.uml—a UML model containing an activity diagram—and Decision.umlact—the concrete syntax information for the model. Open the latter file. You will see the simple activity diagram from the running example. To debug it, click with the right mouse button on the file Decision.uml (not umlact!).

A menu similar to the one from figure 3.9 will appear; choose the item Debug As and then Executable Model. Afterwards, the model execution will begin.

Eidesstattliche Erklärung

Ich versichere, dass ich die vorliegende Arbeit selbständig und ohne fremde Hilfe sowie ohne Benutzung anderer als der angegebenen Quellen angefertigt habe. Alle Ausführungen, die wörtlich oder sinngemäß übernommen wurden, sind als solche gekennzeichnet. Die Arbeit hat in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegen.

Paderborn, 1. September 2009 Nils Bandener