Quality Assurance with Dynamic Meta Modeling

Ph.D. Thesis

Christian Soltenborn

UNIVERSITÄT PADERBORN
Die Universität der Informationsgesellschaft

July 2013

Quality Assurance with Dynamic Meta Modeling

Christian Soltenborn

A thesis submitted to the
Faculty of Computer Science, Electrical Engineering, and Mathematics
of the University of Paderborn
in partial fulfillment of the requirements
for the degree of Dr. rer. nat.

July 2013

In Memoriam
My Father,
Hans-Hermann Soltenborn (1945–2007)

Dedication

First of all, I would like to express my deepest gratitude to my doctoral advisor Prof. Dr. Gregor Engels. Gregor, without your support, especially in the difficult first year of my Ph.D. project, I might not have made it to this point today. I sincerely hope that the flexibility you gave me resulted in solid work, not only scientifically, but also in my teaching. Your fine eye for detail in proof reading of texts and your insightful participation in our discussions enabled me to succeed and is already sorely missed as I write this…

A Ph.D. project such as the one presented in this thesis does not rest solely on the cooperation between a doctoral advisor and a doctoral candidate, it requires the input of other dedicated individuals as well. It is for this reason that I would also like to thank Prof. Dr. Heike Wehrheim, Prof. Dr. Reiko Heckel, and Prof. Dr. Arend Rensink, to whom I owe large parts of my knowledge about model checking and graph transformations. Working with all of you has always been a pleasure, both professionally and personally.

I also would like to thank Prof. Dr. Hans Kleine Büning and Dr. Matthias Fischer as well as the other members of my Ph.D. commission for their willingness to look into my scientific work and for the generous evaluation of my performance.

My Ph.D. project would not have been possible without the groundwork and Ph.D. thesis of Dr. Jan Hendrik Hausmann, who also introduced me extensively into the exciting world of Dynamic Meta Modeling. Another big thank you goes to Dr. Markus Luckey, with whom I had many helpful discussions, and who has acted quite often as moral support. I also owe much gratitude to my former colleagues Svetlana Arifulina, Jan-Christopher Bals, Matthias Becker, Dr. Fabian Christ, Dr. Christian Gerth, Barış Güldalı, Yavuz Sancar, Henning Wachsmuth, and Friedhelm Wegener. It will be difficult moving forward without your positivity, ideas and critical feedback…

Due to the roller coaster like nature of most doctoral studies, I have been through euphoric highs and depressing lows. This is why I am glad I had friends I could rely on, to always keep things in perspective; Thank you, Jonas Gefele, Malte Röhs and Guido Schaumann.

Finally, my thanks go to my father Hans, my mother Johanna and my brother Thomas for always believing in me even if I would once more deviate from the straight path. I would not be who I am today without the unwavering support of a loving family.

Paderborn, August 2013

Christian Soltenborn

Abstract

One way to deal with the complexity of today's software systems is *model-driven development* (MDD), where the target software system is first modeled on a very abstract level in a platform-independent way (e.g., by using UML use cases), and then—step by step—refined. The final, platform-specific model contains enough information to serve as input for code generation of the target system.

MDD has several benefits: For instance, the (usually visual) modeling languages allow for better communication with stakeholders, which is particularly true when using *domain-specific languages* (DSLs), i.e., languages containing concepts of the problem domain. Another advantage is that the modeler's task is simplified by the small complexity of getting from one to the next abstraction level, some steps of which are even applied using automatic model transformations.

MDD is most beneficial if the modeling languages in use have a well-defined syntax and semantics. This is often true for the syntax and static semantics part, e.g. by using *MOF metamodeling* techniques as suggested by the *Object Management Group*. For the behavioral semantics, the situation is usually worse. For instance, the UML has a MOF-based syntax definition, but its behavioral semantics is defined with natural language, leaving room for ambiguities. The same is true for many DSLs.

One reason for this is that semantics specification for behavioral modeling languages is a difficult task. This is where Dynamic Meta Modeling (DMM) comes into play. DMM is a semantics specification technique targeted at MOF-based modeling languages, where a language's behavior is defined by means of graphical operational rules which change runtime models.

The DMM approach has first been suggested by Engels et al. in 2000 [63]; Hausmann has then defined the DMM language on a conceptual level within his PhD thesis in 2006 [96]. Consequently, the next step was to bring the existing DMM concepts alive, and then to apply them to different modeling languages, making use of the lessons learned to improve the DMM concepts as well as the DMM tooling.

The result of this process is the DMM++ method, which is presented within this thesis. Our contributions are three-fold: First, and according to our experiences with the DMM language, we have introduced new concepts such as refinement by means of *rule overriding*, and we have strengthened existing concepts such as the dealing with *universal quantified structures* or *attributes*.

Second, we have developed a test-driven process for semantics specification: A set of test models is created, and their expected behavior is fixed. Then, the DMM rules are created incrementally, finally resulting in a DMM ruleset realizing at least the expected behavior of the test models. Additionally, we have defined a set of coverage criteria for DMM rulesets which allow to measure the quality of a set of test models.

Third, we have shown how functional as well as non-functional requirements can be formulated against models and their DMM specifications. The former is achieved by providing a visual language for formulating temporal logic properties, which are then verified with model checking techniques, and by allowing for visual debugging of models failing a requirement. For the latter, the modeler can add performance information to models and analyze their performance properties, e.g. average throughput.

Contents

1 Introduction **1**
1.1 Motivation . . . 1
1.2 Current State of Dynamic Meta Modeling . . . 2
1.3 Objectives of this Thesis . . . 3
1.4 Structure of this Thesis . . . 4

I Foundations **7**

2 Eclipse Modeling Framework **9**
2.1 Metamodeling and MOF . . . 9
2.2 Ecore Metamodel . . . 12
2.3 The Eclipse Modeling Framework . . . 14
2.3.1 Features of EMF . . . 14
2.3.2 EMF-Based Tools and Frameworks . . . 15

3 UML Activities **17**
3.1 Overview . . . 17
3.2 Syntax . . . 19
3.3 Semantics . . . 21
3.3.1 Tokens and Offers . . . 21
3.3.2 Semantics of Language Elements . . . 22

4 GROOVE **25**
4.1 Graph Transformation in GROOVE . . . 25
4.2 State Space Exploration . . . 30
4.3 Model Checking GROOVE Grammars . . . 31
4.3.1 Temporal Logic . . . 31
4.3.2 The GROOVE Model Checker . . . 32
4.4 Tool Support . . . 33

5 Dynamic Meta Modeling **37**
5.1 Goals of and Requirements on DMM . . . 37
5.2 Dynamic Meta Modeling . . . 38
5.2.1 Overview . . . 38
5.2.2 Language Definition . . . 43
5.3 Evaluation of DMM's Current State . . . 50

Summary **53**

II Dynamic Meta Modeling ++ 55

6 Language Definition of DMM++ 57
6.1 Comparison of DMM and DMM++ 57
6.1.1 Conceptual Changes 57
6.1.2 Pragmatic Changes 58
6.2 Syntax 59
6.2.1 Ruleset Structure 60
6.2.2 Rule Hierarchy 63
6.2.3 Internal Rule Structure 71
6.2.4 DMM Expression Language 79
6.3 Semantics 87
6.3.1 Challenges 88
6.3.2 Basic Transformation Concepts 89
6.3.3 Universal Quantified Structures 96
6.3.4 Attributes 101
6.3.5 Rule Overriding 105
6.3.6 Restrictions 118
6.4 Related Work 119

Summary 123

III Quality of DMM++ Specifications 125

7 Creating DMM Specifications 127
7.1 DMM and Model Transformations 128
7.2 From Syntax Metamodel to Runtime Metamodel 129
7.2.1 The “From Scratch” Approach 130
7.2.2 The Decorator Approach 139
7.3 Creating DMM Rulesets 140
7.4 Related Work 143

8 Test-driven Semantics Specification 147
8.1 Test-Driven Semantics Specification 147
8.1.1 Creating Example Models 148
8.1.2 Creating the Semantics Specification and Deriving Test Cases 154
8.2 Coverage Criteria for Tests of DMM Specifications 158
8.2.1 Covering DMM Specifications 159
8.2.2 Invocation Graph 159
8.2.3 Coverage Criteria 163
8.2.4 Hierarchy of Coverage Criteria 174
8.3 Related Work 176

Summary 179

IV Quality of Models 181

9 Formulating and Verifying Requirements 183
9.1 Functional Requirements 183
9.1.1 Example Requirement: Soundness 184
9.1.2 Pattern Process Specification Language 185
9.1.3 Generalizing (E)PPSL 191
9.1.4 Formalizing and Verifying Soundness 193
9.2 Non-Functional Requirements 195
9.2.1 Example: Process Improvement with Fixed Budget 196
9.2.2 Performance Evaluation Process Algebra 198
9.2.3 Modeling Performance Information in DMM 202
9.2.4 DMM goes PEPA 207
9.2.5 Improving the Example Process 208
9.3 Related Work 209
9.3.1 Functional Requirements 209
9.3.2 Non-Functional Requirements 213

10 Debugging Models 215
10.1 Visual Model Execution 216
10.1.1 Visualizing Runtime Information 216
10.1.2 Defining the Steps of Executions 219
10.1.3 Controlling Execution Paths 222
10.1.4 Example: UML Statemachines 228
10.2 Model Examination 230
10.2.1 Controlling Model Execution 230
10.2.2 Model Execution Process 232
10.2.3 Debugging Models 234
10.3 Implementation 234
10.4 Related Work 237

Summary 239

11 Summary and Outlook 243

List of Figures 249

List of Listings 253

List of Tables 254

Bibliography 255

A Appendix 273
A.1 Custom OCL Operations 273
A.2 DMM: Static Semantics 275

1 Introduction

1.1 Motivation

Every computer scientist knows *Moore's Law*, which basically states that the complexity of CPUs (i.e., the number of transistors they are built of) will double every two years [146]. Gordon E. Moore, one of the founders of Intel, has indeed predicted this as soon as 1965, and the law still seems to hold (although we seem to have reached some physical barriers which might make it impossible to downsize transistors beyond a certain point).

Not surprisingly, the complexity of software has constantly increased in parallel to that of CPUs, making it more and more difficult to deal with that complexity. One way to tackle this problem of overwhelming complexity is *abstraction*: a set of elements is investigated for commonalities, and elements being (more or less) similar to each other with respect to these commonalities are grouped within *classes*. We then do not have to reason about all the single elements; in contrast, it might suffice to reason about the classes of elements, instead of the elements themselves. Since there are usually less classes of elements than elements, this approach reduces complexity. Another benefit of abstraction is that to define classes of elements, one has to really understand what exactly these elements have in common and how they differ amongst each others, therefore usually leading to a better understanding of the problem at hand.

In software engineering, one particular abstraction technique has become increasingly important within the last decade: The use of *visual modeling languages* in general and the *Unified Modeling Language* (UML) [158] in particular. The UML provides 15 different sub-languages dedicated to modeling all aspects of a software system, from identifying and structuring requirements with *use case diagrams* up to the deployment of the finished software system onto an IT infrastructure with *deployment diagrams*.

The UML provides two kinds of diagrams: *structural* and *behavioral* diagrams. The former is—as the name implies—used to describe structures, the most important one being *class diagrams*. With the latter kind of diagrams, the actual behavior of the system can be described; of particular importance is the language of *activities*, which will also be used as an example language within this thesis. UML activities can be used to describe workflows at a relatively high level of abstraction, but they are also equipped to express more low-level behaviors such as algorithms.

Another kind of languages has become rather important within the last

decade: *Domain-specific languages* (DSLs). A DSL is a language targeting a particular, usually rather narrow domain, and it contains language elements which directly refer to concepts from that domain. For instance, a DSL for the domain of cellphone applications might contain an `SMS` element referring to an actual SMS which can be either sent or received by a cellphone. This allows the developer to directly use those concepts within the models she creates (instead of needing to map the concepts to the language elements of a general modeling language such as the UML), therefore reducing the *semantic gap*.

However, to make the most use out of behavioral models, their semantics needs to be defined precisely and un-ambiguously. For instance, executing activities within a workflow engine or generating executable code from activities is only possible if the language's semantics is well-understood by both the creator of the workflow engine or code generator as well as the language user, i.e., the person creating the activities to be executed.

Unfortunately, the UML specification does not fulfill this seemingly simple requirement: The semantics of the UML's behavioral models is specified by means of natural language accompanying the syntax specification (which is provided by means of *metamodels*, i.e., models which describe the structure of valid instances of the UML). The same is true for DSLs: In most cases, their semantics is defined by means of natural language.

As such, the need for semantics specification techniques arose which overcome the obvious drawbacks of such an informal language specification. One promising candidate for such a technique is *Dynamic Meta Modeling* [63, 96].

1.2 Current State of Dynamic Meta Modeling

Dynamic Meta Modeling (DMM) is a semantics specification technique which is targeted at behavioral languages whose syntax is defined by means of a metamodel, as is the case for the UML. In a nutshell, DMM works as follows: In a first step, the syntax metamodel is enhanced with concepts needed to express states of execution of language instances, resulting in a so-called *runtime metamodel*. For instance, the UML specification states that "The semantics of activities is based on token flow" [158, p. 326]. As such, the runtime metamodel of UML activities introduces a `Token` concept; locations of tokens then refer to states of execution of the activity at hand.

The second step of creating a DMM semantics specification contains of defining the actual behavior of the language. This is done by means of *graph transformation rules* [177, 57] typed over the runtime metamodel. A graph transformation rule is a formal means to describe changes on graphs; in the case of DMM, these graphs are instances of the runtime metamodel. For instance, the UML specification states that "... an action can only begin execution when it has been offered control tokens on all incoming control flows... " [158, p. 320]. As a result, the DMM specification for UML activities contains a graph transformation rule which can be executed as soon as this is the case; it will then start the execution of the according action.

Since a DMM specification is backed by the mathematical formalism of graph transformations, it fulfills the requirement of being precise and un-ambiguous. Furthermore, such a formal specification can be analyzed and processed automatically, giving rise to the possibility of actually developing tool support with

explicit support for the language's semantics.

However, a semantics specification should not only be understandable by experts of the formalism used. Fortunately, this is not the case for DMM: Graph transformation rules can be visualized as (annotated) object diagrams, i.e., instances of the language's metamodel. This allows *advanced language users* (i.e., language users who are familiar with the language's metamodel) to easily understand such a semantics specification, and to use it e.g. as a reference of the semantics.

In his Ph.D. thesis [96], Hausmann has thoroughly motivated the need for DMM, presented a formal definition, and demonstrated it using a simplified semantics of UML activities. However, his Ph.D. thesis is more about concepts than applications, i.e., conceptual as well as practical approaches for actually analyzing models equipped with DMM specifications have been out of scope of his work. The Ph.D. thesis submitted herewith closes this gap.

1.3 Objectives of this Thesis

The objective of this Ph.D. thesis is to make DMM usable for *language engineers* as well as *language users*, i.e., we will show how a language engineer can create high-quality DMM semantics specifications, and how language users can use such DMM specifications to create high-quality models. Additionally, we will introduce the tool support we have developed to support those tasks.

Let us describe our goals in more detail. First of all, a DMM semantics specification is pretty much useless if it contains flaws. For instance, if a language user wants to verify the actual behavior of one of her models, she can only rely on the analysis results if the semantics specification is free of errors. Otherwise, it will be very difficult or even impossible to distinguish between flaws of her model and flaws of the specification itself. We tackle this problem by two means: First, we provide tool support for creating DMM semantics specifications and for checking these specifications for syntactical correctness. Second, we have developed the approach of *Test-driven semantics specification* [191], a process for creating specifications following the approach of *test-driven software development* [19].

Now, given a high-quality DMM specification, language users can use that specification for improving the quality of their models. For this, we will describe how to formulate [190] and verify [65] safety and liveness properties such as soundness [203]. Additionally, we will show how to derive a *Performance Evaluation Process Algebra* (PEPA) [84] model from a DMM specification and a user's model, and how to use that PEPA model to verify the user's model for non-functional properties. Finally, we have developed a model-driven approach to visualize the execution of models equipped with a DMM specification in their own language's concrete syntax [12]. Such an animated visualization of a model's behavior can be used in two scenarios which will both help to improve the model's quality: First, the language user will get a better insight in what the model actually does, and second, the described technique can be used to understand and fix flaws of the model identified during analysis of functional properties.

A number of publications have been the foundation of this thesis. In particular, the scientific contributions aggregated within this thesis are

- an extension of DMM with concepts to refine DMM specifications [192, 62],
- a test-driven process for creating high-quality DMM specifications [191],
- coverage criteria for measuring the quality of tests of a DMM specification [5],
- a visual language for describing functional requirements against models and according DMM specifications [190],
- a means to verify functional requirements against models and according DMM specifications using model checking [65], and
- a model-driven means to enhance existing DMM specifications for visual, animated execution and the possibility to visually debug models, including the usage of breakpoints and watchpoints [12].

As such, the result of this thesis are concepts and tools which nearly[1] support the complete lifecycle of behavioral languages, from specifying a language's semantics to analyzing and improving the quality of models.

1.4 Structure of this Thesis

From a high-level view, this Ph.D. thesis is structured into four parts which will present foundations and preliminary work, introduce our enhanced notion of DMM, show how to create high-quality DMM specifications, and discuss how to analyze models equipped with a DMM specification.

Part I – "Foundations" The foundations start with an introduction to the Eclipse Modeling Framework (EMF) in Chapter 2, which is the conceptual and technical foundation not only for the DMM tooling which has been developed, but also for the languages DMM is targeted at. We are showing the relation of EMF to the OMG standard MOF [154], and give a brief overview of EMF's metamodeling language as well as the parts of EMF used by DMM.

Chapter 3 introduces the behavioral language of UML activities, which will then serve as running example for the rest of the thesis. The syntax of the activities will be defined by means of excerpts of the UML metamodel, and we will give an intuition of the semantics of UML activities based on the text accompanying the syntax definition within the UML specification [158].

Chapter 4 presents GROOVE, a set of tools around performing graph transformation. GROOVE is crucial to DMM in that it serves as a graph transformation engine: DMM specifications and models are transformed into GROOVE grammars, which then allows to explore their state space and perform model checking.

Chapter 5 will then briefly present DMM as defined by Hausmann in his Ph.D. thesis [96]. We will start by discussing the use cases and target users of DMM, followed by a brief introduction to DMM's syntax and semantics. Finally, we point out areas where Hausmann's definition of DMM can be improved to support practical application of DMM.

Part II – "Dynamic Meta Modeling++"

[1] Exceptions are requirements specification and syntax definition.

Chapter 6 introduces DMM++, the enhanced version of DMM in which we implemented the necessary improvements as identified in chapter 5. We will briefly present the metamodel of DMM++, which we have developed not only for the sake of supporting the development of tooling, but also to make the abstract syntax of DMM++ explicit. We give an introduction to the DMM language in Sect. 6.2; additionally, Appendix A.2 on page 59 provides DMM++'s static semantics, which we have defined using OCL expressions [155]. Finally, we will define the semantics of DMM++ by describing our mapping of DMM++ into GROOVE graph grammars in a precise way. During this chapter, we will also present our extensions to the DMM formalism [62, 192].

Part III – "Quality of DMM++ Specifications"

Chapter 7 shows how a DMM++ specification can be created with the tooling developed as part of this thesis. We will first introduce two approaches for deriving a runtime metamodel from a syntax metamodel, both having its own advantages and disadvantages. The second part of chapter 7 will deal with creating the DMM++ rules specifying the language's semantics: We will introduce the DMM++ Workbench which not only implements the metamodel and transformation to GROOVE graph grammars as defined in part II, but also provides a visual, easy-to-use editor for creating DMM++ specifications.

Chapter 8 will then introduce our approach of test-driven semantics specification [191, 64], which will help the language engineer to create high-quality DMM++ specifications. The basic idea is that in a first step, example models are created, each of which demonstrating the semantics of few language elements. In a second step, the expected semantics of these models is formalized and verified against the DMM++ specification under creation. Finally, we will define some coverage criteria which will allow us to reason about the quality of our language's tests [5].

Part IV – "Quality of Models"

Chapter 9 will cover the topics of formulating properties which should hold for a language user's model and analyzing if this is indeed the case. The first section of chapter 9 deals with functional requirements: We show how states can be formulated as instances of the language's metamodel, and how temporal properties over those states can be defined by means of a generalized version of the existing pattern process specification language (PPSL) [190] and verified using model checking techniques [65]. In the second section, we show how to formulate performance properties for a DMM++ specification and a given model, and how to transform them into a PEPA model [100] which can be analyzed for non-functional properties such as average throughput.

Chapter 10 will then show how the DMM++ Player [12] can be used to visualize counter examples gained from verifying models as described above, and how the language user can benefit from such visualizations. We will see that the DMM++ Player allows for the completely model-driven specification of visualization information, which will at runtime be used to augment existing visual editors, thus simplifying the development of such visualizations to a great extend.

Finally, we will summarize the thesis' contents and discuss conclusions and further work in the context of DMM++.

Part I

Foundations

2
Eclipse Modeling Framework

The name *Dynamic Meta Modeling* implies that metamodeling is deeply involved in the DMM++ approach. And indeed, not only are DMM++ rules *typed over* metamodels, but the DMM++ language is also defined by means of metamodels. In this chapter, we will introduce the *Eclipse Modeling Framework* (EMF) [44, 194] which provides the metamodeling language *Ecore*, the language of choice in case of DMM++. We first give an introduction to metamodeling and MOF in the next section. Section 2.2 then discusses the Ecore metamodel. Finally, Sect. 2.3 gives a brief overview of the Ecore-based technologies that are used by DMM++.

2.1 Metamodeling and MOF

The idea of metamodeling is pretty simple: A metamodel is a model of models. This means that a metamodel describes how "its" models "look like"; it is a model for a whole set of models. In terms of computer science, a metamodel is a formalization of the abstract syntax of the described set of models, just as a grammar describes a set of textual sentences.

To better understand the concept of metamodeling, let us investigate an example. Figure 2.1 shows a simple metamodel to the left and a corresponding model to the right. Here, the metamodel is a UML class diagram. As such, its semantics is that a `VideoShop` has a name, and that it owns `Videos` which are characterized by their title. To the right, an instance of the metamodel can be seen: a `VideoShop` instance which owns two `Videos`. Metamodels can be rather precise: For instance, the metamodel of Fig. 2.1 clearly answers the question whether all videos must be owned by a shop (which is true here, because the 1 at the association states that every video must be owned by exactly one shop).

The benefits of defining a data structure by means of a metamodel has a number of advantages; see e.g. [154] for a discussion of this topic. The most important advantage is that metamodels are a powerful and understandable means to formally describe the abstract syntax of a set of models. However, if this is best practice, shouldn't then the syntax of the metamodel of Fig. 2.1 also be formally defined? And indeed, the answer is "Yes"; to formally describe a metamodeling language such as UML class diagrams, a metametamodeling language is needed. Additionally, we need to find another way to describe that language formally (obviously, a metametametamodeling language is not an

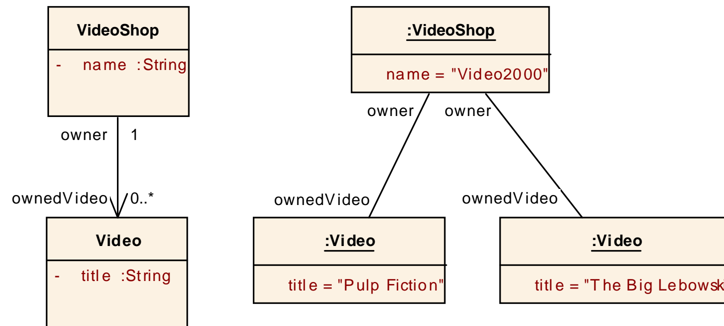

Figure 2.1: A simple metamodel for a video shop, and an instance of the metamodel.

option here).

The answer is OMG's *Meta Object Facility* (MOF) [154] which "provides a metadata management framework, and a set of metadata services to enable the development and interoperability of model and metadata driven systems." [154, p. 5]. MOF is a language whose main purpose is to describe types, their properties and relations. Basically, MOF consists of a core version of UML class diagrams;[1] however, it does not contain any other language elements of the UML.

The formalization of MOF is then provided recursively: MOF is an instance of MOF. E.g., as we will see in the next section, MOF contains the classes `Package` and `Class`; the according MOF instance defining MOF will therefore contain objects of type `Class`, two of which having the `name` attribute set to "Package" and "Class".

As such, MOF forms the "root" of the metamodel hierarchy, an example of which is depicted as Fig. 2.2. The idea is to provide a metameta language at the very top of the hierarchy (M3)– this is the role of MOF. Using MOF, the metamodels of other languages can be defined as instances of MOF; this takes place on level M2. The most prominent example of such a language is the UML. Then, on M1, the language defined on M2 can again be instantiated, resulting in actual models of that language. For instance, the definition of UML class diagrams sits at level M2, and a concrete class diagram model is located on M1. Finally, level M0 depicts the "reality", i.e., concepts of the real world which are represented by the models on M1. Note that the recursive definition of MOF is not visualized in Fig. 2.2.

Note also that restricting ourselves to four metamodel levels can be confusing: For instance, it is of course possible to use the UML to define a language dedicated at defining the syntax of languages; in that scenario, MOF would be M4, the UML would live on M3, the metamodel of the newly defined language would live on M2, models of that language would live on M1, and finally, the real world concepts on M0. The only important point are the *instance-of* relations between the levels. Or as the MOF specification clearly states: "Suffice it to say MOF 2.0 with its reflection model can be used with as few as 2 levels

[1] In fact, the MOF language elements are imported from other packages, and these packages are—besides others—also imported by the UML; thus, MOF basically *is* the core of UML class diagrams.

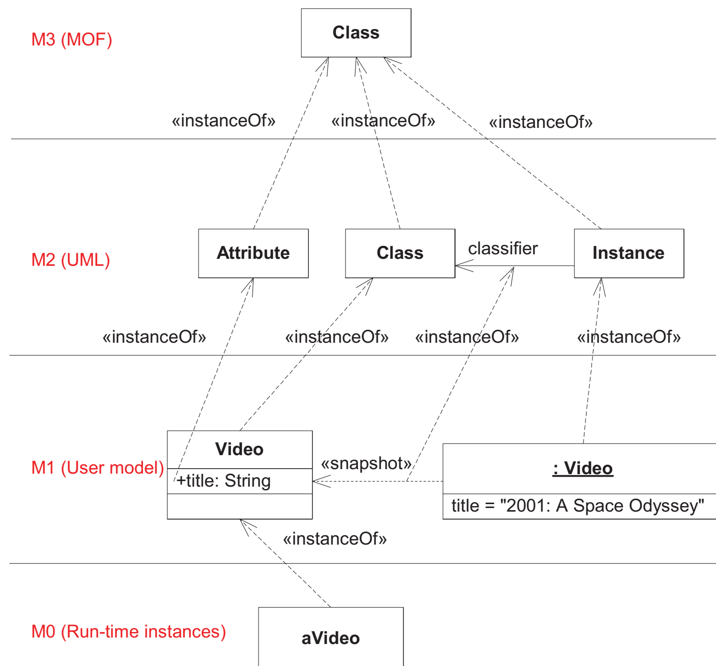

Figure 2.2: An example of the four-layer metamodel hierarchy (from [157, p. 19]).

and as many levels as users define." [154, p. 9] As such, Fig. 2.2 only depicts a common case. See [8] for more information on metamodeling on and accross different levels.

MOF has seen a high adoption in industry as well as academia. However, implementing and actually making use of MOF revealed that it had a couple of drawbacks; in particular, "One of the challenges the MOF 2.0 Core and MOF 2.0 XMI Mapping submissions face is to maintain a stable interchange model (XMI) while MOF 2 and UML 2 are changing quite significantly. To accomplish this, we [have identified] a very small subset of the modeling concepts in MOF. We call this Essential MOF or EMOF which basically models simple classes with attributes and operations to fix the basic mapping from MOF to XML and Java." [154, p. 6]

The most important and de-facto standard implementation of EMOF is Ecore, the metamodeling language of the Eclipse Modeling Framework. In fact, Ecore has influenced the redesign of MOF and the introduction of the EMOF core. To quote [194, p. 40]: "With a focus on tool integration, rather than metadata repository management, Ecore avoids some of MOF's complexities, resulting in a widely applicable, optimized implementation. MOF and Ecore have many similarities in their ability to specify classes and their structural and behavioral features, inheritance, packages, and reflection. They differ in the area of life cycle, data type structures, package relationships, and complex aspects of associations. . . . Development experience from EMF has substantially influenced this latest version of the specification, in terms the layering of the architecture and the structure of the semantic core. Essential Meta-Object Fa-

cility (EMOF) is the new, lightweight core of the metamodel that quite closely resembles Ecore. Because the two models are so similar, EMF is able to support EMOF directly as an alternate XMI serialization of Ecore.". See also [81] and [145] for transformations between MOF and Ecore.

Since Ecore is so similar to EMOF, we do not bother to show the EMOF metamodel – instead, we turn our attention to Ecore in the next section.

2.2 Ecore Metamodel

The Ecore metamodel provides means to model object-oriented data structures. For this, it provides concepts such as types, attributes, etc. A slightly simplified Ecore metamodel is depicted as Fig. 2.3; the first thing to notice is that by convention, all classes of the Ecore metamodel start with a capital "E", therefore clearly indicating their origin (which can be useful in practice when e.g. dealing with both Ecore and UML models programmatically, since as we have seen above, the UML also contains the Ecore language elements). In the following, we will discuss the Ecore concepts.

The root class of Ecore's inheritance hierarchy is the `EModelElement` metaclass, which is able to carry a set of `EAnnotations` – all other Ecore metaclasses (transitively) inherit from `EModelElement`. One example is `ENamed-Element`, which adds a `name` attribute and is again superclass of nearly all Ecore classes.

Ecore models are organized in packages, the metaclass of which is `EPackage`. Most importantly, the `EPackage` metaclass contains the `nsUri` attribute: the *namespace URI* uniquely identifies an `EPackage`. Despite its subpackages, a package contains an arbitrary number of `EClassifiers` and thus—transitively—all other model elements.

`EClassifiers` come in two flavors: `EDataType` represents primitive datatypes such as integers, strings, or enumerations, and `EClass` represents complex datatypes; the latter is the central class of the Ecore language. An `EClass` can be *abstract* or represent an *interface*.[2] The inheritance hierarchy is modeled by the `eSuperTypes` reference (note that Ecore supports multiple inheritance).

Furthermore, an `EClass` owns a set of `EStructuralFeatures`, of which again two kinds exist. `EReference` models references of other `EClasses`, and `EAttribute` models attributes. Both metaclasses are subclasses of `EStruc-turalFeature` and thus from `ETypedElement`, through which they inherit quite a few features.

`ETypedElement` represents an element having some sort of type, be it a complex or primitive type; the `type` reference is refined by the references `eReferenceType` and `eAttributeType` of the subclasses. The metaclass takes care of the cardinalities with the attributes `lowerBound` and `upper-Bound` (where `upperBound = -1` refers to an arbitrary number of elements), from which `many` and `required` are computed (the former is defined as `upper-Bound > 1`, the latter as `lowerBound > 0`). In the case of many elements, features can be declared as `ordered`, meaning that the elements' order will be maintained, and `unique`, specifying whether a single element is prevented from occuring more than once.[3]

[2] Consistency is dealt with by automatic validation of additional constraints.
[3] Note that according to [194, p. 107], "Currently, their use is limited. The behavior of a

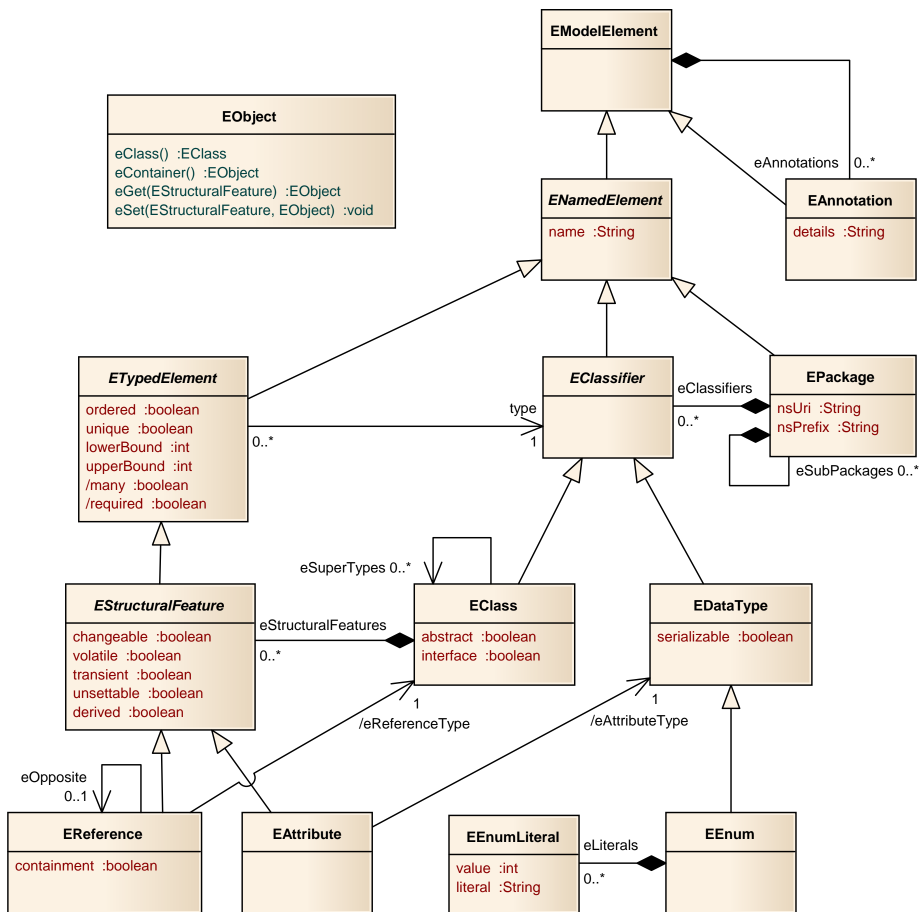

Figure 2.3: Simplified Ecore metamodel.

The EStructuralFeature metaclass adds attributes that determine how elements are stored and accessed: changeable determines whether the feature can be changed by an external caller; derived defines whether a feature's value(s) are computed. A feature being declared transient will not be serialized with the owning object. If a feature is unsettable, it has an additional possible state: An object's feature can either be unset or carry some value(s), and this is distinguishable. Finally, volatile determines whether the feature's value(s) will be stored in the owning object at all; in the case of a derived feature, this is often not necessary.

In addition to EAttribute, which only inherits the features of the above classes, EReference provides two more features: an EReference can be declared to be a containment, meaning that the according elements are not only referenced, but *stored* within the referencing object. This is of particularly importance for serialization: Objects will be serialized according to their location witin the containment tree. Additionally, an EReference can reference another EReference as its opposite. This is Ecore's notion of bidirectionality: The feature can reference another EReference such that the two references are type compatible. If the feature is set, it must always be true that reference.opposite.opposite = reference.

Finally, the EObject metaclass is the class which will be the supertype of all instances of EClasses, and through which these instances inherit the Ecore reflection mechanisms. In other words: All objects contained in a model which has been defined by (and is consistent to) an Ecore metamodel will inherit from EObject, and are thus equipped with the according reflection mechanisms. This e.g. allows to ask an object for its type using the eClass() operation, to traverse the EClass's EStructuralFeatures, and then to receive all values of the object's references and attributes by calling eGet(feature) with the respective EReference or EAttribute.

2.3 The Eclipse Modeling Framework

The Ecore metamodel we have seen in the last section is the core of the Eclipse Modeling Framework (EMF). However, EMF is much more; in this section, we will give a brief introduction to the additional features EMF provides, many of which DMM++ uses. The most important advantage of having a standardized metamodeling infrastructure is easy tool integration; this has particularly been proven by the success of EMF in the modeling community, which resulted in a huge amount of tools, languages, and frameworks around EMF. Again, many of these are used within the DMM++ tooling, as shown in Sect. 2.3.2.

2.3.1 Features of EMF

First of all, as we will see in Sect. 6.2, the syntax of DMM++ is defined by means of an Ecore metamodel. Within that metamodel, despite the defined concepts also Ecore concepts are used, allowing to reference other Ecore metamodels from DMM++ models (i.e., instances of the DMM++ metamodel).

multivalued attribute depends on its uniqueness, but references always behave as if unique is true. Moreover, ordered is ignored by all features, as a List-based implementation is always used."

DMM++ makes heavy use of different metamodeling levels as seen in Sect. 2.1. For instance, we will see in Sect. 7.2.1 how a DMM++ model references the DMM++ metamodel, the Ecore metamodel, and a third metamodel defined by means of Ecore. In addition, DMM++ defines not only the core DMM++ metamodel, but also a couple of other metamodels which e.g. allow to describe event traces (Sect. 8.1.1.5), performance information (Sect. 9.2.3.1), or visual execution information (Sect. 10.1.1).

EMF provides sophisticated code generation support, i.e., an Ecore metamodel can be transformed into Java code ready to be deployed and executed. The first step of generating code consists of generating and potentially adjusting the *EMF generator model*. The generator model allows to influence code generation in a varietly of ways: For instance, code for different runtime platforms can be generated, including code ready to be deployed on RAP [45], a web-based implementation of the Eclipse UI components. For DMM++, the Eclipse IDE is the target runtime platform.

EMF not only generates model code which e.g. includes support for reflection, observation and notification, serialization and referential integrity, but also components allowing for an easy integration of model editing capabilities, including a tree-based editor which is ready to be used, but can be customized; for instance, changing the icon associated with a model element is as easy is replacing the according generated icon file. For usability reasons, the DMM++ tooling makes heavy use of these customization capabilities: As an example, the resulting DMM++ tree editor as well as property editor can be seen in Fig. 7.14 on page 142.

Since all generated model classes inherit from `EObject` as seen in Sect. 2.2, the generated model code supports Ecore reflection, allowing for a convenient as well as efficient implementation of e.g. the bidirectional transformation between Ecore models and graphs of the GROOVE toolset (we will ge to know GROOVE in Sect. 4 and the transformation in Sect. 6.3).

2.3.2 EMF-Based Tools and Frameworks

As mentioned above, EMF has had a huge success in the modeling community, resulting in a great number of tooling based on EMF/Ecore. In the following, we briefly mention the components used within the DMM++ tooling:

- Due to Ecore's de-facto standard status, many other modeling languages have been implemented based on Ecore and EMF – all of these languages play nicely with the DMM++ tooling. The most prominent example is the Eclipse Foundation's implementation of the UML metamodel [51], which closely follows the specification of the UML as described in [157, 158], including the definition of several basic packages which are then merged to packages supporting the different UML compliant levels, following the package merge semantics as described by the OMG.

- The static semantics of the DMM++ language has been defined by means of OCL constraints [155], which was possible because of the Ecore implementation of the Eclipse OCL project [47]. The DMM++ editors support powerful model validation including visual feedback which has been implemented using the EMF Validation framework [48].

- The visual editors allowing to intuitively edit DMM++ models (Sect. 7.3) have been developed using the *Graphical Modeling Framework* (GMF) [90, 50], a framework which allows to generate basic visual editors in a model-driven way which can then be customized through runtime extensions.
- For the transformations of DMM++ models into GROOVE grammars (see Sect. 6.3), a number of unit tests exist, the most basic of which make use of the EMF Compare framework [46] allowing for the comparison of arbitrary Ecore based models.
- The textual language for describing event traces (Sect. 8.1.1.5) has been developed on top of the XText framework [69, 54], which allows to generate a language's Ecore metamodel and a text editor including syntax highlighting and completion from a grammar.

Summarizing, the implementation of sophisticated tool support for working with DMM++ would not have been possible without the extensive usage of the several components available from EMF and the surrounding modeling community. As such, DMM++ itself can be taken as a proof for the success of metamodeling techniques.

3 UML Activities

The UML contains a number of diagram types, some of which deal with modeling behavior. Of those, UML activities are used to model control and data flow. In this thesis, we use UML activities as a running example; therefore, in this chapter we will give an introduction to that language. We start with a general overview of activities in the next section. Section 3.2 will then present the most important activity contructs, and Sect. 3.3 will give a brief introduction to the semantics of UML activities.

3.1 Overview

UML activities model the flow of control between actions, which are the places where actual work is performed. They can be used at different abstraction levels: activities can e.g. model workflows, where actions correspond to tasks such as "Check claim probability", but also algorithms, where an action might increase the attribute value of the incoming object by 1.

An example UML activity from the UML specification [158] is depicted as Fig. 3.1. It models a workflow and is thus on a rather high level of abstraction. Its semantics is as follows: First, it is tried to record the problem. If this is not succesful, the process ends. Otherwise, if the problem is rectified, we skip to reporting (see below). Only if this is not the case, we try to reproduce the problem. If we can not do this, the problem statement needs to be improved; we skip to reporting. If we can however reproduce the problem, then we either develop and test a solution, or we use a known solution. Finally, we report the results of the process by communicating them to the reporter and by auditing and recording the process – the latter two steps can be performed concurrently.

Now that we have an intuition of the semantics of UML activities, let us investigate their definition a bit closer: The UML specification defines the activities language by means of several packages which depend on each other. The dependencies of these packages are depicted as Fig. 3.2. In the following, we briefly introduce the package's contents.

FundamentalActivities Defines an activity to consist of actions.

BasicActivities Introduces control and data flow constructs such as activity edges.

IntermediateActivities The basic control flow constructs are defined here, such as decisions and concurrency. The UML specification states that the

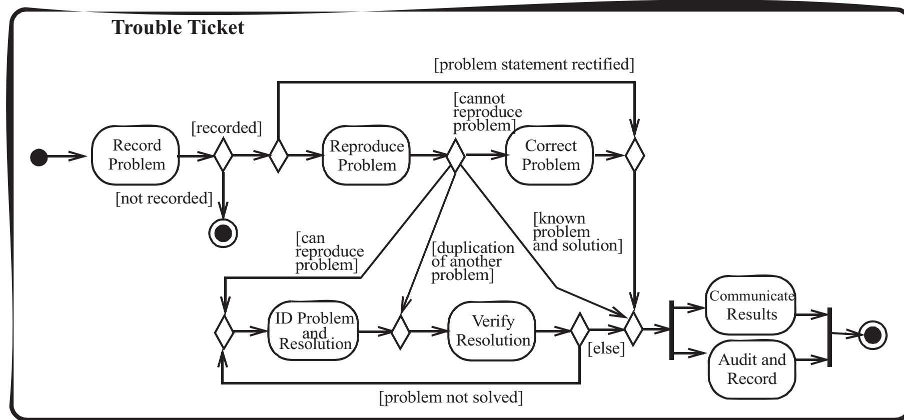

Figure 3.1: Example UML activity modeling a workflow (from [158, p. 333].

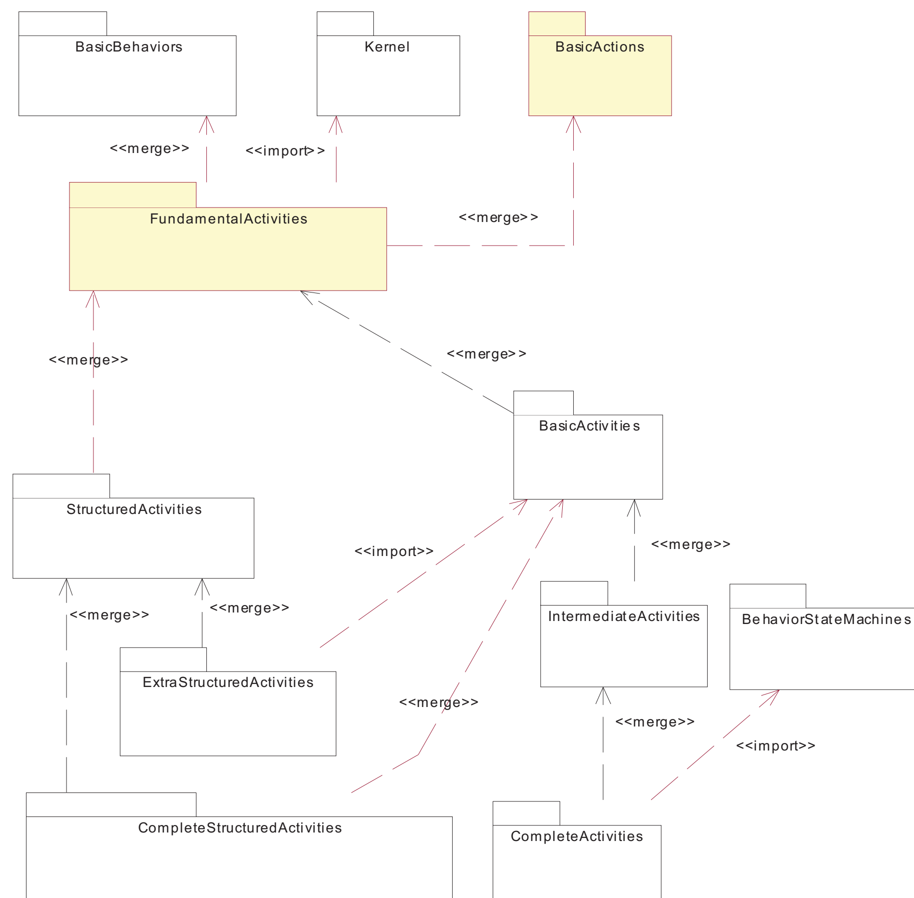

Figure 3.2: UML activity packages and their dependencies (from [158, p. 305].

package "supports modeling similar to traditional Petri nets with queuing" [158, p. 303].

CompleteActivities Enhances the basic constructs, e.g. by introducing edge weights and streaming.

StructuredActivities Introduces constructs representing traditional programming structures such as loops and conditions.

CompleteStructuredActivities Adds support for data flow output pins to the constructs introduced in StructuredActivities

ExtraStructuredActivities Introduces exception handling and dealing with sets of values.

The formal semantics used within this thesis, in particular the one developed by Hornkamp [105], basically realize the activity constructs up to the IntermediateActivities package, plus some additional features such as edge weights and basic exception handling. In the next section, we will introduce the metamodel of the IntermediateActivities package.

3.2 Syntax

As we have seen in Sect. 2, the syntax of the UML is defined by means of a metamodel. Figure 3.3 shows a slightly simplified version of the UML activities metamodel, i.e., the classes of the IntermediateActivities package. An `Activity` consists of `ActivityNodes` and `ActivityEdges` which connect `ActivityNodes`. There are two kinds of `ActivityEdges`: `ControlFlows` model flow of control, and `ObjectFlows` model data flow.

In the case of `ActivityNodes`, several types exist, which can first of all be grouped into `Actions`, `ControlNodes`, and `ObjectNodes`. `Actions` are where the actual work is being performed; the UML provides several kinds of `Actions`, which we do not show in Fig. 3.3 – note, however, that the `Action` metaclass is abstract. Two important kinds of `Actions` are the `OpaqueAction`—which basically allows to store a string within the `Action`—and the `CallBehaviorAction` which allows to invoke other `Behaviors`. Since the `Activity` class inherits from `Behavior`, this can be used to decompose an activity into subactivities to be performed if the according `CallBehaviorActions` are executed.

`ObjectNodes` are used to store objects for a limited amount of time; for instance, an `Action's` `InputPin` will store data which the `Action` consumes as soon as it is executed. Another example are `ActivityParameterNodes`, which can be used to provide input to an `Activity`. In contrast, `ControlNodes` can not store data. Instead, they are used to route the flow of control and the data through the activity.

There are three basic kinds of `ControlNodes`, which are concerned with the start and end of an `Activity`, decisions, and concurrency. For the former kind, four `ControlNodes` exist: An `Activity` starts at its `InitialNodes` and `ActivityParameterNodes`, and ends with either a `FlowFinalNode` or `ActivityFinalNode`.

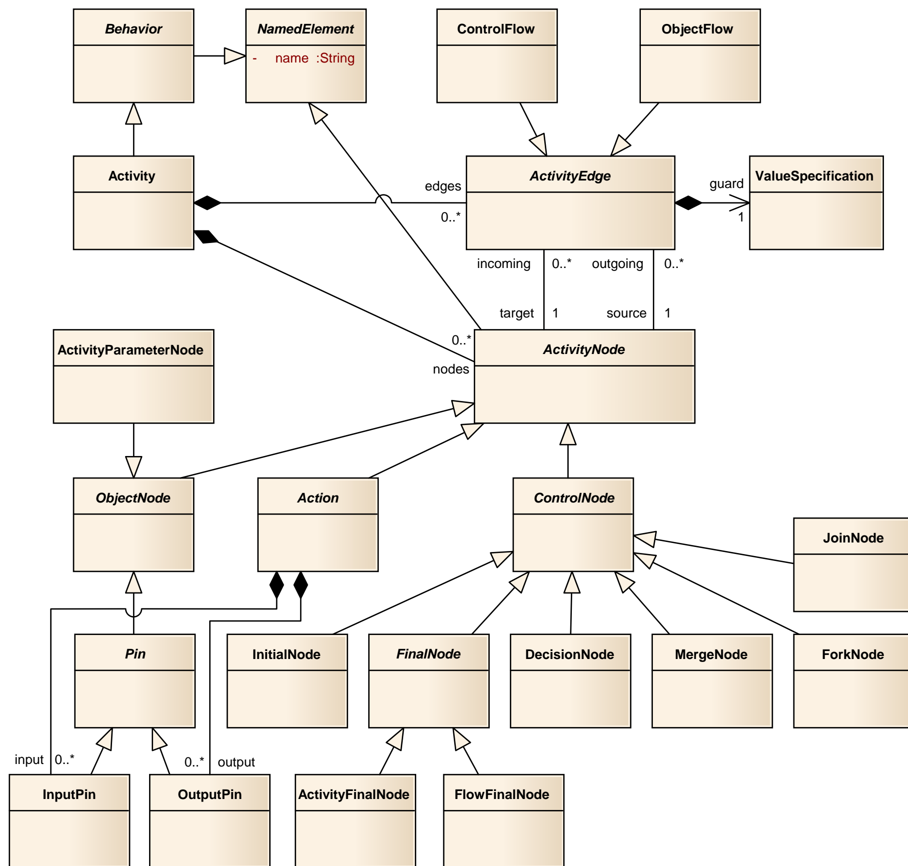

Figure 3.3: Simplified metamodel of UML activities.

Decisions are modeled by `DecisionNodes` and `MergeNodes` (which have the same concrete syntax: a diamond). They differ in that a `DecisionNode` has one incoming and several outgoing `ActivityEdges`; for the `MergeNode`, it is vice versa. In the case of a `DecisionNode's` outgoing edges, they can be equipped with *guards*: At activity execution time, the guards are evaluated, and the `ActivityEdge` whose guard has evaluated to true will be taken.

Finally, concurrency is modeled by means of `ForkNodes`, which start concurrency, and `JoinNodes`, which are points of synchronization. Again, the two elements have the same concrete syntax.

Now that we have a brief understanding of the syntax of UML activities, let us discuss their semantics in the next section.

3.3 Semantics

As mentioned earlier, the behavioral semantics of UML activities is based on token flow. However, this is only part of the truth: In fact, tokens flowing through an activity are only offered to edges. In the next section, we will explain the general idea of the token/offer semantics of UML activities, before we define the semantics of each language element in terms of token/offer flow in Sect. 3.3.2.

3.3.1 Tokens and Offers

We have seen above that the UML specification gives activities a "petri-like semantics". However, the semantics is in fact much more complicated, as we will see in the next paragraphs. Let us first make our understanding of token flow more precise: The basic idea is that if an activity is started, control tokens are placed at the activity's `InitialNodes`, and object tokens are placed on the activity's `ActivityParameterNodes` and associated with the parameter objects; these tokens then start to flow along the activity's edges.

Now, an `Action` can be executed as soon as its input requirements are fulfilled: All incoming `ControlFlows` must carry at least one control token, and all `InputPins` must carry an object token. If the `Action` executes, it consumes the incoming tokens; after execution, it produces tokens on its outgoing `ControlFlows` and `OutputPins`, which can then continue flowing through the activity.

However, the situation is more complicated. The UML specification states: "The object flow prerequisite is satisfied when all of the input pins are offered all necessary tokens, as specified by their minimum multiplicity, and accept them all at once up to their maximum multiplicity, precluding them from being consumed by any other actions. This ensures input pins on separate actions competing for the same tokens do not accept any the action cannot immediately consume, causing deadlock or starvation as actions wait for tokens taken by input pins of other actions but not used." [158, p. 320].

In other words: Tokens stick at their places until an `Action`'s complete prerequisites are fulfilled. If this is the case, all required tokens of an `Action` are consumed at once. This concept is also called *traverse-to-completion*: A token is offered all the way down, until it is accepted somewhere – it can not get stuck at some intermediate control node. This has an important implication:

The semantics of activities is *non-local*. This means that the decision whether a specific token moves might depend from elements which are far away from the token's location. E.g., consider the Action "Record problem" of the activity shown as Fig. 3.1; after execution, the Action will offer a token on its outgoing edge. However, in the extreme case (i.e., the problem is recorded, and the statement is rectified) the token will stay at Action "Record problem" until it is accepted by both "Communicate results" and "Audit and record". More on this topic can be found in [196], in which Störrle and Hausmann try to map the token flow of UML activities to Petri nets – they conclude that traverse-to-completion, but also other activity features such as exception handling are not suitable for being expressed with Petri nets.

In the next section, we will define the semantics of the activity language elements by means of token and offer flow.

3.3.2 Semantics of Language Elements

The UML specification states that an execution of an activity is controlled by an activity execution; as such the activity execution handles the tokens and offers flowing through the activity. In package IntermediateActivities, the Activity metaclass receives the isSingleExecution attribute of type boolean. If the attribute's value is true, each execution of the activity will be handled by an own activity execution; otherwise, more than one execution of the activity may result in tokens competing with each other.

When an activity is started, an activity execution is created, a control token is put on each of the activity's InitialNodes, and an ObjectToken is put on each of the activity's ActivityParameterNodes and associated with the object serving as parameter. Note that this is indeed possible: The UML specification states that "Tokens cannot "rest" at control nodes, such as decisions and merges, waiting to move downstream. Control nodes act as traffic switches managing tokens as they make their way between object nodes and actions, which are the nodes where tokens can rest for a period of time. Initial nodes are excepted from this rule." [158, p. 327]

Then, the tokens are offered to the InitialNodes' and ActivityParameterNodes' outgoing edges – if accepted, the tokens' offers start flowing through the activity until finally accepted at an action or consumed by a FinalNode. This concept is also referred to as *traverse-to-completion*.

The semantics of Actions is defined as follows: "Except where noted, an action can only begin execution when it has been offered control tokens on all incoming control flows and all its input pins have been offered object tokens sufficient for their multiplicity. The action begins execution by accepting all the offers of control and object tokens allowed by input pin multiplicity. When the execution of an action is complete, it offers control tokens on its outgoing control flows and object tokens from its output pins." [158, p. 320].

It remains to explain the other ControlNodes' semantics. We start with the FinalNodes, of which two types exist. The FlowFinalNode consumes every offered token. The ActivityFinalNode does the same, but additionally, it also consumes all other tokens flowing through the activity, therefore ending the activity's execution immediately.

DecisionNodes are used to route tokens according to their outgoing ActivityEdges' guards – each ActivityEdge whose guard evaluates to true is

offered the token. `MergeNodes` just pass every token offer they receive to their single outgoing edge. They are needed because of the "implicit join" semantics of `Actions`: If the control flow is split by a `DecisionNode`, it can not be joined by means of an `Action`, since an `Action`—as we have seen above—can only execute if all its `ControlFlows` are offered a token. However, it will only ever be offered a token on one of the `ControlFlows` originating from the split of control.

Finally, the semantics for `ForkNodes` is that if they are offered a token on their incoming edge, they will offer copies to all outgoing `ActivityEdges`; the offers then start to flow concurrently. The semantics of the `JoinNode` is the exact opposite: It will produce an offer on its single outgoing `ActivityEdge` as soon as it has offers on all its incoming `ActivityEdges`. Note that the semantics of concurrency has intentionally left rather vague by the OMG: For instance, it is not clear what happens if offers arrive at a `JoinNode` which belong to different `ObjectTokens`.

Summarizing, we have seen that UML activities are used to describe flow of control and possibly data, and that the syntax of activities is defined by means of the UML's metamodel. The semantics of activities is only provided as natural language and is—due to the traverse-to-completion semantics of token and offer flow—rather complex. As such, UML activities serve well as an example language for demonstrating DMM++, as we will do in the remainder of this thesis.

4

GROOVE

GROOVE is a set of tools for working with graphs and graph transformation. It has been developed at the University of Twente by Arend Rensink and others (see e.g. [166, 82]). GROOVE is an abbreviation of “GRaphs for Object-Oriented software VErification”. The rational of GROOVE is that object-oriented data structures consist of objects which keep primitive data and point to other, referenced objects. Especially the latter implies that graphs would be a suitable formalism for representing such data structures; as such, GROOVE aims at providing modeling and verification techniques for graph transformation systems.

Implementing the DMM++ approach involved the need to implement graph transformation, which is far from being trivial. After careful consideration, we decided not to implement an own graph transformation engine, but to use an existing one. It turned out that from the available graph transformation tools, GROOVE was the only one focusing on state space exploration and model checking; additionally, many DMM++ concepts mapped nicely to GROOVE concepts. Therefore, we are using GROOVE for this purpose.

In the next section, we will introduce the notion of graph transformation GROOVE supports. Section 4.2 will then show how the state space of a graph can be explored using different strategies, and 4.3 will investigate the model checking capabilities of GROOVE. Finally, Sect. 4.4 will present the tool support provided by GROOVE, of which DMM++ makes heavy use.

The following is based on the GROOVE version used by the DMM++ tool support. In the meantime, GROOVE has seen quite some progress; for instance, dedicated type graphs can now defined, and GROOVE provides support for advanced abstraction techniques [172, 18, 169]. We do not explain those features here; see [82] for a discussion of GROOVE’s current modeling and analysis capabilities.

4.1 Graph Transformation in GROOVE

Graph transformation is a formalism where graphs are transformed by *graph transformation rules*. The basic idea of graph transformation is pretty simple: A graph transformation rule r consists of a left-hand graph G_L and a right-hand graph G_R. Giving a *host graph* G, r *matches* G if a morphism from G_L to G can be found (i.e., if G contains G_L as a subgraph). If this is the case, r can be *applied* to G, basically meaning that the occurence of G_L in G is replaced

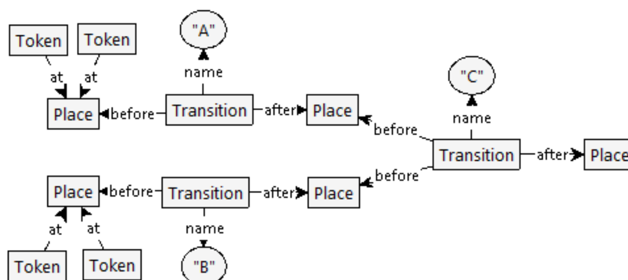

Figure 4.1: Example GROOVE host graph representing a Petri net.

by G_R, resulting in a new graph G'. This implies that a node or edge which is contained in both G_L and G_R will not be changed, a node or edge only contained in G_L will be deleted from G, and a node or edge contained only in G_R will be created in G.

The notion of host graphs GROOVE supports are directed graphs with binary, labeled edges. Nodes are not able to carry labels; however, they can of course have self edges, and the labels of these edges can be displayed by GROOVE as node labels for convenience.

Additionally, GROOVE nodes can be equipped with attributes. This is modeled by special nodes which correspond to (primitive) attribute values such as integers, booleans, or strings; an ordinary node n has an attribute with name `myValue` and value `0` if n has an outgoing edge e with label `myValue` that has as target node an attribute node representing the value `0`. GROOVE supports the datatypes `String`, `int`, `real`, and `bool`.

Let us consider an example GROOVE host graph, which is depicted as Fig. 4.1. The graph represents a Petri net with three transitions and four places, with two tokens on each of the very left places. Note that as mentioned above, GROOVE displays a node's self edges' labels as node labels. The transition nodes additionally have a `name` attribute of type string.

Let us now turn to the notion of graph transformation rules GROOVE supports. First of all, GROOVE does not use a left-hand and right-hand graph to define a rule; instead, the single rule graph is annotated. For instance, let n be a node which is contained only in the left-hand graph of a common graph transformation rule, the effect being that this node will be deleted by the rule's application. In the case of GROOVE, node n will be contained in the (single) rule graph and will be annotated `del:`. Similarly, a node annotated `new:` will be created (corresponding to a node only being contained in the right-hand graph).

GROOVE follows the Single-Pushout approach to graph transformations [31] and supports non-injective matchings, but provides so-called *merge-embargo edges* which will prevent the matching of two vertices connected with such an edge to a single vertex in the state graph. Thus, GROOVE rules can be made to match injectively by an exhaustive usage of merge-embargo edges. Additionally, GROOVE allows to enable injective matching on the grammar level.

GROOVE allows to equip each rule of a GROOVE grammar with a *priority*: If two rules with different priorities match the same state graph, only the rule with higher priority can be applied to that graph.

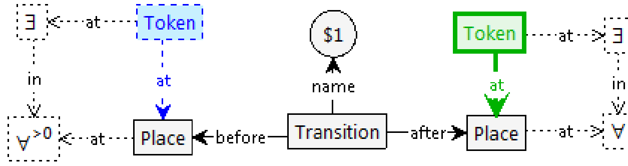

Figure 4.2: Example GROOVE rule describing the semantics of a firing transition.

Within GROOVE rules, attributes can be used to influence a rule's matching, and attribute values can be computed. For the latter, a number of operations on the above datatypes are provided.

The expressiveness of GROOVE rules is increased enormously by their support for *universal quantified structures* (UQS), which allow to define rules which do not only treat the explicitly occuring nodes and edges, but *all* nodes and edges matching a certain structure. The notion of UQS supported by GROOVE is particularly powerful since it allows for nesting of UQS. See [168, 170] for more details on GROOVE's UQS semantics.

Our goal shall now be to give the Petri net model of Fig. 4.1 an execution semantics. For this, we use a single graph transformation rule which is depicted as Fig. 4.2 and defines the semantics of a firing transition. In the lower middle of the rule, the static Petri net structure can be seen: a `Transition` with a single source and target `Place`. Above the `Places`, `Tokens` can be seen which differ in color and line shape. The colors and shapes are the concrete syntax of GROOVE's annotations: the `Token` to left (blue, thin dashed border) implies that the node is to be deleted, and the `Token` to the right (green, thick solid border) will be created by the rule. Of course, the same applies for the `at` edges of those nodes.

The structures to the very left and very right of the rule indicate UQS: the `Places` are annotated with $\forall$ and $\forall^{>0}$, meaning that *all* source and target places of the transition shall be matched. In addition, the `Token` nodes are annotated with $\exists$, and that $\exists$ nodes are nested in their respective $\forall$ nodes. This can be read as "For all `Places`, at least one `Token` must exist, and that token will be deleted" (left side of rule).

Overall, the rule realizes exactly the desired semantics: It matches for `Transitions` which carry at least one `Token` on all their source `Places`. Application of the rule deletes one `Token` from each source `Place` and creates one `Token` on each target `Place` of the `Transition`.

One of the main goals of GROOVE is to enable the analysis of systems of graphs and graph transformation rules. Therefore, GROOVE allows to explore the *state space* of a such a system; it is represented as a labeled transition system (LTS), where host graphs are states, and transitions are applications of graph transformation rules. This is of course also possible for the graph of Fig. 4.1 and the rule of Fig. 4.2. However, since there only exists a single rule, the transition system would not be very useful without investigating the graphs of the states. Note that due to the overloading of "transition" in the context of Petri nets and labeled transition systems, we will from now on only use the metaclass notation `Transition` for the former.

We are now ready to explain the missing element of the rule of Fig. 4.2:

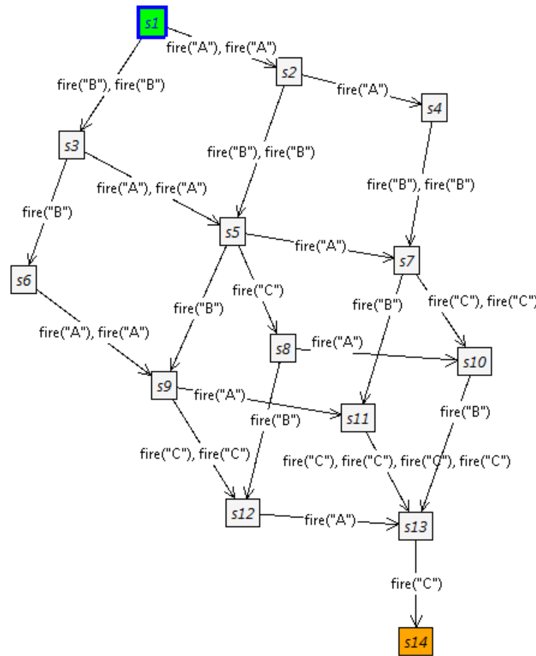

Figure 4.3: Labeled transition system resulting from the graph of Fig. 4.1 and the rule of Fig. 4.2.

the node labeled `$1` is called a *rule parameter*: It results in the value of the according node's attribute (here: the `name` attribute of the `Transition` node) to be displayed as part of the transition's label. In our example, this means that we can see from the transition system which transition has fired, without inspecting the underlying state graphs.

The LTS resulting from our model and rule is depicted as Fig. 4.3. Due to the concurrently flowing tokens, it is rather complex. As explained above, the transitions are labeled with the applied rule's name, in which the `Transition's` name is encoded. The LTS has been computed by using the example graph as the start graph. From that state on, the single rule has been applied for all matches, resulting in new state graphs. That process has been repeated until no new states are found.

Note that some transitions are labeled with more than one rule label. This is because the according rule has been applied more than once for different matches; however, all applications resulted in the same (i.e., isomorphic) state graph.

The LTS of Fig. 4.3 shows all possible ways the Petri net of Fig. 4.1 can execute. However, before we turn to the analysis of LTS in the next section, let us introduce two more GROOVE constructs DMM makes use of.

First, besides the possibility of annotating nodes and edges to be deleted or created, they can also be annotated with `not:`. This models a *negative application condition* (NAC), i.e., structures which must not be contained in the host graph for the rule to match. As a result, a rule graph matches a graph if the non-NAC part matches the graph, and if the complete rule graph

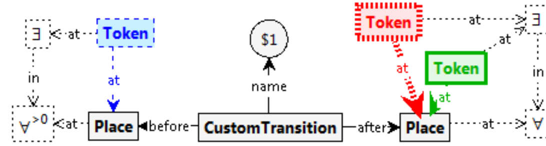

Figure 4.4: Example GROOVE rule customFire describing the semantics of a customized transition.

(including the NAC part) does *not* match that graph.

The second feature concerns typing. As mentioned above, the GROOVE version used by the DMM tooling does not have explicit support for type graphs. However, it provides limited typing support: A label can be prefixed with `type:`, followed by the type's name; this is possible in both host and rule graphs. Additionally, a *type hierarchy* can be defined, which is a set of (unique) type names and their inheritance relations to each other. The effect is that when computing matchings, a rule node can not only be mapped to a node having the exact same type, but also subtypes of the rule node's type. For instance, a type hierarchy could look as follows: `A > B, C; B > C, D; E > F`. The semantics is that `B` and `C` are subtypes of `A`, `C` is a subtype of `B`, and `F` is a subtype of `E`.

Let us complete our introduction to graph transformation in GROOVE by extending the above Petri net example. We introduce a type `CustomTransition` as a subtype of `Transition`. Then, we define the semantics of the new element by means of a second rule customFire, which is depicted as Fig. 4.4.

The new rule differs from the rule of Fig. 4.2 in three places. First, the node in the very middle is now typed `CustomTransition`, making sure that the node can not be mapped to an ordinary `Transition`. Second, the rule contains a NAC: The left `Token` node (red, thick dashed) of the right `Place` node makes sure that the rule can only match if the target `Places` of the `CustomTransition` do not carry `Tokens`.

The third difference is more subtle: compared to Fig. 4.2, the node labels of the customized rule of Fig. 4.4 are typed bold face. By that, GROOVE indicates that these are not simple labels, but labels annotated `type:` (and thus participating in the type hierarchy as explained above).

The new rule realizes the semantics of the new element. However, the basic fire rule does also match if the new rule matches. This is because the new rule's graph is an extension of the basic rule's graph. The only exception is the `CustomTransition` type; however, due to the type hierarchy of `Transition` and `CustomTransition`, the `Transition` node of the basic rule can also be mapped to a state node typed `CustomTransition`.

In this example, the situation can be resolved by adding a NAC to the basic rule which explicitly forbids to map the `Transition` node to a node typed `CustomTransition` – the change is rather simple, and thus we do note show the resulting rule. Note that we will use this mechanism later in this thesis to realize *rule overriding* (Sect. 6.3.5).

Let us now make use of the two rules by changing the type of the "B" `Transition` to the new `CustomTransition`. The resulting LTS is depicted as Fig. 4.5. It is considerably smaller than the one of Fig. 4.3, the reason

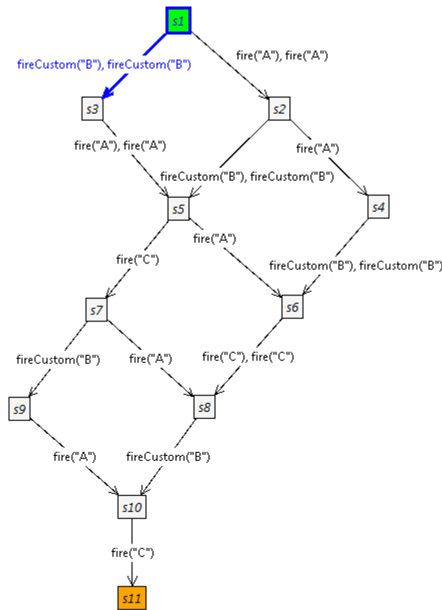

Figure 4.5: Labeled transition system resulting from the graph and rules as described in the text.

being that some options of execution are not available any more. In particular, `CustomTransition` "B" can not fire consecutively any more; `Transition` "C" needs to fire before the second token can pass `CustomTransition` "B". This results in a smaller state space.

Now that we have a precise understanding of GROOVE and its notion of graph transformation, let us investigate the analysis of the resulting LTS in the next section.

4.2 State Space Exploration

We have seen above that GROOVE—given a start graph and a set of graph transformation rules—can be used to compute the state space, i.e., all states reachable from the start state by recursively applying the rules. It should be noted that this is in fact a quite distinguishing feature: From the existing graph transformation tools, only one other tool (AUGUR, see [121]) is capabale of this task. The other tools can be used for linear exploration mainly. See [82] for a comparison of existing graph transformation tools with special respect to analysis capabilities.

GROOVE allows to explore a graph grammar's state space in different ways. The most simple ones are *breadth-first* and *depth-first*, which realize the well-known graph exploration stategies. In the case of *linear* exploration, only one transition will be followed per state. However, the resulting linear trace is deterministic for a running instance of GROOVE; if the aim is to explore a random path, the *random linear* exploration is the stratey of choice. Of particular interest is the *linear confluent* exploration; the idea is that some rules can be

declared as being confluent (which roughly means that they will always result in the same final graph, no matter in which order the rules are applied). Linear confluent exploration follows each transition of every state found, except for confluent rules: If more than one of those rules match a state, only one rule is chosen and applied. Finally, a *conditional* exploration is available; here, a certain rule is chosen as a termination criterion – states resulting from applying that rule will not be explored further, therefore restricting the size of the explored state space.

Each of the above exploration strategies can be combined with an *acceptor*, and the exploration can be configured to be interrupted as soon as the acceptor's condition has been fulfilled for a certain, configurable number of times. The most simple acceptor is the *final* acceptor which accepts a state that does not have any outgoing transitions. The *check variant* acceptor is bound to a certain rule and accepts as soon as that rule is applicable; i.e., it stops exploration before application of that rule, in contrast to the *rule application* acceptor which accepts as soon as the bound rule has been applied.

4.3 Model Checking GROOVE Grammars

Using the exploration strategies and acceptors as described in the previous section, GROOVE can compute the state space of a graph and transformation rules in the form of a *labeled transition system* (LTS). In this section, we introduce the model checking capabilities GROOVE offers for analyzing LTS. Before we do that, let us give a brief introduction to *temporal logic*.

4.3.1 Temporal Logic

With software systems getting more and more complex, the need arose to *verify* properties of such systems, for instance in the area of *safety-critical systems*. In general, two kinds of such properties are of interest: *safety* properties ensure that a system will never enter some (failure) state, whereas *liveness* properties ensure that a system will—at some point in time—enter some desired state.

But how to formulate such properties? This is where temporal logic comes into play, which has first been suggested by Pnueli [165]. Temporal logic allows to express statements such as "I will *always* be hungry", "I will *eventually* be hungry", or "I will be hungry *until* I eat something".[1]

For this, *temporal operators* are defined. For instance, the **G** operator expresses that something shall always be true (**G**enerally), the **F** operator expresses that something shall happen somewhere in the future (**F**uture), and the **U** operator expresses that some property p will be true until a property q holds.

Let us formulate our example properties in terms of temporal logic. For doing this, we need to define two predicates: let h be true if I am hungry, and let e be true if I eat something. Formulating the properties is then straightforward: The first translates to $\mathbf{G}h$, the second to $\mathbf{F}h$, and the last translates to $h\mathbf{U}e$.

Now, given an LTS we can use temporal logic to formulate properties about the *traces* of the LTS, i.e., the possible pathes of execution through the LTS, starting at the start state. The semantics of the above formulas is thus defined

[1] Examples from Wikipedia (http://en.wikipedia.org/wiki/Temporal_logic).

on the traces. Let us again consider an example: A trace can be described as a sequence of states, $t := s_0 s_1 \dots s_n$. An LTS gives rise to a number of traces T, i.e., all traces through that LTS. Let $p(s)$ be true if state s fulfills property p (q similar). Now, the temporal logic formula $\mathbf{F}p$ is true if and only if $\forall t \in T \exists i \in \{0, \dots, n\} : p(s_i)$. The other two properties are formalized similarly: $\mathbf{G}p$ translates to $\forall t \in T \forall i \in \{0, \dots, n\} : p(s_i)$, and $p\mathbf{U}q$ is true if $\forall t \in T \exists i \in \{0, \dots, n\} : (j < i \Rightarrow p(s_j)) \wedge q(s_i)$. Other temporal operators (which we do not define here) are ne**X**t ($\mathbf{X}p$ if p holds in the next state) and **R**elease ($p\mathbf{R}q$ if q is either always true or until the first occurence of p). Note that all above temporal operators consider the future; there are also operators considering the past, which can however be expressed by combinations of the future operators and are thus not considered here. Note also that **X** and **U** are the *fundamental operators* of the above temporal logic, meaning that all other operators can be expressed in terms of these two operators and the fundamental operators of predicate logic $\vee$ and $\neg$.

We have seen that the above temporal logic operators are considering complete traces, and that they only hold if all traces of an LTL fulfill the according temporal logic formula. Therefore, the temporal logic dialect presented above is called *linear-time temporal logic* (LTL). However, there are some interesting properties which can not be expressed by LTL. For instance, sometimes we want to express that from all states, a certain state *can* be reached. This can of course be true for traces which do not actually reach that state. LTL is not able to express such a requirement.

As a consequence, other temporal logic variants have been developed, an important one being *computation tree logic* (CTL) [60]. Here, the idea is to combine each temporal operator with a *path quantifier*, which can either be **E** (there exists a path) or **A** (on all paths). This results in expressions such as $\mathbf{AG}p$, $\mathbf{AF}p$, or $\mathbf{A}(p\mathbf{U}q)$. In particular, we are now able to express the above requirement: $\mathbf{AGEF}p$ is true if on all pathes, it is always true that there exists a path such that finally p is true. Note, however, that as there are CTL expressions which can not be expressed with LTL, the opposite is also true: for instance, the LTL formula $\mathbf{FG}p$ can not be expressed by means of CTL.

Finally, the temporal logic CTL* unifies both LTL and CTL: all LTL and CTL expressions can also be expressed by CTL*. See [61, 205] for in-depth discussions of the relation of LTL and CTL.

4.3.2 The GROOVE Model Checker

The GROOVE tooling provides means to model check the labeled transition systems representing the state spaces of start graphs and graph transformation rules. Temporal properties can be expressed with both LTL and CTL.

However, GROOVE does not perform model checking about properties of the state graphs of the LTS. Instead, temporal logic expressions are formulated over the transition labels, i.e., about the applications of the rules. Note that this indeed reveals knowledge about the states themselves: If an LTS contains a transition (s, l, s'), we know that state graph s contains structures such that the rule represented by label l has matched s.

Thus, we are now able to model check the labeled transition systems of Figs. 4.3 and 4.5. For instance, let us make sure that `Transition` “C” fires

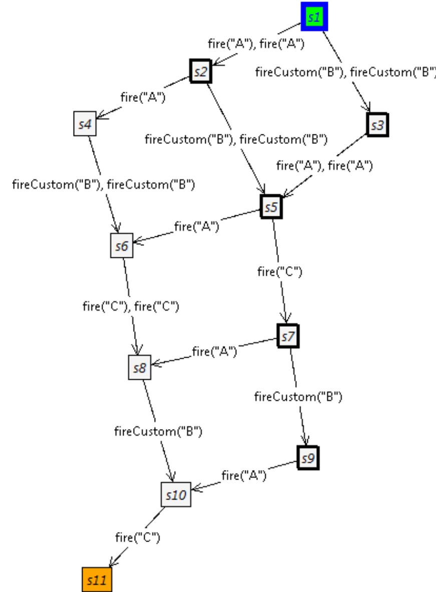

Figure 4.6: Counter example for property **AG**(!fire("A")).

two times in both cases. The CTL expression to check is

$$\mathbf{AF}(\text{fire("C")}\&\mathbf{AXAF}(\text{fire("C")}));$$

it makes sure that on all paths, transition label fire("C") occurs, and from that state on, it occurs again on all paths.

Feeding the above CTL formula into GROOVE produces the same answer for both LTS: "No counter example". This is how GROOVE (and all model checkers) answer a request for the validity of a temporal logic formula: If the formula does not hold, the model checker provides a counter example, i.e. a path starting from the start graph on which the property is violated. This counter example can then be used to understand the reason for the system not behaving as desired (assumed that the temporal logic expression is correct). Since GROOVE did not find a counter example, we know that rule fire("C") has indeed been applied two times, corresponding to the "C" *Transition* firing two times.

Let us now model check the property **AG**(!fire("A")). Obviously, the property does not hold for any of the two LTS, and GROOVE indeed reports counter examples; the one for the LTS of Fig. 4.3 is depicted as Fig. 4.6. Note that all states are marked which have an outgoing transition with label fire("A"), which indeed violate our property.

4.4 Tool Support

The GROOVE tool is developed at the University of Twente by Arend Rensink. The implementation language is Java, thus, the tool can be run on basically any platform. The pictures within this section have been created with the help of GROOVE. GROOVE consists of the following components:

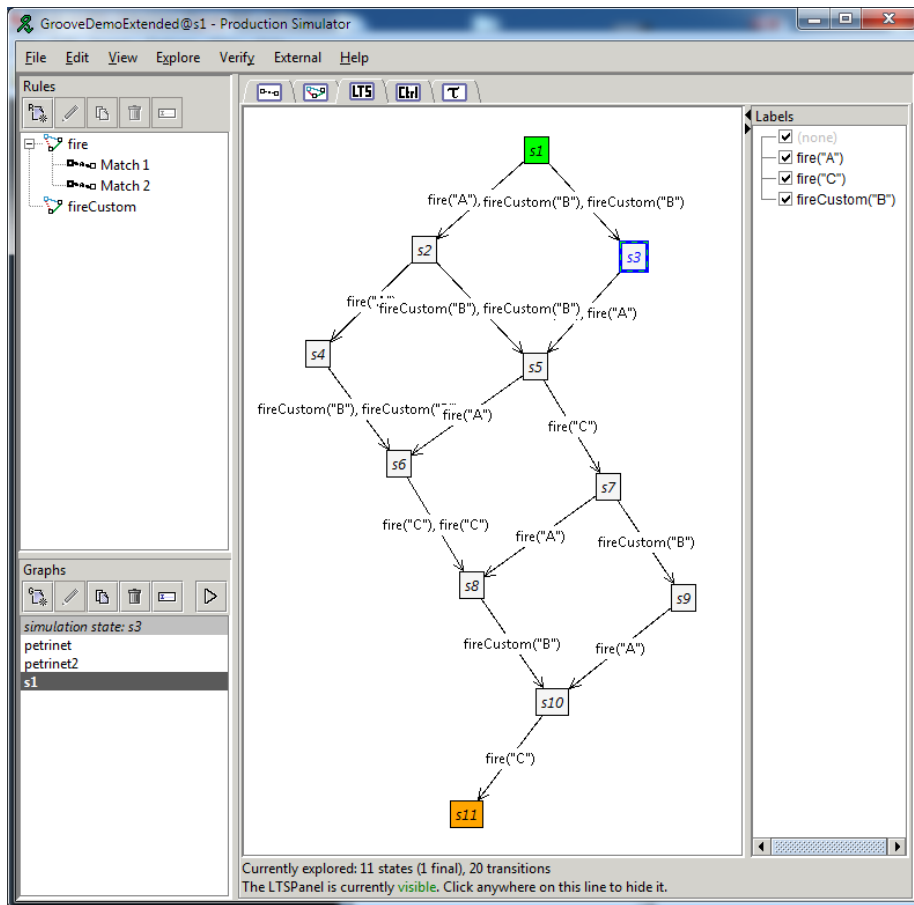

Figure 4.7: Screenshot of the GROOVE simulator.

- The GROOVE editor allows for the visual specification of GROOVE grammars. In particular, state graphs as well as rule graphs can be edited.
- The GROOVE imager allows to automatically layout GROOVE graphs, to display them (which is used in the editor), and to export those graphs into various formats.
- The GROOVE generator takes a state graph s_0 and a set of graph transformation rules $r_0, \ldots, r_n$ as input and computes a labeled transition system (LTS), where s_0 serves as the start state, and transitions are applications of the rules (with which they are labeled).
- The GROOVE model checker component accepts LTL and CTL expressions, evaluates them on the given LTS, and—if they do not hold–provides a counter example.
- Finally, the GROOVE simulator combines all above components into a single user interface which allows to explore transitions systems stepwise by applying one of the (possibly many) currently matching rules and to browse the resulting LTS.

Figure 4.7 shows a screenshot of the GROOVE Simulator. In the middle, an LTS can be seen (it is the one of Fig. 4.3). To the bottom, some information about the LTS is displayed. To the left, the rule and start state views are located. The rule view shows the rules' matches for the current state, which is s_3 (it is selected in the LTS view). To the right, the rule labels occuring in the LTS are depicted; these can be emphasized or hidden for the sake of better understandability of the LTS.

In addition to its graphical user interface, the GROOVE tool also provides a Java API, which e.g. allows to load GROOVE grammars, generate LTS, and model check temporal properties. The DMM++ tooling makes use of this API at several places.

Summarizing, GROOVE is a powerful toolset for performing graph transformations with emphasis on computing the state spaces of graph grammars, i.e., a set of GROOVE graph transformation rules and an according start graph. A state space can then be explored by means of model checking techniques, for which GROOVE supports the temporal logic dialects LTL and CTL. The DMM++ tooling makes heavy use of the GROOVE toolset as a graph transformation and model checking engine, which is achieved through GROOVE's Java API.

5 Dynamic Meta Modeling

The first paper concerning Dynamic Meta Modeling (DMM) has been published as early as 2000 [63]. DMM has been inspired by Plotkin's *structural operational semantics* (SOS) paradigm [163]. Plotkin argued that the semantics of a computer program can be expressed in terms of transition systems, where states are states of execution of the program, and consecutive states of that program give rise to transitions between the states of the transition system representing the program's semantics.

Plotkin suggested to express such semantics by means of *operational rules* defined on the syntactic structure of the program as well as the data the program works with: "The announced 'symbol-pushing' nature of our method suggests what is the truth; it is an operational method of specifying semantics based on syntactic transformations of programs and simple operations on discrete data." [163, p. 20].

In [63], Engels et al. have incorporated this approach in the context of visual languages whose syntax is defined by means of a metamodel (we have seen in Sect. 3 that the UML is such a language). The idea is as follows: First, the syntax metamodel is enhanced by concepts needed to express states of execution of a model; the new metamodel is called *runtime metamodel*. Second, operational rules are defined which work on instances of the runtime metamodel (i.e., *runtime models*). A transition system representing a model's semantics therefore has runtime models as states and applications of operational rules as transitions.

This section is dedicated to pointing out the current state of DMM as defined by Hausmann in his PhD thesis [96]. In the next section, we will explore the goals and requirements put on DMM in more detail. Section 5.2 will then briefly show the syntax and semantics of DMM specifications, and Sect. 5.3 will discuss the benefits and drawbacks of the current state of DMM.

5.1 Goals of and Requirements on DMM

When using a visual modeling language such as the UML, every involved person needs to have a clear understanding of the *semantics* (i.e., the meaning) of the models created. Otherwise, the models will be source of misunderstandings, and the benefits of using a visual modeling language will be nullified to a great extent. However, the UML specification [158] does not foster such a clear understanding for a number of reasons: First, the UML's semantics is expressed in natural language which is obviously subject to human interpretation. Second,

the current version of the UML specification [158] consists of about 740 pages, and the semantics information even of single language element is sometimes spread over several parts (we have seen examples for this in Section 3.3).

Providing a language's description in natural language has another severe drawback: Such descriptions can not be processed automatically. This rules out techniques such as automatic consistency checking of the semantics specification or derivation of language implementations (e.g., a workflow engine executing UML activities by means of interpreting the semantics description). Last but not least, there is no way to prove that a manual language implementation is consistent with the semantics specification.

Consequently, Hausmann requires that a semantics specification language must be *formal*,[1] *precise*, and *analyzable*, thus eliminating the drawbacks discussed above.

Additionally, the target audience of a semantics specification should obviously be able to understand that specification. In the case of the UML, Hausmann identifies two user groups: *language engineers*, e.g., the people who have created the UML, and *advanced language users*, i.e., users who “have a deeper interest in its [the UML's] inner workings, e.g., academics, tool builders, writers of UML books, UML consultants, and generally people who employed UML in such a breadth and depth as to be aware of the detailed problems it comprises” [96, p. 21].

As a result, Hausmann requires a semantics specification language to be *highly understandable* for advanced language users.

One advantage of natural languages is their flexibility. For instance, some parts of the UML specification are descriptive, while others are operational; some explanations are on a rather abstract level, while others are much more concrete. The actual usage of natural language depends on what is appropriate in the particular context.

Accordingly, the resulting requirements are that a semantics specification language must be *adequate* and *universal*.

5.2 Dynamic Meta Modeling

In this section, we will see how DMM is defined, and why its definition addresses the requirements stated above. We first provide an overview of DMM in Sect. 5.2.1, followed by a more detailed investigation in Sect. 5.2.2.

5.2.1 Overview

According to [96], a DMM specification consists of three parts, each of which is briefly discussed in the following sections. We start with the *runtime metamodel* in Section 5.2.1.1, which adds runtime information to the metamodel defining the language's syntax. Section 5.2.1.2 then discusses the relation between syntax and runtime metamodel, which can be defined by means of *meta relations*, and Section 5.2.1.3 describes how the actual behavioral semantics of the language is defined by means of operational rules. Finally, Section 5.2.1.4 shows how a DMM specification can be used to compute transition systems which describe

[1] Note that we are only giving intuitions for the requirements Hausmann identified; for a more thorough discussion, please refer to Sect. II.2.4 of [96].

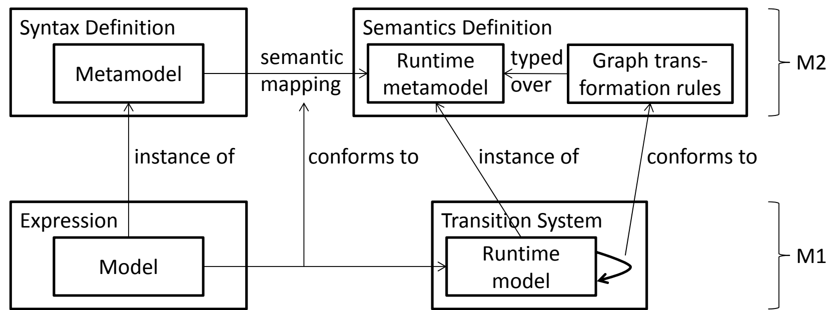

Figure 5.1: Overview of DMM (after [96, p. 42]).

the complete behavior of the according models. An overview of DMM is provided as figure 5.1.

5.2.1.1 Runtime Metamodel

As we have already seen in Section 1.1, the syntax of the UML is defined by means of the UML metamodel. However, that metamodel does not contain any information concerning the execution of behavioral models. For instance, we have seen in the last chapter that the semantics of UML activities is based on the concept of token flow, but the UML metamodel does not contain a `Token` concept. As such, instances of that metamodel can never contain tokens and thus cannot express states of execution of an activity.

This is where the runtime metamodel comes into play – it contains all concepts necessary to actually execute a model. An example runtime metamodel which has been provided by Hausmann as part of a case study is depicted as Fig. 5.2. It shows several concepts which are needed to describe states of execution of an activity. For instance, to the bottom left we can see that two kinds of `Tokens` exist: `ControlTokens` model simple flow of control, whereas `ObjectTokens` are used in the case of object flow; consequently, the latter kind of tokens has a relation to a single `Object` representing the actual data flowing through the activity. The `Offer` concept is used to model the fact that "... a source node can only offer tokens to the outgoing edges, rather than force them along the edge" [158, p. 327].

The reader familiar with the UML metamodel of activities will notice that the runtime metamodel of figure 5.2 contains the concepts of `Node` and `Edge`, whereas in the UML metamodel, we have `ActivityNode` and `ActivityEdge` classes [158, p. 307]. Moreover, in the UML metamodel, the class `Activity-Edge` is abstract and is refined by classes `ControlFlow` and `ObjectFlow`, which do not appear at all in the runtime metamodel. The reason for this is that "semantically, ... there is no difference in how the different kinds of edges treat the passing tokens. We do thus integrate both concepts into a single class" [96, p. 240].

Concluding, a DMM runtime metamodel contains all concepts needed to express states of execution of the target language's models. It usually contains concepts of the syntax metamodel and runtime enhancements, but is not nec-

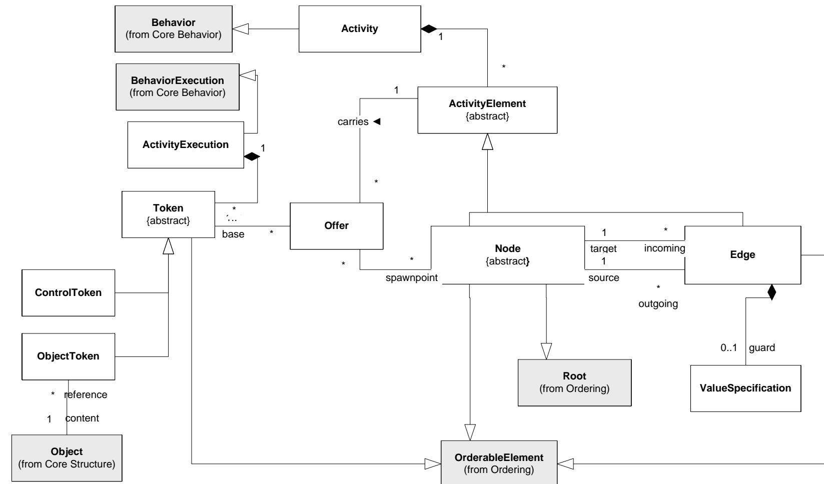

Figure 5.2: Excerpt of the runtime metamodel (from [96, p. 229]).

essarily an "enhanced version" of the syntax metamodel.[2] Thus, in the next section we will investigate how syntax and runtime metamodel are related to each other by means of *meta relations*.

5.2.1.2 Meta Relations

In the last section, we have seen that instances of the runtime metamodel correspond to states of execution of models (i.e., to instances of the syntax metamodel). But how are the two metamodels related to each other? This question is not only relevant on the type level, but also on the instance level. As such, a technique is needed which allows to define relations between metamodel elements on the type level, and to instantiate these relations on the instance level. This is what meta relations can be used for.

Let us make this more clear using our running example of UML activities. An excerpt of the meta relations as defined by Hausmann [95] is depicted as Fig. 5.3. In the last section, we have seen that the UML concept of `ActivityEdges` is reflected by the runtime concept `Edge`. Now, we not only want to be able to express this very fact, but, we also need to make sure that for a concrete UML activity containing several `ActivityEdge` objects, we keep track of the according `Edge` object. Relating the metaclasses happens on the type level, and relating the actual instances of the metaclasses happens on the instance level.

To understand how this is achieved technically, let us briefly consider the metamodel of meta relations which is depicted as figure 5.4, and which reveals that the elements actually used to model relations (metaclasses `Relation` and `Tuple`) both inherit from `Classifier`. As such, relations can be defined on the type level by linguistically instantiating these concepts, and applied to concrete models by ontologically instantiating them (see [8] for more information on linguistic and ontological instantiation).

[2] In Section 7.2, we will show an alternative way to define a runtime metamodel by decorating the syntax metamodel with runtime information.

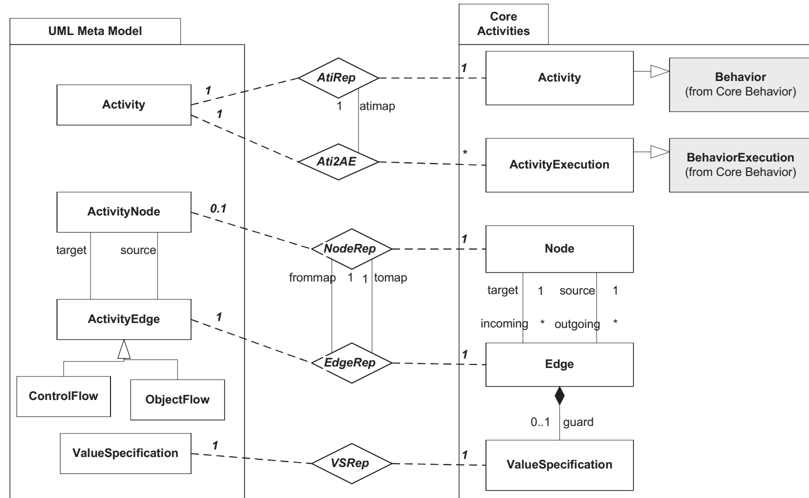

Figure 5.3: Excerpt of the UML activity meta relations (from [96, p. 243]).

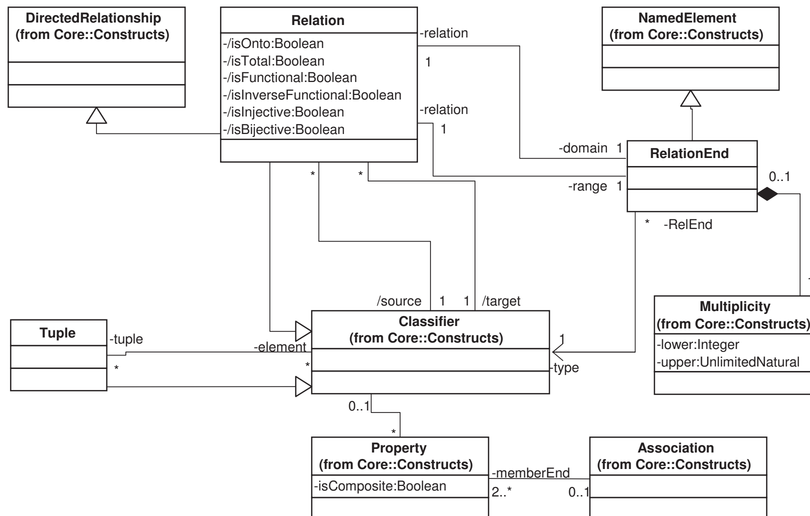

Figure 5.4: Metamodel of the meta relations language (from [96, p. 56]).

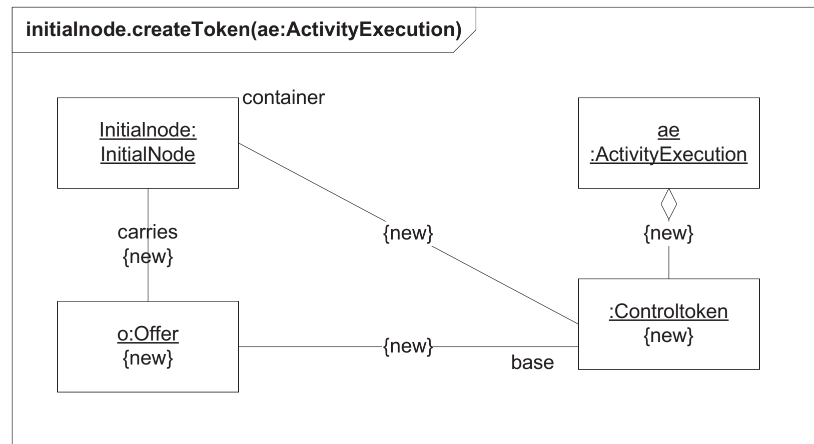

Figure 5.5: Example DMM rule initialNode.createToken() (from [96, p. 247]).

Note that meta relations can only be a base for an executable transformation from syntax models into runtime models – this can easily be seen in Fig. 5.3: The meta relation `Ati2AE` connects the `Activity` metaclass with the runtime concept `ActivityExecution`. However, the cardinality of `ActivityExecution` in this relation is $*$. A language giving rise to an executable model transformation would obviously need to be more precise at this place.

Concluding, meta relations are a pragmatic and visual means to specify the relations between elements of the syntax and the runtime metamodel. They can be used to define these relations on the type as well as the instance level. Meta relations are not executable.

5.2.1.3 Operational Rules

So far, we have defined the semantic domain of our language by means of the runtime metamodel, and we have related the syntax and the runtime metamodel by means of meta relations. The next step is to define the language's behavior. For this task, DMM uses operational rules. The idea is that each state of execution of a model corresponds to an instance of the runtime metamodel; the operational rules basically describe how these instances change through time (and, thus, lead to new instances).

To get a first impression of DMM rules, let us again consider an example. Figure 5.5 shows an example DMM rule. Note first that the rule's concrete syntax is visual, therefore making it relatively easy to understand the rule's semantics. In fact, Hausmann has re-used the visual appearance of UML *communication diagrams* to further strengthen DMM's understandability, as these kind of diagrams is expected to be well-known to advanced language users.

However, some elements are annotated with the non-UML stereotype `{new}`. This is the actual behavior of the rule: If it is applied, it will create `Token` and `Offer` objects and associate them with the according elements. In other words: After application of the rule, the `InitialNode` object will carry a `Token` and an `Offer` which are now ready to flow through the activity.

Technically, DMM rules are *typed graph transformation rules* as we have seen them in Chapter 4. Just like GROOVE the concrete syntax of DMM rules merges the two graphs into one single graph; DMM uses the stereotypes `{new}` and `{destroyed}` to annotate the according nodes and edges.

Note that DMM does not provide a mechanism that allows to let one rule *refine* another rule, be it within a single ruleset or across rulesets.

Concluding, DMM's operational rules have a visual, communication-diagram-like concrete syntax and are therefore precise yet easily understandable. They manipulate instances of the runtime metamodel which correspond to states of execution of the model at hand.

5.2.1.4 Labeled Transition Systems

One of the most important goals of creating a DMM specification for a visual modeling language is to reason about models of that language in a formal and thus automatable way. In this section, we will briefly discuss how this can be done.

The first step consists of translating the abstract syntax of the model to be analyzed into an instance of the runtime metamodel. For this, the meta relations are used: As we have seen above, they describe which syntax metamodel concepts are mapped to which runtime metamodel concepts.

Now, a *labeled transition system* (LTS) is defined as follows: The runtime model serves as the start state of the LTS. On that state, all matching rules are applied, giving rise to a set of new states. These states are added to the LTS, and they are connected to the initial state by means of a directed edge which is labeled with the applied rule's name. Then, the process is repeated with the newly found states until no new states are found. The resulting LTS contains the whole behavior of the model under consideration.

5.2.2 Language Definition

Now that we have a given a brief overview of the DMM approach, we will investigate the DMM semantics specification language in more detail. We have seen above that the backing formalism of DMM are graph transformations; consequently, in Section 5.2.2.1 we start with a discussion of the different types of graphs DMM makes use of, followed by an introduction to the notion of typed graph transformation rules of DMM. Section 5.2.2.2 will then investigate more advanced features of DMM rules such as negative application conditions (NAC) and universally quantified structures (UQS). Finally, Section 5.2.2.4 will present Hausmann's ideas on translating DMM specifications into graph grammars suitable for the graph transformation tool GROOVE [166]. This section basically is a summary of parts of Chapters IV and VIII of [96].

Note that our goal is not to provide a comprehensive definition of DMM within this section. The interested reader is pointed to either [96] or [189] (the latter fixes a couple of minor flaws contained in Hausmann's original definitions).

5.2.2.1 Graphs in DMM

DMM specifications employ three different, but related kinds of graphs: *Type graphs* are used as a mathematical model for the representation of metamodels,

instance graphs represent instances of metamodels (i.e., models), and *rule graphs* are used within the graph transformation rules.

DMM type graphs basically follow the well-known definitions of typed graphs as can e.g. be found in [29, 9]: A type graph consists of a set of nodes, a set of directed edges, an injective labeling function associating nodes with type names, a set of inheritance edges, a set of abstract nodes, and a set of datatype nodes.

DMM instance graphs as well as DMM rule graphs both consist of a simple graph and a typing morphism from the graph's nodes into a DMM type graph's nodes.

Let us investigate some of Hausmann's definitions for the sake of getting an impression of how he defined DMM. We start with the *DMM Type Graph*:

Definition 1 (DMM type graph) *A graph* $G_{TG} = (V, E, l_v, I, A, DT)$ *is called a* DMM type graph*, where* V *is the set of nodes,* Σ *is some alphabet,* $E \subset V \times \Sigma \times \Sigma \times \Sigma \times V$ *is the set of labeled edges,* $l_V \subset V \to \Sigma$ *is an (injective) labeling function,* $I \subset V \times V$ *is the set of* inheritance edges *which must not contain circles,* $A \subset V$ *is the set of* abstract *vertices, and* $DT \subset V$ *is a set of datatypes.*

This definition formalizes quite a few features of UML class diagrams: Datatypes are represented by DT, classes are represented by $V \setminus DT$ (and are additionally contained in A if they are abstract), and associations are represented by E; class $v_2 \in V$ is a subtype of $v_1 \in V$ iff there exists an edge $(v_1, v_2) \in I$; a class v has an attribute of datatype $d \in DT$ if there exists an $e \in \{v\} \times \Sigma \times \Sigma \times \Sigma \times \{d\}$. The three labels carried by an edge refer to the two roles and the name of the according association or attribute.

A labeled graph can now be *typed over* a type graph by means of an *edge-label-preserving morphism* or elp-morphism.

Definition 2 (Elp-morphism) *An* elp-morphism *is a relation between two graphs* G, H*:*

$$G \sim_{elp} H :\Leftrightarrow$$

$$\exists m : V^G \to V^H \; \forall (v_1, l_1, l_2, l_3, v_2) \in E^G : (m(v_1), l_1, l_2, l_3, m(v_2)) \in E^H$$

Sometimes the function m *is also called elp-morphism.*

The idea is that for every edge in G there is a corresponding edge in H which connects "the same" vertices (i.e., the vertices of H to which the vertices of G are mapped). This can now be used to define DMM rule and instance graphs. We only show the (slightly simplified) definition of the DMM rule graph here:

Definition 3 (DMM rule graph) *Let* G_{TG} *be a DMM type graph. A* DMM rule graph $G_R = (G, type)$ *is a graph* $G = (V, E, l_V)$ *with a typing elp-morphism type between* G *and* G_{TG}*.*

The above definition abstracts from the fact that a type graph as defined above distinguishes abstract from non-abstract types and additionally contains inheritance relations. In Hausmann's actual definition, this is dealt with by flattening the type graph into an *abstract (concrete) closure*, over which the rule (instance) graphs are typed.

Based on that, a *DMM rule* is defined as follows:

Definition 4 (DMM rule) *A* DMM rule $r = (L, R)$ *consists out of two DMM rule graphs L and R, where L and R are called* left-hand *and* right-hand *graphs.*

At this definition, Hausmann slightly varies the common definition of graph transformation rules, where a morphism is used to "connect" vertices being contained in both L and R. Instead, he assumes that L and R are defined over a common set of vertices.

Now, a DMM rule $r = (L, R)$ is said to *match* a DMM instance graph G if there exists an injective elp-morphism m from L to G. If a DMM rule matches a DMM instance graph, the rule can be *applied*, giving rise to a new graph G'. The basic idea is that for each vertex $v \in L \setminus R$, $m(v)$ is removed from G, and for each vertex $v \in R \setminus L$, $m(v)$ is added to G. Edges are treated similarly: if one or both vertices of an edge are in $L \setminus R$, the edge is removed, and if one or both vertices of an edge are in $R \setminus L$, the edge is added. The resulting graph is G'.

Finally, given a set of DMM rules $r0, \ldots, r_n$ and an initial DMM instance graph G_0, a transition system can be computed as follows: every rule r_i which matches G_0 is applied to G_0, leading to new DMM instance graphs $G_1, \ldots, G_m$. This is repeated for all newly found DMM instance graphs until no new graphs are found. Note that the resulting transition systems can be of arbitrary size (even infinite).

5.2.2.2 Advanced Features

In addition to the formalization of DMM rules, instance graphs etc., DMM supports a number of additional features which we want to briefly present within this section.

Negative Application Conditions In many situations it is useful to not only describe which structures must exist in an instance graph for a DMM rule to match, but also to describe structures which *must not* exist for that rule to match. For this, DMM supports the concept of *negative application conditions* (NACs). A NAC is an extension of the left-hand graph of a rule. A DMM rule $r = (L, R)$ can have several NACs $N0, \ldots, N_n$. If this is the case, that rule will only match an instance graph G if there exists an injective morphism from L to G, and if for all NACs N_i, no injective morphism from N_i to G exists. More information on NACs can be found in [96, p. 77].

Universally Quantified Structures Another quite common use case is that one wants to treat *all* vertices of a certain kind in the same way. For instance, the semantics of UML activities we have developed contains a rule that creates a token on each `InitialNode` if the according activity is started.

In many cases, such behavior can be realized "manually": In our example, we could have provided a rule which would add a token to an `InitialNode` if that node does not yet carry a token. That rule would match and be applied over and over until all `InitialNodes` indeed carry a token. Another approach—which gives more control to the language engineer—would be to use rule invocations (see below) for that purpose. However, these approaches have two major drawbacks: They unnecessarily increase the resulting transition systems, and they tend to complicate the resulting DMM rulesets.

To cope with that, Hausmann has added basic support for *universal quantified structures* (UQS) to DMM: Only vertices can be marked as UQS – the vertices' adjacent edges then implicitly belong to that UQS. A rule can have more than one UQS. If vertices marked as UQS are directly connected by an edge, they form a single UQS. A rule containing one or more UQS matches the maximum number of elements of the graph under consideration which fulfills the rules' conditions.

Technically, rules containing UQS are *unfolded* to a number of plain graph transformation rules, each rule matching a fixed number of elements. We will explain this in more detail in section 5.2.2.4. More information on UQS can be found in [96, p. 79].

Rule Invocation One challenge when creating DMM semantics specifications is that many language engineers are not used to working with rule-based systems. The main difference is that in these systems, one for instance can not specify an explicit order of execution of rules; instead, the rules need to be designed such that they only match if it "makes sense". DMM improves on this situation by giving the language engineer a control mechanism called *rule invocation*.

This means that DMM rules have the possibility to *invoke* other rules. These (invoked) rules can only match if they are indeed invoked: Triggers[3] are used to activate them. DMM therefore provides two kinds of rules:

- Bigstep rules
  These rules do not need to be invoked; they basically act as "traditional" graph transformation rules, but may invoke smallstep rules.

- Smallstep rules
  These rules need to be invoked by a bigstep rule or another smallstep rule. They can not match without having been invoked.

The invocation mechanism of DMM enhances "normal" rules with a *sequence number* (in case several rules are invoked within one rule), a *context node* on which a rule is invoked, and a number of *parameter nodes* which can be passed to the invoked rule. We will learn more about rule invocation in Sect. 5.2.2.4.

Premise Rules The concept of rule invocation can be seen as an extension to the right-hand side of a rule: The effect of applying the rule is spread over several rules, each of them being smaller and therefore easier to understand and to maintain.

Premise rules are an extension to the left-hand side of a rule. If a rule r invokes a premise Rule p, r can only match if the structures contained in r and p do exist in the state graph. Thus, the premise rules invoked by r need to be merged into r, resulting in a "plain" DMM rule r'. The advantages of premise Rules are similar to those of smallstep rules: First, they allow for smaller left-hand sides of rules; second, premise rules can be invoked by several other rules and thus allow for decomposition in the object-oriented sense.

[3] A *trigger* is a vertex having a special purpose, e.g. enabling a certain rule; the rule can only match if the trigger is contained in the graph. Triggers have been introduced in [80].

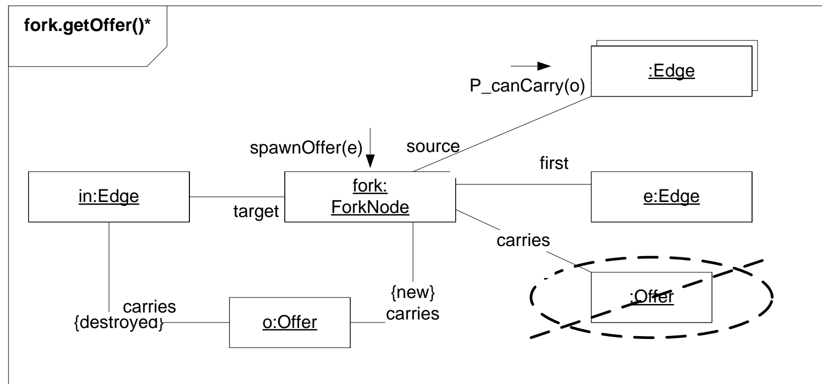

Figure 5.6: Example DMM rule fork.getOffer()* (from [96, p. 261]).

5.2.2.3 Visual Representation of DMM Rules

We have seen above that DMM rules closely follow the visual appearance of UML communication diagrams. However, the rule introduced as Fig. 5.5 on page 42 did not contain any of the advanced features of DMM. In this section, we want to briefly investigate the visual appearance of those features.

Fig. 5.6 shows a DMM rule which makes use of all of these concepts:

- Negative application conditions are annotated with a "stop sign". An example of this is the `Offer` node to the rule's right bottom.
- Universal quantified structures are drawn as multi-objects (see the `Edge` object to the rule's upper right).
- Invocations of smallstep rules are drawn as arrows pointing to the invocation's target node, and labeled with the invoked rule's name. Additionally, parameters can be passed to the invoked rule, which are referenced by means of the parameter node's name. See the arrow labeled `spawnOffer(e)` pointing to the `ForkNode` object.
- Finally, Invocations of premise rules are drawn just like invocations of smallstep rules, but their label contains the prefix "P_".

5.2.2.4 Automatic Application of DMM Specifications

We have seen in the last sections that the DMM semantics specification language is completely defined formally. This has the benefit that the semantics of DMM specifications themselves is unambigiuos. A language engineer can therefore use a semantics specification for a language such as UML activities as a reference to figure out the precise behavior of UML activity models.

However, this usage has strong limitations – due to the complexity of the task at hand (the semantics specification of UML activities a provided by Hausmann contains 88 rules and still only covers about 60% of the language), only small examples can thoroughly be investigated. Therefore, an automatic application of DMM specifications is needed.

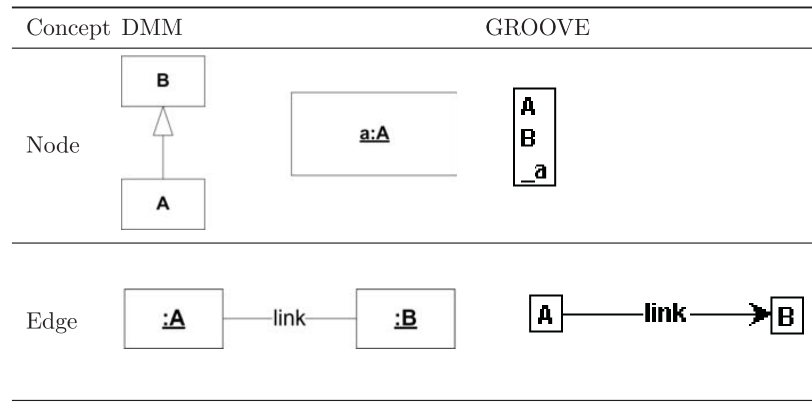

Figure 5.7: Graph concepts in DMM and GROOVE (from [96, p. 183]).

Since the backing formalism of DMM are graph transformations, Hausmann provided the sketch of a translation of DMM specifications into GROOVE grammars. The general idea of translation is straight-forward: each DMM node (edge) is mapped to a GROOVE node (edge). However, the notion of graph transformation used by DMM and GROOVE differs in quite a few details. Therefore, Hausmann needed to map certain DMM features to constructs available in GROOVE.

The first and most important distinction is that GROOVE[4] did not support the typing of graphs and graph transformation rules. Therefore, the typing information had to be added to state and rule graphs as follows:

- Each rule node carries a self-edge with the name of the node's concrete type.

- In the case of state graphs, each state node is equipped with self-edges for each type of that node. This makes sure that a rule node can be matched to a state node even if the state node's type is a subtype of the rule node's type (see Fig. 5.7 for an illustration).

Despite that, the translation of the basic graph transformation features such as node/edge creation and deletion is rather straight-forward: in GROOVE, the labels of elements to be created/deleted are prefixed with `new:`/`del:`.

The translation of DMM's negative application conditions is again simple: the labels Nodes/edges belonging to a NAC are prefixed with `not:`. Also rather straight-forward is the translation of invocations of premise rules: Since these rules are an extension of the left-hand graph of the invoking rules, the premise rules are merged into the invoking rules during translation.

Translating the universal quantified structures, however, is more complicated. Since at the time of writing of Hausmann's PhD thesis, GROOVE did not provide any support for UQS, Hausmann decided to *unfold* those structures: For every UQS contained in a DMM rule, a number of GROOVE rules are created, each of which taking care of a particular number of elements of

[4]At the time Hausmann was writing his PhD thesis.

that UQS. Since there is no upper limit for the number of occurences of a particular structure, the maximum number of occurences has to be fixed during translation.

Let us illustrate the latter with a simple example: The DMM rule fork.-getOffer()* we have seen as Fig. 5.6 on page 47 contains a single UQS (the Edge object in the rule's upper right corner). The translation performed by Hausmann as part of his PhD thesis results in four GROOVE rules handling the cases where zero, one, two, and three occurences of edges would appear in the state graph. In other words, the resulting GROOVE grammar has been created with an upper limit of three for the treatment on UQS.

The drawbacks of this approach are three-fold. First, and most important, the resulting DMM specification is not able to deal with models with more than three edges. Second, the resulting GROOVE rules are rather difficult to understand. Third, if a rule contains more than one UQS, all combinations of numbers of elements have to be contained in the GROOVE grammar. As such, if n is the upper limit for UQS treatment, and if a rule contains m UQS, the translation of that single DMM rule will result in n^m GROOVE rules.

Finally, the concept of *invocation of rules* has to be translated. For this, Hausmann has introduced an explicit *invocation stack* which is part of the GROOVE state graphs.

- Bigstep rules can only match if the stack is empty, i.e., if no smallstep rules are currently to be executed. If a bigstep rule does not contain any invocations, the invocation stack remains empty.

- A smallstep rule can only match if a node corresponding to that rule is on top of the stack. If this is the case, application of the rule will (besides the "actual" changes of the rule) remove that node from the stack.

- If a rule (be it a bigstep or smallstep rule) contains invocations, an Invocation node is pushed onto the stack for each of these invocations.

An example invocation stack is depicted as Fig. 5.8. In that state, a smallstep rule with name "do" has been invoked. Since the invocation stack is not empty (i.e., the invocation on top of the stack is not the "_bottom" invocation), no bigstep rule can match; instead, only the "do" rule can match. The context node of the invocation is connected to the invocation node by means of a self edge, and the rule's parameters are connected by means of param edges.

Fig. 5.9 gives an overview of the mapping of DMM application control concepts to GROOVE rules. At the top, we see how a new invocation of rule "make" is pushed on top of the stack. Below we see that a smallstep rule can only match if its corresponding Invocation node is on top of the stack, and that application of the rule will remove that node from the stack and activate the next invocation on the stack (which might be the "_bottom" invocation). A bigstep rule has an empty invocation stack in its application context. Finally, the invocation of a premise rule results in a GROOVE rule where the premise rule's structure has been merged into the invoking rule.[5]

[5]Note that the premise rule translation contains an error: The R node is neither contained in the invoking nor in the invoked rule.

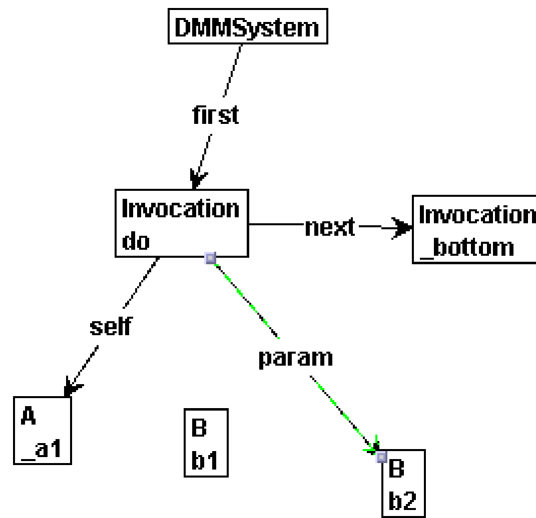

Figure 5.8: Example state of the invocation stack (from [96, p. 185]).

5.3 Evaluation of DMM's Current State

We have seen in the last sections that DMM as defined by Hausmann certainly has its benefits:

- It is defined completely formally, using the well-suited formalism of graph transformations. This ensures that the semantics of DMM itself is well-defined, which of course is an important requirement on semantics specification techniques. Additionally, DMM specifications can (at least in theory, see below) be analyzed, and even proofs (e.g. about a certain specification fulfilling some requirements) can be performed.
- By reusing UML communication diagrams, the concrete syntax for DMM rules developed by Hausmann can be expected to be easily understandable for advanced language users, i.e., users familiar with the language's metamodel.
- The description of a translation of DMM specifications into GROOVE grammars given in [96, Ch. 8] points the way towards tool support based on the DMM concept.

As such, DMM has found quite some attention in the academic community. For instance, O'Keefe [160] has investigated different possibilities for giving the UML a formal semantics; he—remarkably—concludes that DMM might be the most appropriate formalism for this task [160, p. 179]. Another example is the work of Baresi et al. [14], which has—according to their own words— been inspired by DMM. Finally we want to mention the work of Chiaradia and Pons [26], who suggest to improve the definition of the OCL by means of applying design patterns to the OCL metamodel, and by specifying the evaluation of OCL constructs by means of sequence diagrams (which are formalized by means of DMM).

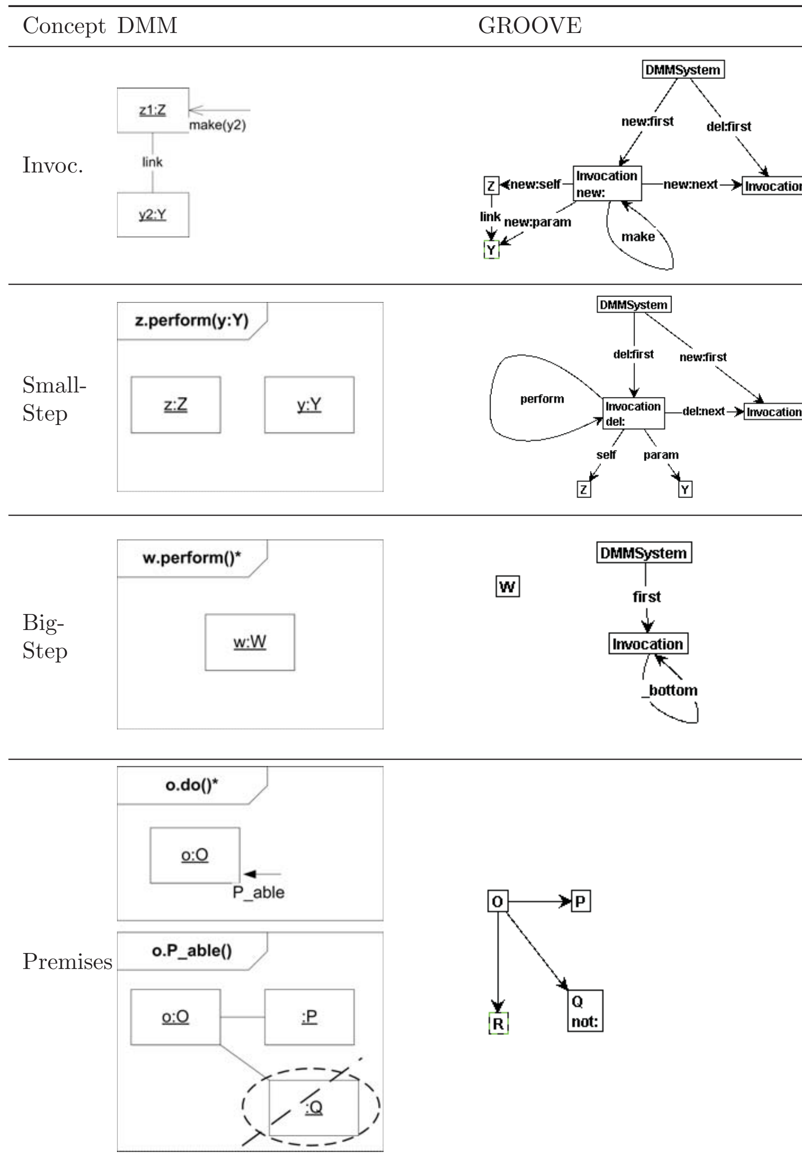

Figure 5.9: Realization of application control concepts in GROOVE (from [96, p. 188]).

However, the definition of DMM as provided by Hausmann focuses on concepts, but not so much on applications. This is reflected in [96] in many ways:

- The basic graph transformation theory Hausmann used to define DMM does not translate well into tool support. This is for a number of reasons:
  - Hausmann did not provide an explicit abstract syntax for DMM specifications.
  - The meta relations, which are used to relate syntax and runtime metamodel, are not executable (see Section 7.2). As such, in many cases they can only be a base for a model transformation which automatically translates instances of the syntax metamodel into instances of the runtime metamodel.
  - Hausmann has only very briefly described the treatment of attributes.
  - The description of translating DMM specifications into GROOVE grammars is very informal and only gives an idea on how to perform this automatically. Additionally, the issue of translating universally quantified structures has only been adressed conceptionally by Hausmann.
  - Additionally, to rely on the analysis results of GROOVE, one would have to formally prove that the translation of DMM specifications into GROOVE grammars is *semantics preserving*, i.e., that the resulting GROOVE grammars indeed realize the semantics of the source DMM specifications.
- Hausmann claims that due to its formal nature, language engineers can *prove* that their languages fulfill certain requirements. However, DMM specifications tend to be too complex to tackle such proofs. In fact, our experience has shown that even for very simple DMM specifications, such proofs seem to be very difficult (see [106, 107, 186]).
- Hausmann also claims that DMM allows to do automatic analysis of a model's behavior with model checking techniques, but does not describe how.
- The quality of DMM semantics specifications (which is absolutely crucial for analyzing the quality of models equipped with such a specification) is only addressed by means of pragmatic guidelines.
- The only way to reuse existing DMM specifications is by removing or adding DMM rules. However, DMM does not support any kind of *refinement* of rules.
- Finally, the formalization as presented by Hausmann in [96] contained a few minor flaws, which is a clear sign that that formalization has not been used in any of the above ways.

Summary of Part I

We have started the foundations of this thesis with an introduction to the Eclipse Modeling Framework (EMF), a framework around the metametamodeling language Ecore. We have seen that Ecore is an implementation of the OMG's EMOF standard, and we have investigated the Ecore metamodel with its core metaclasses `EPackage`, `ECLass`, `EReference`, and `EAttribute`. Finally, we have briefly investigated how the DMM tooling makes use of EMF and other EMF-based frameworks such as GMF, Eclipse OCL, or XText.

In Chapter 3, we have then introduced the language of UML activities, which serves as a running example within this thesis. Activities allow to model workflows, algorithms and other process-based behavior and have a semantics roughly comparable to Petri nets. The syntax of activities is defined by means of the UML's metamodel. However, the mentioned "Petri-like" semantics of the language is described with natural language, which has many drawbacks, most importantly:

- Natural language is subject of human interpretation. Therefore, the semantics of UML activities (and many other behavioral languages, not only the ones contained in the UML) is ambigious.
- Natural language can not be processed automatically by computers. As such, it is not possible to analyze the quality of models with analysis techniques such as model checking.

Chapter 4 has given an introduction to GROOVE, a toolset for performing graph transformation. GROOVE uses a notion of directed, labeled graphs and allows to formulate graph transformation rules with powerful features such as typing, full attribute support, negative application conditions, and nested quantification. Given a graph grammar, i.e., a graph and a set of graph transformation rules, the GROOVE toolset allows to generate the grammar's state space in terms of a labeled transition system (LTS), where states are graphs, and transitions are rule applications (and are thus labeled with the applied rule's name).

Additionally, we have got to know the GROOVE model checker, a component which allows to analyze an LTS by means of verifying temporal logic formulas against that LTS. GROOVE supports both linear-time as well as computation-tree temporal logic (LTL and CTL) and provides a counter example in case an LTL or CTL expression does not hold for the LTS at hand. The expressiveness of the expressions is increased by the fact that rule parameters allow to extend the LTS's labels with the runtime values of the source state graph's nodes (e.g., the rule modeling the execution of a UML `Action` can display the value of the actually executed `Action`'s `name` attribute as part of the transition label).

Finally, in Chapter 5 we have given a brief introduction to Dynamic Meta Modeling as defined by Hausmann in his PhD thesis [96], a semantics specification technique targeted at behavioral languages whose syntax is defined by means of a metamodel. We have started by defining the requirements on DMM, the most important of which are:

- For the sake of being unambigious and automatically processable, DMM specifications shall be *formal* and *analyzable*.

- To make it as easy as possible to work with DMM specifications, they shall be *highly understandable*.

We have shown how Hausmann meets these requirements by using *graph transformations* as DMM's underlying formalism, which are both formal and visual (the latter resulting in semantics specifications which are relatively easy to understand). We have given a brief introduction to the three parts a DMM specification consists of:

- The *runtime metamodel* enhances the syntax metamodel with concepts to express states of execution of models.
- *Meta relations* are used to relate concepts of the syntax and runtime metamodel on the type as well as the instance level.
- *Graph transformation rules* are used to describe the actual behavior of the models by formalizing how runtime models change through time.

We have concluded that although a great step into the right direction, DMM in its current state is more about concepts than implementation of those concepts, making it difficult to actually work with DMM specifications. For instance, due to a couple of reasons, the graph transformation theory underlying DMM does not translate well into tool support. It is also rather unclear how to create a DMM specification of high quality, which is crucial when analyzing the quality of models based on such a specification. Finally, it needs to be shown how a given high-quality DMM semantics specification can be used for quality analysis of behavioral models such as UML activities.

In this thesis, we will improve the DMM approach to overcome the practical issues discussed in the last paragraph. We will make use of EMF to define DMM's syntax, and we will define the semantics of DMM by means of mapping DMM specifications and according models to GROOVE grammars. We will then show how to create high-quality semantics by means of applying *test-driven semantics specification*, and we will briefly show the tool support we provide for creating syntactically correct DMM specifications. Finally, we will enable the formulation of functional and non-functional requirements against models equipped with a DMM specification, and we will show how to analyze and—if necessary—fix models using the DMM tool support.

Part II

Dynamic Meta Modeling ++

6 Language Definition of DMM++

In the last section, we have identified several potential improvements of the current state of DMM. In this chapter, we will introduce our extended version of DMM, which is a first step towards realizing those improvements.

In the next section, we will first give an overview of the changes we performed. We will then define the new state of DMM—which we call DMM++—by first giving an overview of DMM and its syntax in Section 6.2, followed by a discussion of the semantics in Section 6.3.

6.1 Comparison of DMM and DMM++

The differences between DMM as defined by Hausmann and our extended version of DMM can broadly be divided into two classes: *conceptual* and *pragmatic* changes. The latter refers to changes which make it easier to actually work with DMM specifications (remember from Chapter 5 that DMM had been defined completely formally, which in many cases does not translate well into tool support), whereas the former class refers to changes which introduce new concepts into the DMM language and therefore improve its expressiveness.

6.1.1 Conceptual Changes

One area where the current state of DMM can be improved is its handling of attributes, which is somewhat restricted. Attributes can only be used in two ways: to influence the matching of rules by specifying concrete values, and to specify the concrete values of attributes in case a node is created by a DMM rule. Note, however, that [96] does not provide any examples on these two cases: All information concerning attributes is contained on pages 67 and 69 of [96].

As a result, we have extended DMM by means of dedicated support for attributes which allows to

- influence the matching of a rule in a much more complex way. For this, we have implemented several operations for comparison of attribute values. See sections 6.2.3.10 and 6.3.4.2 for more information.

- compute new values for attributes. Again, we have defined several operations which can be used for this. See sections 6.2.3.11 and 6.3.4.3 for more information.

- refer to other nodes' attribute values when comparing or newly computing attribute values.
- show the values of attributes at rule application time in the resulting transition systems' labels, which allows to refer to these values when doing model checking. See sections 6.2.3.12 and 6.3.4.6 for more information.

The treatment of universally quantified structures (UQS) in the current state of DMM is powerful. However, its semantics is based on the *unfolding* of rules (see [96, p. 79]), the main reason being that GROOVE at that time did not provide support for UQS. In the meantime, a powerful notion of UQS has been added to GROOVE which we apply for re-defining the semantics of UQS in Sect. 6.3.3.

The area of reusability of DMM specifications has been identified as a target of potential improvement: We have seen in the last chapter that DMM only allows to add rules to a ruleset. To improve on this situation, we have added support for *overriding* of rules [62, 192]: A rule can either *completely* or *softly* override a given rule. See sections 6.2.2.12, 6.2.2.13, and 6.3.5 for more information.

Finally, we will see in Sect. 9.1.2 that model checking properties of a certain model is done against applications of DMM rules (recall from the last chapter that in the transition system representing a model's semantics, transitions are labeled with applications of DMM rules). To also be able to reason about arbitrary states of our model, we have therefore introduced a new type of DMM rule called *property rule*. See Sect. 6.2.2.8 for more information.

6.1.2 Pragmatic Changes

The formal nature of the DMM definition as provided by Hausmann makes it rather difficult to develop tool support. This is not only because the currently available frameworks for creating (visual) languages do not support such formalisms; moreover, the resulting tool support must exactly reflect the formal definitions, and this very fact must be proven by the language engineer.[1] Since our focus is the practical applicability of DMM, we have decided to take another approach for language definition:

- Instead of implicitly defining the DMM language's syntax by means of set theory, we have chosen a *metamodeling* approach (see Sect. 2.1 on page 9). The metamodel defining the DMM language's syntax is presented in the next section. OCL constraints defining the language's static semantics are provided in the appendix.
- Consequently, we have provided a *compiler semantics* for DMM, i.e., we have defined DMM's semantics by defining a transformation of DMM rulesets into GROOVE grammars (the semantics of which is well-defined). This transformation is described in Sect. 6.3.

To further enhance the practical applicability of DMM, we have optimized our transformation: While translating an Ecore model into a GROOVE state

[1] Recall that Hausmann claims that the formal definition of DMM does allow to formally reason about DMM specifications. Such proofs would loose all reliability in practice if the tool support behaves only slightly different to the formal definition of DMM.

graph, we filter out model elements which will not influence the computation of the transition system. The rational behind this filtering step is described in Sect. 6.3.2.1.

Another, simpler change we have made is the introduction of a `Package` element, which does not have any semantics, but still allows to structure DMM specifications (which can easily consist of hundreds of DMM rules).

For the sake of improving readability, we will from now on refer to DMM++ as DMM unless explicitly stated otherwise.

6.2 Syntax

In this section, we will introduce the newly defined DMM language (formerly referenced as DMM++). For each language element, we will show its role in the DMM language's metamodel and give a brief description of the element's properties.

The metaclasses are presented basically following the style of the UML specification [158] (which is expected to be known by many readers of this thesis). In particular, this means that each metaclass is accompanied with a textual description consisting of

- a brief description of the metaclass's purpose,
- the super metaclasses,
- a more detailed description of the metaclass and its usage,
- its attributes,
- its associations,
- an intuition of the metaclass's semantics (we will see more details in Sect. 6.3 on page 87, and
- the notation, i.e., the concrete syntax of instances of the metaclass.

Since the DMM metamodel is rather complex, the class diagram representation is spread out accross different views. In Sect. 6.2.1 we show the general structure of DMM rulesets, which contains of rules organized by packages. Section 6.2.2 shows the hierarchy of the different types of DMM rules and rule overriding relations, followed by a description of the internal rule structure of nodes, edges, and rule invocations in Sect. 6.2.3. Finally, in Sect. 6.2.4 we introduce the DMM expression language which is used to describe conditions over and assignments to attributes.

To improve readability of the DMM language definition, we are differing from the UML specification's style of representation in the following ways:

- The metaclasses are neither organized in packages (since the DMM metamodel consists of a single package) nor alphabetically. The metaclasses are organized by means of the four views mentioned above, and within each of the according sections, the classes are ordered by means of their relations (e.g., metaclass `Rule` is explained before metaclass `BigstepRule` which is a refinement of `Rule`).

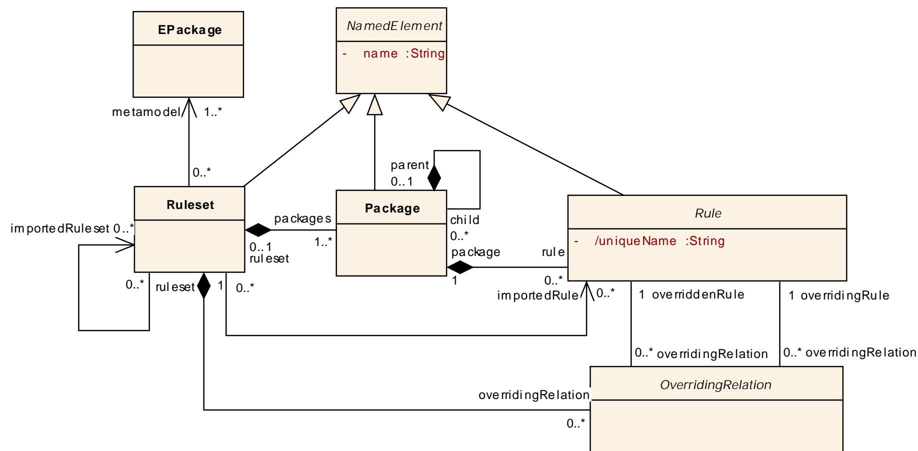

Figure 6.1: Ruleset view of the DMM metamodel.

- In case a metaclass does not have any (additional) attributes, the UML specification explicitly states this fact. In contrast, we have left out the according paragraph in such cases. The same holds for all other parts of the metaclasses' descriptions.
- In the UML specification, the static semantics is presented as part of the metaclass descriptions by means of OCL constraints. We have defined the DMM language's static semantics in the same way; however, we have moved the actual OCL constraints into the thesis' appendix.
- Some metaclasses appear in more than one class diagram. In these cases, the section of the metaclass's main view will contain the metaclass's textual description, and the other sections will contain references to that description.

6.2.1 Ruleset Structure

The view of the DMM metamodel described in this section contains the most high-level structure of DMM rulesets: A `Ruleset` consists of `Packages` which themselves can contain other `Packages` or DMM `Rules`. Additionally, a `Ruleset` contains zero or more `OverridingRelations` – these relations are stored within the `Ruleset` itself since they might be associated to `Rules` *accross* `Packages`. An overview of this high-level structure is depicted as Fig. 6.1.

6.2.1.1 NamedElement

An element with a name.

Description

`NamedElement` factors out the `name` attribute which is used by some DMM metaclasses.

Attributes

- name: String [1]
  The name of the element

6.2.1.2 Ruleset

A `Ruleset` instance represents a complete DMM ruleset.

Generalizations

- `NamedElement` on page 60

Description

A `Ruleset` is a collection of DMM rules which are typed over the ruleset's associated metamodels. It usually describes the semantics of a particular modeling language; other usages include the specification of *property rules* (see `PropertyRule` on page 68) or the extension of an already existing ruleset.

A ruleset's name should be distinct across different rulesets for better understandability.

Associations

- metamodel: `EPackage` [1..*]
  The metamodels over which the ruleset's rules (i.e., the rules' nodes and edges) can be typed
- package: `Package` [1..*]
  The packages (transitively) containing the ruleset's rules
- overridingRelation: `OverridingRelation` [0..*]
  A set of `OverridingRelations` which model the overriding of rules by other rules
- importedRuleset: `Ruleset` [0..*]
  The rulesets which are imported by this ruleset – see semantics below for use cases
- importedRule: `Rule` [0..*]
  The rules which are imported by this ruleset – see semantics below for use cases

Semantics

A (DMM) ruleset is a collection of rules which are designed for the sake of manipulating models in a certain way. Basically, there are no restrictions on how DMM can be used: For instance, in Section 7.2 we are using DMM rulesets to transform syntax models into runtime models. However, the most common use case for DMM is specifying the behavioral semantics of a certain language which is syntactically defined by means of a metamodel.

In this latter scenario, there are basically three contexts in which DMM rulesets can be used:

- As stand-alone semantics specification
  Here, the rules contained in the ruleset describe the complete semantics of a particular language. As such, the ruleset and a model of that language can be transformed into a GROOVE grammar and executed.
- As an extension of an already existing ruleset
  In this case, the ruleset is not executable as-is; instead, it imports another ruleset, for which it defines extensions. We will see an example for this in Section 6.3.5 on page 105.

- For defining properties
  To verify certain properties against a model, so-called *property rules* need to be defined (more on this in Section 9.1.4 on page 193). These rules can be stored within their own ruleset (which usually imports the ruleset describing the actual semantics).

Technically, a ruleset's semantics is defined by transforming it into an (executable) GROOVE graph grammar; the transformation is described in Sect. 6.3.

Notation
A ruleset does not have a dedicated notation, but is the sum of its packages (see below).

6.2.1.3 Package

`Packages` within a DMM ruleset can be used to structure the rules of that ruleset.

Generalizations

- `NamedElement` on page 60

Description
A `Package` is a collection of DMM rules which are typed over the ruleset's associated metamodels. Packages are only used to structure rules (and can therefore contain other packages), but they do *not* have any semantics, e.g. by defining some kind of namespace.

For better understandability, the name of a package should be distinct to the names of all packages at the same level (i.e., all packages directly contained in the ruleset or the same package should have distinct names).

Associations

- ruleset: `Ruleset` [0..1]
  The ruleset containing this package (if any)

- parent: `Package` [0..1]
  The package containing this package (if any)

- child: `Package` [0..*]
  The packages contained in this package

- rule: `Rule` [0..*]
  The rules contained in this package

Notation
A package is visualized as a rectangle containing nodes (representing rules) and other packages. The package's name is depicted in the left upper corner of a package. An example package is shown as Figure 6.2: The package's name is "FundamentalActivities", and it contains five rules.

6.2.1.4 Rule

See Section 6.2.2.2 on page 63.

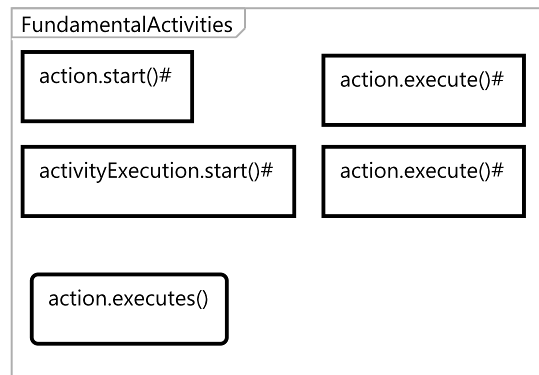

Figure 6.2: Example package with name "FundamentalActivities" and five contained rules.

6.2.1.5 OverridingRelation

See Section 6.2.2.11 on page 69.

6.2.2 Rule Hierarchy

The view of the DMM metamodel described in this section shows the hierarchy of the different types of DMM rules and overriding relations. An overview of this hierarchy is depicted as Fig. 6.3.

6.2.2.1 NamedElement

See Section 6.2.1.1 on page 60.

6.2.2.2 Rule

A `Rule` describes certain structures contained in a model, which can (optionally) be manipulated.

Generalizations

- `NamedElement` on page 60

Description

DMM `Rules` contain the actual description of a language's behavior. They describe operationally how to modify a model, and under which circumstances to do that modification. Technically, DMM rules are *graph transformation rules*.

There are four kinds of DMM rules, which we briefly describe below:

- Bigstep rules (see `BigstepRule` on page 66)
  Bigstep rules are applied as soon as they *match*, i.e., if the described object structure is found in the underlying model.
- Smallstep rules (see `SmallstepRule` on page 67)
  Smallstep rules need to be explicitly invoked by bigstep or smallstep rules to be applied.

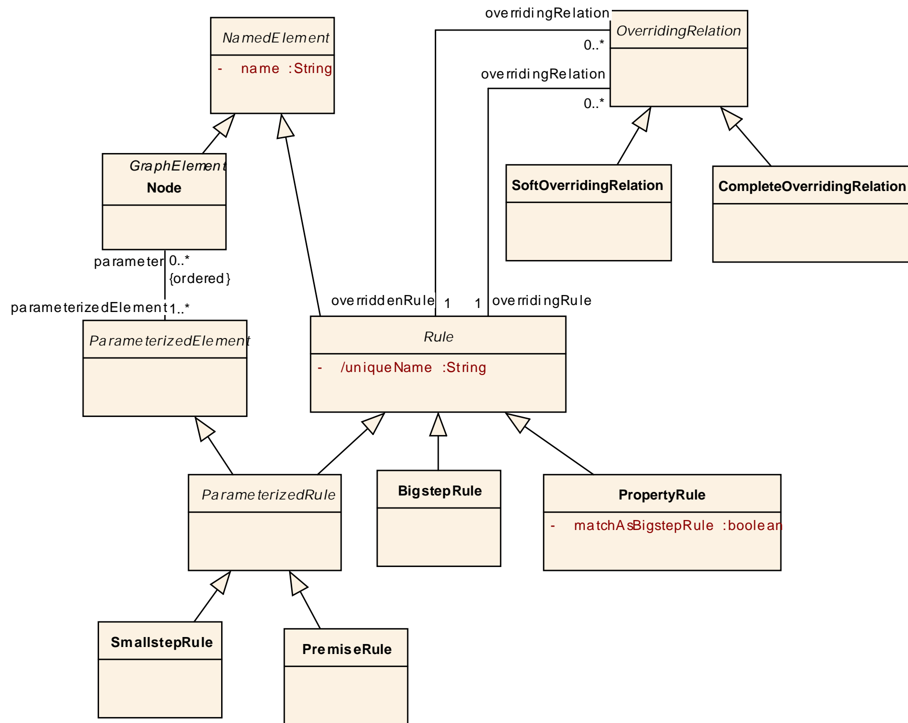

Figure 6.3: Rule hierarchy view of the DMM metamodel.

- Premise rules (see `PremiseRule` on page 67)
  Premise rules can be used to "factor out" object structures common to several other rules; they are merged into the invoking rules at execution time. Premise rules do not manipulate the underlying models.

- Property rules (see `PropertyRule` on page 68)
  Property rules are only used in the context of model checking a model's behavior. They do not modify the underlying model; instead, their matching signals that the described object structure is found in the according model state.

Note that each rule contains a dedicated node representing the *context* in which a rule is applied – the type of this node can be seen to *own* the rule's behavior (and is thus comparable to the `this` object in Java code).

Note also that DMM rule names in general do *not* have to be distinct – in contrast, it is often desirable to have rules with the same name in one ruleset. This is because within a DMM rule we can not model alternative behavior; however, we can achieve the same effect by providing rules of the same name which each handle one of the different cases.

Attributes

- /uniqueName: String [1]
  The unique name of the rule within the rule's ruleset.

Associations

- package: `Package` [1]
  The package containing this rule
- overridingRelation: `OverridingRelation` [0..*]
  The overriding relations in which this rule is overridden
- overriddenRelation: `OverridingRelation` [0..*]
  The overriding relations in which this rule is overriding
- node: `Node` [0..*]
  The nodes of this rule
- edge: `Edge` [0..*]
  The edges of this rule
- invocation: `Invocation` [0..*]
  The invocations of this rule
- emphasizedAttribute: `EmphasizedNodeAttribute` [0..*]
  The emphasized attributes of this rule
- contextnode: `Node` [1]
  The context node of this rule

Semantics
A (DMM) rule is the smallest unit of behavior specification provided by DMM. It contains a number of nodes and edges, the former corresponding to objects, the latter to references between those objects. A rule *matches* an object structure if an occurence of the rule's own structure of objects and references can be found in that object structure. If this is the case, the rule is *applied*, i.e., the changes described by the rule are performed on the found occurrence (e.g., objects and/or edges can be destroyed or created, attributes can be manipulated etc.).

Notation
The graphical notation of DMM rules is shown at the concrete subclasses of `Rule`. Additionally, a DMM rule also has a textual representation called *signature*. This is build as follows: the signature starts with the name of the rule's context node, followed by a dot "." and the name of the rule. A rule's signature ends with two brackets "()". Some concrete subclasses slightly differ in the definition of their signature – these differences will be pointed out where they occur.

6.2.2.3 ParameterizedElement

A `ParameterizedElement` is an element which has nodes as parameters.

Description
DMM `ParameterizedRules` as well as `Invocations` have an ordered set of nodes acting as parameters. This class factors out the `parameter` association.

Associations

- parameter: `Node` [0..*]
  The ordered set of nodes acting as parameters for this element

Figure 6.4: Bigstep rule with name action.start()#.

6.2.2.4 ParameterizedRule

A `ParameterizedRule` is a rule which additionally has parameters.

Generalizations

- `Rule` on page 63
- `ParameterizedElement` on page 65

Description
DMM `ParameterizedRules` are rules which additionally inherit from `ParameterizedElement`, through which they inherit a set of nodes acting as parameters.

Notation
In addition to a rule signature as defined in Sect. 6.2.2.2, the signature of a parameterized rule also contains the names and types of the parameter nodes, separated by commas ",". The parameters are placed within the rule signature's brackets.

6.2.2.5 BigstepRule

A `BigstepRule` is a rule which can match without being explicitly invoked.

Generalizations

- `Rule` on page 63

Description
DMM bigstep rules are usually used to model some self-contained behavior – if a bigstep rule is completely processed, then that behavior is completed. If the behavior to be described is too complex for one rule, a bigstep rule can invoke smallstep rules, which will be processed before the next bigstep rule can match and be applied.

Semantics
DMM bigstep rules are the rules which are closest to normal graph transformation rules in the sense that they are applied as soon as they match (and there is no smallstep rule which had been invoked and has not yet been applied). How this behavior is realized will be shown in Sect. 6.3.

Notation
A bigstep rule is depicted as a rectangle with solid borders. In addition to a rule signature as defined in Sect. 6.2.2.2, a bigstep rule's signature always ends with a hash sign "#". An example bigstep rule is depicted as Fig. 6.4.

Figure 6.5: Smallstep rule with name action.executes().

6.2.2.6 SmallstepRule

A `SmallstepRule` is a rule which must be explicitly invoked by a bigstep rule or another smallstep rule to be applied.

Generalizations

- `Rule` on page 63
- `ParameterizedRule` on page 66

Description

Some behavior is too complex to be described by a single DMM rule. In this case, smallstep rules can be used to divide that behavior into smaller steps, which are then performed one by one. This gives the language engineer more control over rule application – the alternative would be to carefully design bigstep rules such that they can only match in a certain order, which is significantly more difficult.

However, some care still has to be taken when designing smallstep rules: A smallstep rule can only be applied if it is invoked *and* matches the current state at that very point in time. Thus, a DMM specification giving rise to situations where a smallstep rule is invoked but no smallstep rule matches the current state is erronous.

In contrast to bigstep rules, smallstep rules can have parameter nodes.

Semantics

DMM smallstep rules can only be applied if they are invoked *and* match the current state. This is realized by maintaining an *invocation stack* within the state graph, and by adding structures to the GROOVE rule resulting from a smallstep rule which make sure that the smallstep rule can only match if an according invocation is on top of the stack. The details of this will be shown in Sect. 6.3.

Notation

A smallstep rule is depicted as a rounded rectangle with solid borders. Additionally, a smallstep rule's signature can be distinguished from that of a bigstep rule because it does *not* end with a hash sign #. An example smallstep rule is depicted as Fig. 6.5.

6.2.2.7 PremiseRule

A `PremiseRule` is a rule which describes some premises a state must fulfill for a smallstep or bigstep rule to match.

Generalizations

- `Rule` on page 63
- `ParameterizedRule` on page 66

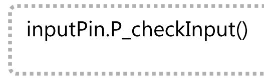

Figure 6.6: Premise rule with name inputPin.P_checkInput().

Description
In some cases, a certain structure must exist within several rules for these rules to match. In this case, that structure can be factored out into a single premise rule, and the according rules can invoke that premise rule. As such, a premise rule can be used to improve the understandability and maintainability of a DMM ruleset.

As smallstep rules, premise rules can have parameter nodes.

Semantics
DMM premise rules are not independent rules, but are merged into the invoking rules – they act as an enhancement of the left-hand graphs of the invoking rules. The details of this will be shown in Sect. 6.3.

Notation
A premise rule is depicted as a rounded rectangle with dotted borders. In addition to a rule signature as defined in Sect. 6.2.2.2, in the case of premise rules the rule's name is prefixed with "P_". An example premise rule is depicted as Fig. 6.6.

6.2.2.8 PropertyRule

A `PropertyRule` is a rule which describes properties of a state.

Generalizations

- `Rule` on page 63

Description
DMM property rules describe a certain object structure; if a property rule matches a state, then the rule's structure must be contained in that state. As such, property rules do not belong to the semantics of a language (they do not change the state). Instead, they can be used to describe object structures the language user is interested in: Matching of a property rule will result in a self-transition of the according state, and that information can be used for model checking. We will see examples for this in Sect. 9.1.

Attributes

- matchAsBigstepRule: `Boolean` [1]
  Whether the property rule shall only match if no pending invocations of smallstep rules exist (i.e., in states where bigstep rules can also match)

Semantics
DMM property rules are rules which have the same left-hand and right-hand graph. The details of translating property rules into GROOVE rules will be shown in Sect. 6.3.

Notation
A property rule is depicted as a rectangle with dashed borders. In addition to

Figure 6.7: Property rule with name my.propertyRule()!.

a rule signature as defined in Sect. 6.2.2.2, a property rule's signature always ends with an exclamation mark "!". An example property rule is depicted as Fig. 6.7.

6.2.2.9 Node

See Section 6.2.3.5 on page 73.

6.2.2.10 GraphElement

See Section 6.2.3.4 on page 73.

6.2.2.11 OverridingRelation

An `OverridingRelation` models that one DMM rule overrides another DMM rule.

Description
Rule overriding allows to refine DMM rules by introducing the ability for rules to override other rules. A more thorough description is provided at the concrete subclasses `CompleteOverriding` (see Sect. 6.2.2.13 on page 70) and `Soft-Overriding` (see Sect. 6.2.2.12 on page 69).

Associations

- overriddenRule: `Rule` [1]
  The rule overridden through this overriding relation

- overridingRule: `Rule` [1]
  The rule overriding through this overriding relation

6.2.2.12 SoftOverridingRelation

A `SoftOverridingRelation` models that one DMM rule *softly overrides* another DMM rule.

Generalizations

- `OverridingRelation` on page 69

Description
Soft overriding models a dynamic kind of rule overriding: If a rule r softly overrides another rule r', and if r does not match the current state, the execution will be delegated to r'.

Semantics
If a rule r softly overrides rules $r_0, \ldots, r_n$, it will first be checked whether r can be applied. If this is not the case, rules $r_0, \ldots, r_n$ will be checked for applicability.

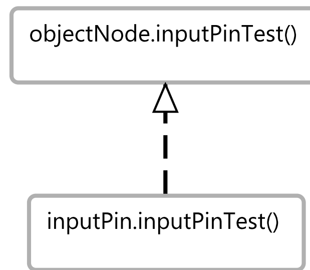

Figure 6.8: Rule inputPin.inputPinTest() softly overrides rule objectNode.inputPinTest().

This process is repeated until an applicable rule is found, which will then be applied. If no applicable rule can be found at all, a failure state is reached. DMM specifications allowing for such states are considered to be erronous.

In a nutshell, the semantics of soft overriding relations is implemented by means of some helper rules which will—one level after the other—activate small-step rules until a matching one is found. In Sect. 6.3, we will see this process in detail. Since application of the helper rules results in transitions within the final transition system, we decided to not let bigstep rules participate in those transitions. Otherwise, we would have to "try out" all levels of the bigstep rule overriding relations whenever the invocation stack is empty, cluttering the transition system to a huge extend.

Notation
Soft rule overriding is depicted with a dashed arrow as known from UML inheritance. See Fig. 6.8 for an example.

6.2.2.13 CompleteOverridingRelation

A `CompleteOverridingRelation` models that one DMM rule *completely overrides* another DMM rule.

Generalizations

- `OverridingRelation` on page 69

Description
Complete rule overriding can be used to avoid that another rule matches at all in the context of the overriding rule's type.

Semantics
If a rule r completely overrides rules $r_0, \ldots, r_n$, then $r_0, \ldots, r_n$ can never match in the context of a node whose type is the same or a subtype of r's context node's type.

Notation
Soft rule overriding is depicted with a solid arrow as known from UML inheritance. See Fig. 6.9 for an example.

behaviorExecution.destroy()

actionExecution.destroy()

actionExecution.destroy()

Figure 6.9: Two versions of rule actionExecution.destroy() which both completely override rule behaviorExecution.destroy().

Figure 6.10: Internal rule structure view of DMM metamodel.

6.2.3 Internal Rule Structure

This section presents the internal structure of DMM rules. A `Rule` basically consists of `Nodes` (corresponding to objects), `Edges` (corresponding to links between objects), and `Invocations` (which model the operational part of DMM rulesets by explicitly invoking other DMM rules). `Nodes` and `Edges` both extend the metaclass `GraphElement` from which they inherit an `ElementRole` (modeling the action to perform for the element, e.g., deletion or creation) and a `Quantification` (which describes the number of objects to which an element is matched). In contrast to `Edges`, `Nodes` can additionally have `Conditions` which must be true for a `Node` to be matched, `Assignments` which are used to manipulate attribute values of the object a `Node` is mapped to, and `EmphasizedNodeAttributes` which are used to make values of the object's attributes visible within a transition system. An overview of the internal structure of DMM rules is depicted as Fig. 6.10.

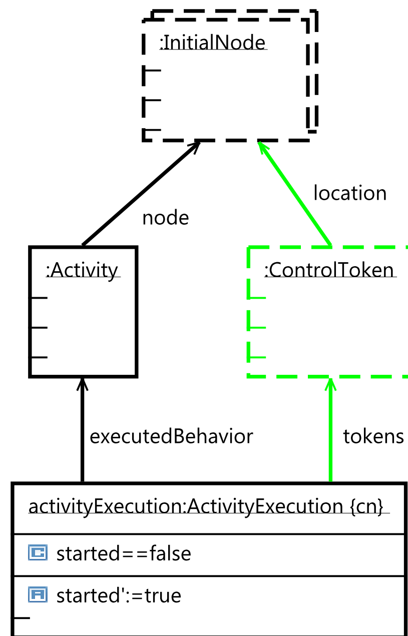

Figure 6.11: Internal structure of rule activityExecution.start()#.

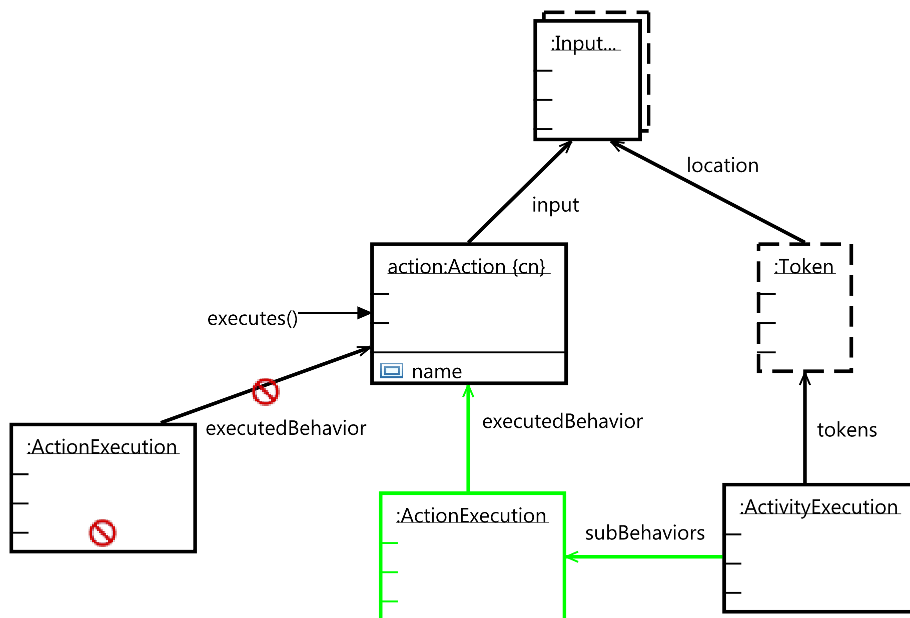

Figure 6.12: Internal structure of rule action.start()#.

6.2.3.1 NamedElement

See Section 6.2.1.1 on page 60.

6.2.3.2 Rule

See Section 6.2.2.2 on page 63.

6.2.3.3 ParameterizedElement

See Section 6.2.2.3 on page 65.

6.2.3.4 GraphElement

A `GraphElement` is an element having an `ElementRole` as well as a `Quantifier`.

Description
`Nodes` as well as `Edges` have a `role` and a `quantification`. This class factors out these associations.

Associations

- role: `ElementRole` [1]
  The role of this graph element

- quantification: `Quantifier` [1]
  The quantification of this graph element

6.2.3.5 Node

A `Node` is a graph node within a DMM rule and corresponds to an object.

Generalizations

- `NamedElement` on page 60

- `GraphElement` on page 73

Description
A `Node` represents an object within the object structure described by the rule the node is contained in. It has a role which describes whether the node belongs to the rule's application context, is to be created, to be deleted, or belongs to a negative application conditon of the rule (see Sect. 6.2.3.8). In addition, a node has a quantification determining whether that node e.g. belongs to a universal quantified structure (see Sect. 6.2.3.9).

Nodes may have a name which can be used to refer to them from the same rule within conditions and assignments (see below).

A node may contain `Conditions` (see Sect. 6.2.3.10) which have to be fulfilled for the state object – otherwise, the node can not be mapped to that object. Additionally, a node may contain `Assignments` (see Sect. 6.2.3.11) which can be used to manipulate the state object's attribute values. Finally, a node can be associated with `EmphasizedNodeAttributes` (see Sect. 6.2.3.12); if this is the case, the value of the according attributes of the state object the node is

mapped to will be displayed in the transition system (and can e.g. be used for model checking purposes).

Associations

- rule: `Rule` [1]
  The rule owning this node
- type: `ecore::EClass` [1]
  The type of this node
- incoming: `Edge` [0..*]
  The edges having this node as the target node
- outgoing: `Edge` [0..*]
  The edges having this node as the source node
- invocation: `Invocation` [0..*]
  The invocations having this node as the target node
- parameterizedElement: `ParameterizedElement` [0..*]
  The parameterized elements (i.e., invocations or parameterized rules) having this node as at least one of its parameters
- condition: `Condition` [0..*]
  The conditions of this node
- assignment: `Assignment` [0..*]
  The assignments of this node
- emphasizedAttribute: `EmphasizedNodeAttribute` [0..*]
  The attributes of this node which are shown at the transitions resulting from applications of this node's rule

Semantics

If a rule matches a state, each node of the left-hand graph of that rule is mapped to one or more nodes of the state. The state nodes will then (optionally) be modified, i.e., they will be deleted or created, or their attribute values will be modified. The details of this are given in Sect. 6.3.

Notation

A node is basically depicted in object notation, i.e., as a rectangle carrying the node's type (and name, if any) at the rectangle's top. In DMM, a node rectangle has three additional (and potentially empty) compartments for the node's conditions, assignments (see node `activityExecution:ActivityExecution` in Fig. 6.11 on page 72), and emphasized attributes (see node `action:Action` in Fig. 6.12 on page 72). The colors of nodes are explained in Sect. 6.2.3.8, the different node shapes in Sect. 6.2.3.9.

6.2.3.6 Edge

An `Edge` is a graph edge within a DMM rule and corresponds to a link (i.e., an instance of an association) between two objects.

Generalizations

- `GraphElement` on page 73

Description
An `Edge` represents a link within the object structure described by the rule the edge is contained in. It has a role which describes whether the edge belongs to the rule's application context, is to be created, to be deleted, or belongs to a negative application conditon of the rule.

Associations

- rule: `Rule` [1]
  The rule owning this edge

- reference: `ecore::EReference` [1]
  The association this edge is typed over

- source: `Node` [1]
  The source node of this edge

- target: `Node` [1]
  The target node of this edge

Semantics
If a rule matches a state, each edge of the left-hand graph of that rule is mapped to one or more edges of the state. The state edges will then (optionally) be modified, i.e., they will be deleted or created. The details of this are given in Sect. 6.3.

Notation
An edge is basically depicted in UML notation, i.e., as an arrow carrying the edge's name (which is the name of the reference the edge is typed over). See e.g. Fig. 6.11 on page 72 for examples. The colors of edges are explained in Sect. 6.2.3.8, the different edge shapes in Sect. 6.2.3.9.

6.2.3.7 Invocation

An `Invocation` is used to invoke either a `SmallstepRule` (see Sect. 6.2.2.6) or a `PremiseRule` (see Sect. 6.2.2.7).

Generalizations

- `ParameterizedElement` on page 65

Description
`Invocations` represents the operational part of DMM semantics specifications. They are used to explicitly invoke smallstep or premise rules – the former being only able to match when explicitly being invoked, the latter being merged into the invoking rule.

An invocation is perfomed on a target node, which will act as the context node in the invoked rule. If a rule contains more than one invocation, the order of execution of these invocations is determined by the sequence number of the invocations.

Attributes

- sequenceNumber: `Integer` [1]
  The sequence number determines the order of execution of invocations within a rule
- invokedRule: `String` [1]
  The name of the invoked rule

Associations

- rule: `Rule` [1]
  The rule owning this invocation
- targetNode: `Node` [1]
  The target node of this invocation

Semantics
The semantics of invocations is two-fold: invocations of premise rules result in those rules being merged into the invoking rules at ruleset translation time, i.e., the resulting GROOVE grammar will not contain premise rules at all, but their structures are contained in the invoking rules. In contrast, invocations of smallstep rules result in these rules being only able to match if they are indeed invoked. Technically, the latter is achieved by maintaining an *invocation stack* within the state graph, and by adding structures to the smallstep rules such that they can only match if they are on top of that stack (in which case application will remove them from the stack). The details of this are given in Section 6.3.

Notation
An invocation is depicted as an arrow, the arrowhead of which pointing to the invocation's target node. On the other end of the arrow, the name of the invoked rule is printed, with the names of the parameter nodes within brackets. The sequence number is found at the beginning of the invocation's label, and is separated from the invoked rule's name by a colon ":". If no sequence number is provided, the sequence number is implicitly set to 0. See Fig. 6.12 on page 72 for an example invocation.

6.2.3.8 ElementRole

An `ElementRole` models whether a graph element belongs to the left-hand graph, the right-hand graph, both, or a negative application of a rule.

Description
An `ElementRole` represents the different roles a graph element can take within a rule. It has four literal values:

- `EXISTS`: The graph element needs to exist for the rule to match and will not be deleted (application context).
- `DESTROY`: The graph element needs to exist for the rule to match and be will deleted during the rule's execution.
- `CREATE`: The graph element will be created during the rule's execution.
- `NOT_EXISTS`: The graph element must not exist for the rule to match (negative application condition).

Semantics
Graph elements with role EXISTS exist in both the left-hand and the right-hand graph of the graph transformation rule. Graph elements with role DESTROY (CREATE) only exist in the left-hand (right-hand) graph of the rule. Graph elements with role NOT_EXISTS belong to a negative application condition of the rule. The details of this are given in Section 6.3.

Notation
The role of a graph element is depicted by the element's color: elements with role EXISTS are depicted in black, elements with role DESTROY are depicted in red, and elements with role CREATE are depicted in green. The only exception are elements with role NOT_EXISTS – these are annotated with a stop sign.[2]

6.2.3.9 Quantifier

A Quantifier models how many elements will be considered during matching and application of a rule.

Description
An ElementRole represents cardinalities of a graph element within a rule. It has four literal values:

- ONE: The graph element will be matched to exactly one element.
- ZERO_TO_MANY: The graph element will be matched to at least zero, but as many elements as possible.
- ONE_TO_MANY: The graph element will be matched to at least one, but as many elements as possible.
- NESTED: The graph element will be matched in conjunction with an element quantified ZERO_TO_MANY or ONE_TO_MANY. It can be used to describe situations like "For all initial nodes, create one token" (where the initial node is quantified ZERO_TO_MANY or ONE_TO_MANY, and the token is quantified NESTED).

Semantics
Quantifications of a graph element are directly mapped to their GROOVE counterparts. The details of this are given in Sect. 6.3.

Notation
A node with quantification ONE is depicted with solid borders. A node with quantification ZERO_TO_MANY or ONE_TO_MANY is depicted in multi-object notation, where the front rectangle has solid borders in the case of ONE_TO_MANY and dashed borders in the case of ZERO_TO_MANY – the back rectangle always has dashed borders. Finally, a node with quantification NESTED is depicted with dashed borders.

Edges with quantification ONE and NESTED are depicted with a solid arrow.[3] Edges with quantification ZERO_TO_MANY and ONE_TO_MANY are depicted with a dashed arrow, the latter starting with a solid part.

[2] Note that the DMM tooling we developed also supports the concrete syntax suggested by Hausmann [96], where roles are modeled by the annotations {create} and {destroy}, and the color scheme used by GROOVE [166] where elements to be deleted are blue, elements to be created are green, and elements belonging to negative application conditions are red.

[3] We do not distinguish visually between these quantifications of an edge since an edge's

6.2.3.10 Condition

A `Condition` of a node describes a predicate over attributes of that node.

Description
A `Condition` is a boolean expression over attributes of a node. A rule node can only be mapped to a state node if the state node's attribute values fulfill the modeled condition.

Associations

- node: `Node` [1]
  The node owning this condition
- expression: `Expression` [1]
  The textual expression describing this condition

Semantics
A condition is described by a textual expression. Within that expression, attribute values of the node owning the condition as well as of other nodes (referenced by the nodes' names) can be used, in conjunction with several operations as described in Sect. 6.2.4. The details of mapping expressions to GROOVE constructs are given in Sect. 6.3.

Notation
The textual expressions of conditions are contained in the upper compartment of a node.

6.2.3.11 Assignment

An `Assignment` of a node allows to assign a newly computed value to an attribute of a node.

Description
An `Assignment` is an expression over attributes of nodes. It will be evaluated at runtime, and the result of the evaluation will be assigned to the according attribute of the owning node.

Associations

- node: `Node` [1]
  The node owning this assignment
- expression: `Expression` [1]
  The textual expression describing this assignment
- assignTo: `AttributeExpression` [1]
  The attribute expression representing the attribute the new value will be assigned to

Semantics
An assignment is described by a textual expression. Within that expression, attribute values of the node owning the assignment as well as of other nodes

quantification is obvious from the quantification of the nodes that edge has as source and target nodes – if at least one of the nodes is quantified `NESTED`, then the edge also has that quantification.

(referenced by the nodes' names) can be used, in conjunction with several operations as described in Sect. 6.2.4. The details of mapping expressions to GROOVE constructs are given in Sect. 6.3.

Notation
The textual expressions of assignments are contained in the middle compartment of a node.

6.2.3.12 EmphasizedNodeAttribute

An `EmphasizedNodeAttribute` of a node makes sure that the value of the according attribute is shown within the transition label when the rule is applied.

Description
An `EmphasizedNodeAttribute` is used to display an attribute value if the according rule is applied. The transition labels carry those attribute values, which can then e.g. be used in model checking.

Associations

- rule: `Rule` [1]
  The rule owning this emphasized attribute

- node: `Node` [1]
  The node of which an attribute value shall be shown

- attribute: `ecore::EAttribute` [1]
  The attribute whose value shall be shown

Semantics
If a rule matches a state, the rule's emphasized attributes are bound to the values of the state nodes the emphasized attributes' nodes are mapped to. The details of this are given in Section 6.3.

Notation
The names of the emphasized attributes of a node are contained in the lowest compartment of that node.

6.2.4 DMM Expression Language

DMM makes use of an internal, textual expression language used for the formulation of assignments and conditions. This section will introduce that language. The language's concrete syntax is defined by means of a grammar which is presented in the next section. From Sect. 6.2.4.2 on, the metamodel elements of the DMM expression language are introduced, following the same scheme as in the rest of Sect. 6.2. The expression part of the DMM metamodel is depicted as Fig. 6.13.

The DMM expression language has been developed by Eduard Bauer within his bachelor thesis [16] – we have added basic support for enumerations. We only provide the language definition here; for details on how the expression language was implemented, please refer to Bauer's thesis.

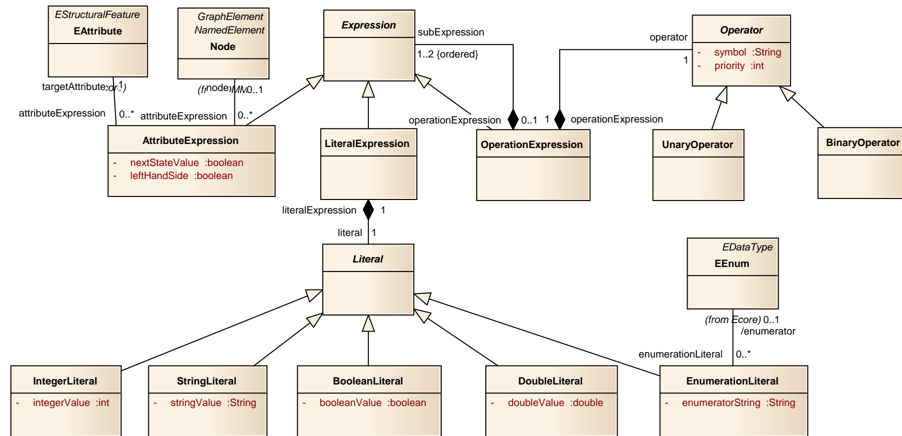

Figure 6.13: Metamodel of the DMM Expression language.

Table 6.1: Lexical Tokens of the DMM Expression language (after [16, p. 20])

<table>
<tr><th>Name of regular expression</th><th>Regular expression</th></tr>
<tr><td>boolean_literal</td><td>(true|false)</td></tr>
<tr><td>integer_literal</td><td>(0-9)+</td></tr>
<tr><td>double_literal</td><td>(0-9)+.(0-9)*(exp)?|.(0-9)+(exp)?|
(0-9)+exp</td></tr>
<tr><td>enum_literal</td><td>'(subident)'</td></tr>
<tr><td>exp</td><td>(e|E)(+|-)?(0-9)+</td></tr>
<tr><td>string_literal</td><td>"(~("|\)|\(\|"))*"</td></tr>
<tr><td>identifier</td><td>(subident.)?subident(')?</td></tr>
<tr><td>subident</td><td>(_|a-z|A-Z)(_|a-z|A-Z|0-9)*</td></tr>
<tr><td>set of operators[4]</td><td>+, -, *, /, %, !, &, |, min, max,
abs, <, <=, ==, >=, >, :=</td></tr>
</table>

6.2.4.1 Grammar of Expression Language

The syntax of expressions to be used within DMM assignments and conditions is defined by means of a grammar, which is represented in this section. We start by defining the lexical tokens used within expressions, followed by the productions from which DMM expressions can be built.

Lexical Tokens DMM expressions make use of a couple of datatypes and operators which are evaluated on these datatypes. Additionally, DMM expressions allow to refer to attribute values – for the latter, the concept of an identifier is needed.

The lexical tokens representing those datatypes are shown in Table 6.1. Within the regular expressions, two simplifications are used to present them more compactly: First, regular expressions such as `0-9` are used as an abbreviation for `0|1|2|3|4|5|6|7|8|9`. Second, the regular expression `~(a|b)` describes all characters *except* `a` and `b`.

[4]The operators are not given as a regular expression, but just listed for reference; the

Productions Finally, the productions from which DMM expressions can be derived is provided as Table 6.2. Note that the priorities are achieved by "delegating" from one nonterminal to the next. For instance, the nonterminal `<orExp>` can either be derived to an expression containing the operator `|` with priority 1, or to the nonterminal `<andExp>` increasing the priority by 1.

Table 6.2: Grammar of the DMM Expression language (after [16, p. 25])

<table>
<tr><th>No.</th><th>Productions of grammar</th><th>Prio</th></tr>
<tr><td>(1)</td><td><assignment> ::= identifier ':=' <orExp></td><td>-</td></tr>
<tr><td>(2)</td><td><orExp>
::= <orExp> '|' <andExp>
::= <andExp></td><td>1</td></tr>
<tr><td>(3)</td><td><andExp>
::= <andExp> '&' <equalityExp>
::= <equalityExp> \\</td><td>2</td></tr>
<tr><td>(4)</td><td><equalityExp>
::= <equalityExp> '==' <inequalityExp>
::= <inequalityExp></td><td>3</td></tr>
<tr><td>(5)</td><td><inequalityExp>
::= <inequalityExp> ('>=' | '>' | '<=' |
'<' | '!=')
<additiveExp>
::= <additiveExp></td><td>4</td></tr>
<tr><td>(6)</td><td><additiveExp>
::= <additiveExp> ('+' | '-')
<multiplicativeExp>
::= <multiplicativeExp></td><td>5</td></tr>
<tr><td>(7)</td><td><multiplicativeExp>
::= <multiplicativeExp> ('%' | '*' | '/')
<unaryExp>
::= <unaryExp></td><td>6</td></tr>
<tr><td colspan="3" align="right">Continued on next page</td></tr>
</table>

literals are directly contained within the language's grammar (see Table 6.2)

Table 6.2 – continued from previous page

No.	Productions of grammar	Prio
(8)	<unaryExp> ::= <unaryExp> ('-' | '!') <mathPrefixExp> ::= <primaryExp> ::= <mathPrefixExp>	7
(9)	<mathPrefixExp> ::= ('max' | 'min') '(' <orExp ',' <orExp> ')' ::= 'abs' '(' <orExp> ')' \\	8
(10)	<primaryExp> ::= <literalExp> ::= identifier ::= '(' <orExp> ')'	-
(11)	<literalExp> ::= integer_literal ::= double_literal ::= boolean_literal ::= string_literal ::= enum_literal	-

6.2.4.2 Expression

An `Expression` is the superclass of the different kinds of expressions.

Description

An `Expression` is the most general kind of the expressions which can be used within DMM conditions and assignments, from which the concrete expression types `AttributeExpression` (see Sect. 6.2.4.3), `LiteralExpression` (see Sect. 6.2.4.4), and `OperationExpression` (see Sect. 6.2.4.5) inherit. Expressions are built recursively, making use of the composite pattern [78, p. 163].

Associations

- operationExpression: `OperationExpression` [0..1]
  The `OperationExpression` of which this expression is a sub expression

Notation

Expressions can be *unparsed* into a textual representation (and such textual representations can be *parsed* into an expression). The grammar of expressions is shown in Sect. 6.2.4.1.

6.2.4.3 AttributeExpression

An `AttributeExpression` represents an attribute.

Generalizations

- `Expression` on page 82

Description
An `AttributeExpression` represents an attribute of some node. It can occur in two contexts:

- Within an expression that forms a condition or the right side of an assignment. In this context, the attribute expression is at runtime bound to that attribute's value.

- On the left side of an assignment. In this context, the result of evaluating the right side of the assignment will be stored into the attribute.

Attributes

- nextStateValue: `Boolean` [1]
  Whether this attribute expression refers to the attribute's value at rule matching time or after rule application time.

- leftHandSide: `Boolean` [1]
  Wether this attribute is the left side of an assignment

Associations

- targetAttribute: `ecore::EAttribute` [1]
  The attribute this attribute expression refers to

- node: `Node` [1]
  The node of which the attribute's value will be bound to the attribute expression

Semantics
Attribute expressions may refer to the value of an attribute in the state the expression's rule is currently matching or in the next following state (i.e., the state we get from applying the rule) – in the latter case, the `nextStateValue` attribute will be true. Furthermore, an attribute can be used either at the left side of an assignment or within an expression, i.e., in either the right side of an assignment or in a condition. In case the `leftHandSide` attribute is true, the former is the case, the meaning being that the evaluation of the assignment's right side will be assigned to that attribute.

Notation
See Sect. 6.2.4.1.

6.2.4.4 LiteralExpression

A `LiteralExpression` represents a literal, i.e., a number, string, boolean, or enumeration literal.

Generalizations

- `Expression` on page 82

Description
A `LiteralExpression` represents actual literals to be used within expressions. The `LiteralExpression` therefore refers to a single `Literal` instance which has several subclasses referring to the type of literal to be used.

Associations

- literal: `Literal` [1]
  The literal of this literal expression

Semantics
A literal expression represents the value of its literal.

Notation
See Sect. 6.2.4.1.

6.2.4.5 OperationExpression

An `OperationExpression` represents an operation to be performed on its operands.

Generalizations

- `Expression` on page 82

Description
An `OperationExpression` represents an operation. Within the composite pattern of which DMM expressions are built, it represents the container: An operation expression can have one or two sub expressions, which can again be operation expressions (or literal expressions or attribute expressions). Additionally, an operation expression is associated with exactly one operator determining the operation to be performed.

Associations

- operator: `Operator` [1]
  The operator representing the operation to perform

Semantics
The semantics of an operation expression is the (recursive) evaluation of that expression according to the operator's semantics.

Notation
See Sect. 6.2.4.1.

6.2.4.6 Operator

An operator represents some unary or binary operation to be performed on the operands of the operation expression the operator belongs to.

Description
An operator stores the symbol representing the very operation represented by that operation expression as well as a priority which is used when unparsing expressions. Operators can be unary or binary.

Attributes

- symbol: `String` [1]
  The symbol of this operator
- priority: `Integer` [1]
  The priority of this operator (used for unparsing of expressions)

Associations

- operationExpression: `OperationExpression` [1]
  The operatition expression owning this operator

Semantics
An operator's semantics is naturally given by its symbol's semantics. The currently available operators, their signatures, notations, and priorities are listed in Table 6.3.

Notation
See Sect. 6.2.4.1.

6.2.4.7 UnaryOperator

See Sect. 6.2.4.6 on page 84.

6.2.4.8 BinaryOperator

See Sect. 6.2.4.6 on page 84.

6.2.4.9 Literal

A `Literal` represents some fixed mumerical, enumeration, or string value.

Description
A literal represents a number, string, or enumeration literal. The metaclass `Literal` itself is abstract – the according concrete subclasses are

- `BooleanLiteral`: represents a boolean value
- `IntegerLiteral`: represents an integer value
- `DoubleLiteral`: represents a floating point value (Java datatype `double`)
- `StringLiteral`: represents a string value, i.e., an array of characters
- `EnumerationLiteral`: represents one of the possible values of an enumeration

Each subclass has an accordingly typed attribute which stores the value the literal instance represents.

Attributes
None (but see description above).

Associations

- literalExpression: `LiteralExpression` [1]
  The literal expression owning this literal

Semantics
The semantics of a literal is the literal's value as described above.

Table 6.3: Operators of the DMM Expression language (from [16, p. 23])

Operation name	Operator	Signature	Notat.	Prio
Conjunction	&	bool × bool → bool	infix	2
Disjunction	|	bool × bool → bool	infix	1
Negation	!	bool → bool	prefix	7
Equality	==	bool × bool → bool	infix	3
Addition	+	int × int → int	infix	5
Subtraction	-	int × int → int	infix	5
Multiplication	*	int × int → int	infix	6
Division	/	int × int → int	infix	6
Modulo	%	int × int → int	infix	6
Minus	-	int → int	prefix	7
Minimum	min	int × int → int	prefix	8
Maximum	max	int × int → int	prefix	8
Absolute value	abs	int × int → int	prefix	8
Greater than	>	int × int → bool	infix	4
Greater or equal	>=	int × int → bool	infix	4
Equality	==	int × int → bool	infix	3
Less or equal	<=	int × int → bool	infix	4
Less than	<	int × int → bool	infix	4
Addition	+	double × double → double	infix	5
Subtraction	-	double × double → double	infix	5
Multiplication	*	double × double → double	infix	6
Division	/	double × double → double	infix	6
Minus	-	double → double	prefix	7
Minimum	min	double × double → double	prefix	8
Maximum	max	double × double → double	prefix	8
Absolute value	abs	double × double → double	prefix	8
Greater than	>	double × double → bool	infix	4
Greater or equal	>=	double × double → bool	infix	4
Equality	==	double × double → bool	infix	3
Less or equal	<=	double × double → bool	infix	4
Less than	<	double × double → bool	infix	4
Concatenation	+	string × string → string	infix	5
Greather than	>	string × string → bool	infix	4
Greater or equal	>=	string × string → bool	infix	4
Equal	==	string × string → bool	infix	3
Less or equal	<=	string × string → bool	infix	4
Less than	<	string × string → bool	infix	4

6.2.4.10 IntegerLiteral

See Sect. 6.2.4.9 on page 85.

6.2.4.11 StringLiteral

See Sect. 6.2.4.9 on page 85.

6.2.4.12 BooleanLiteral

See Sect. 6.2.4.9 on page 85.

6.2.4.13 DoubleLiteral

See Sect. 6.2.4.9 on page 85.

6.2.4.14 EnumerationLiteral

See Sect. 6.2.4.9 on page 85.

6.3 Semantics

In Sect. 5.2, we have seen that DMM as defined by Hausmann in [96] was self-contained in the sense that Hausmann has provided a complete, set theory based formalization, including the definition of rule matching and rule application. This is the most precise way to specify a language's semantics, but has the drawback that it does not translate well into tool support. For this reason, Hausmann has only briefly sketched a translation of DMM rulesets into GROOVE grammars which can then be executed using the GROOVE tooling.

In contrast, the semantics of DMM++ is defined differently: Instead of providing an own formalization of the semantics of DMM, we focus on the transformation into GROOVE grammars, thus giving DMM++ a *compiler semantics*: The semantics of a DMM rule is the semantics of the GROOVE rule resulting from the transformation.

This section will define the the translation from DMM rulesets into GROOVE grammars, which has two major parts:

First, Ecore models are transformed into GROOVE graphs, and graphs resulting from such a transformation (and modified by a set of GROOVE graph transformation rules) are transformed back into the corresponding Ecore models.[5] Second, DMM rulesets are transformed into GROOVE graph transformation rules as mentioned above. Figure 6.14 shows an overview of the involved transformations.

Since the generated GROOVE rules need to be "compatible" to the GROOVE graphs generated from Ecore models, we will not present the above transformations separately. Instead, we will—feature by feature— show in parallel how the transformations work. Before we do that, we first point out the challenges we met in the next section. Section 6.3.2 will then present the actual transformation(s) as (basically) defined by Hausmann in [96]. Sections 6.3.3 to 6.3.5 will

[5] A first version of this transformation had been developed by Thomas Rheker as part of his bachelor's thesis [173]. Since then, the transformation has been advanced within the works of Bandener [11, 12], Bauer [16], and the author [192].

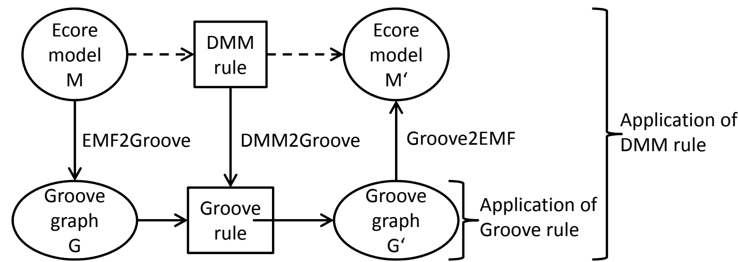

Figure 6.14: Overview of the transformations from DMM to GROOVE.

point out the newly introduced concepts for the treatment of universally quantified structures, attributes, and rule overriding. Finally, Sect. 6.3.6 will point out restrictions of the current state of the transformations (i.e., Ecore features which are not yet supported or only in a restricted way).

6.3.1 Challenges

As we have seen in Chapters 2 and 4, the structures of Ecore models and GROOVE graphs as well as DMM rules and GROOVE rules have a lot of similarities, but also differences. We start by pointing out the similarities:

- An Ecore model can structurally be seen as a graph, where objects are nodes, and references between objects are (directed) edges.
- Each object contained in an Ecore model has an explicit type from an Ecore metamodel; on the other hand, GROOVE allows to type nodes with the `type:` aspect.
- Ecore objects may have attributes – the same is true for GROOVE nodes.
- The notion of universal quantified structures and negative application conditions used by DMM and GROOVE are rather similar.
- As DMM rules, GROOVE rules allow to constrain the mapping of rule nodes to graph nodes by means of conditions of the nodes' attributes as well as the manipulation of attribute values.

However, not all Ecore and DMM concepts can be mapped to GROOVE concepts in such an easy way. The most important differences are:

- As mentioned above, each type being used in an Ecore model belongs to an Ecore metamodel (which technically is an `EPackage`). These metamodels each each have a unique URI; therefore, the name of an Ecore type in addition with the URI of the type's package uniquely identify that type. In contrast, GROOVE typenames are just strings.
- Not all datatypes which can be used within Ecore models have a natural representation in GROOVE. In particular, GROOVE does not support enumeration datatypes.

- As we have seen in Sect. 6.2, DMM rules can invoke other DMM rules, i.e., smallstep and premise rules. GROOVE does neither support the explicit invocation of GROOVE rules, nor does it allow to “factor out“ common structures of several rules into separate rules.
- GROOVE does not allow for the refinement of rules, as is the case with the DMM `OverridingRelation` elements.

As a result, some Ecore and DMM constructs can be translated rather easily into according GROOVE structures, but for others, much more effort is needed. In the next section, we will describe the transformation and show how we realized the advanced Ecore and DMM constructs with the simpler GROOVE constructs.

6.3.2 Basic Transformation Concepts

6.3.2.1 Preliminary Actions

Before the actual transformation of a DMM ruleset into the according GROOVE grammar can take place, a couple of preliminary actions are necessary: First of all, we have seen above that DMM premise rules are an extension of the invoking rule’s left-hand graph. Therefore, premise rules are merged into their invoking rules and do not need to be explicitly taken into account by the transformation. The merging of premise rules is explained below.

The second prelimary action is the computation of unique strings, each of which representing an `EClass` from one of the ruleset’s typing metamodels.

GROOVE rules match non-injectively by default. However, we have seen in Chapter 4 that GROOVE allows to configure this by means of a properties file belonging to each GROOVE grammar. Therefore, an according properties file is generated, and the property `matchInjective` is set to `true`.

Finally, a number of generic helper rules will be copied into the directory which will later contain the complete GROOVE grammar. These helper rules will at model execution time help to identify error states, e.g. in the case of failed invocations, or realize the semantics of soft rule overriding (see Sect. 6.3.5).

Merging Premise Rules The merging of premise rules is rather straightforward (it is implemented in class `de.upb.dmm.ruleset.util.Premise-RuleMerger` of plug-in `de.upb.dmm.ruleset`). Let *origRule* be the (smallstep, bigstep, or property) rule containing one or more invocations of premise rules. For each of the invoked premise rules *pRule*, the following happens:

1. A mapping from *pRule*’s context node to the invocation’s target node and from *pRule*’s parameter nodes to the invocation’s parameters is created. For each node of *pRule* neither being the context node nor a parameter node, a new node is created in *origRule* and mapped to that node.
2. For each *pEdge* of *pRule*, an edge is created in *origRule* (if it not yet exists) having the same properties as *pEdge*, but source and target node from *origRule* according to the mapping computed in step 1.
3. Finally, the conditions of *pRule*’s nodes are copied into the according *origRule* nodes – during this process, the target node of each `Attribute-Expression` is set to the according *origRule*’s node. Again, the mapping from step 1 is used.

4. Finally, for each invoked premise rule of *pRule* (if any), the above process is restarted recursively. The map is also built up recursively: Each context and parameter node of each premise rule will be mapped to its parent node, which might again be node of a premise rule (i.e., the invoking premise rule). The corresponding node in *origRule* will be the one in the sequence of mappings not having a parent node.

The algorithm terminates since invocations of premise rules must not contain circles. The result of the above algorithm is that each rule with premise invocations now contains the invoked rules' structures, and therefore only matches if these structures are contained within the state graph, as desired.

Computing Unique Type Strings and Type Hierarchy Since GROOVE[6] supports typing by providing a special `type:` aspect which will be mapped to typed nodes accoding to a type hierarchy, we have to create a unique string for each type of all involved DMM `Nodes`, and we have to compute the type hierarchy.

The types' unique strings are computed in class `de.upb.dmm.transformation._2groove.mapping.NameFactory` of the plug-in `de.upb.dmm.transformation._2groove.mapping`. The goal of the algorithm is to create type labels which are as simple and close to the original type names as possible. As such, the type strings are computed as follows: A bidirectional mapping between types and type names is created. Then, for each `EClass` contained in one of the involved metamodels,

1. Let *typeName* be the name of the `EClass`. If *typeName* is not yet contained in the mapping, we add it and are done. Otherwise,

2. we prefix *typeName* with the name of the package the *EClass* is contained in, and check again whether *typeName* is contained in the mapping. If it is not, we are done. Otherwise,

3. we repeat step 2 until we have reached the root package. If we haven't found a valid type name yet, we set *typeName* as the `EClass`'s package's namespace URI, followed by the `EClass`'s name, and add *typeName* to the mapping.

Since the names of `EClasses` must be unique within a `EPackage`, and since each `EPackage` must have a unique namespace URI, the above algorithm will always result in a valid mapping, and most unique type names will look like `MyClass` or `myPackage.MyClass`.

We have seen in Chapter 4 that GROOVE allows to define a type hierarchy, which will then taken into account when computing matchings. The according configuration string is computed by first identifying the direct subtypes of each involved `EClass`, and to then add the according subtyping relation `>`, using the unique type names as described above. The string is then added to the grammar's property file as value for property `subtypes`.

[6] At the time of writing, dedicated type graph support had already been added to GROOVE, but could not be reused by the DMM tooling due to time constraints.

Computing Elements to be Filtered Out We have seen in Sect. 6.2 that DMM rule nodes and edges are typed over the metamodels of the DMM ruleset they belong to, and that the matching of a DMM rule depends on the existence of the rule's nodes and edges within the state graph: For instance, if the DMM rule contains a node typed `A`, then that node can only be mapped to a state node also having type `A` or a subtype of `A` – the same holds for edges.

Let us now consider the other way around: If a model contains a node typed `A`, but none of the rules of a DMM ruleset contains any node typed `A` or supertype of `A`, then that node can neither affect the matching of any of the DMM rules, nor can it ever be changed (e.g., deleted) by the DMM rules. As such, that object (or reference) does not need to be translated when computing a GROOVE state graph from an Ecore model. The advantage of not translating these objects is obvious: The state graph's size will be reduced, leading to a faster and more memory-efficient computation of the transition system describing the model's behavior.

Consequently, as part of the preliminary actions, we collect all `EClasses`, `EReferences`, and `EAttributes` which are used in any of the rules of the DMM ruleset, and we use this collection of meta elements to decide during translation of an Ecore model which of the model's actual elements need to be translated at all (see Sect. 6.3.2.3).

6.3.2.2 Models and Rulesets

In general, the transformation will take an Ecore model and a DMM ruleset as input, and will produce a GROOVE grammar ready to be loaded into the GROOVE simulator, model checked etc. The Ecore model will be transformed into the start graph of the grammar, and each DMM rule (including the ones from imported rulesets) is transformed into a GROOVE rule to work on that start graph.

6.3.2.3 Objects and Links

First of all, all objects within the Ecore model and DMM nodes of the DMM rules are mapped to GROOVE nodes, each of which equipped with a typing edge whose label consists of `type:` and the unique type name of the object's/node's type. The only exception are objects whose type is not used within any of the rules (see Sect. 6.3.2.1); these objects are stored within a dedicated model for the sake of being available when translating a GROOVE state graph back into an Ecore model (see below).

In case of the Ecore model, the GROOVE node of the start graph additionally receives a label with the object's id (if any). In case of DMM rule nodes with role `ElementRole::CREATE`, `ElementRole::DESTROY`, or `ElementRole-::NOT_EXISTS`, the GROOVE node will also be equipped with an according aspect edge `new:`, `del:`, or `not:`.

The transformation of references between the Ecore model's objects is also straight-forward: As long as the references are used in any of the DMM rules, they result in a GROOVE edge between the according GROOVE nodes, being labeled with the reference's name. If a reference's cardinality is greater than one, one GROOVE edge is created for each of the referenced's objects, resulting

in many edges pointing from the referencing object to each of the referenced ones.[7]

The transformation of DMM edges to edges in the according GROOVE rules works exactly the same, and additionally, if the DMM edge has a role other than `ElementRole::EXISTS`, the resulting GROOVE edge is equipped with an according aspect as above.

The translation from GROOVE state graphs back to Ecore models is again not difficult: The source Ecore model (i.e., the one which served as start graph) is modified to reflect the different structure of the GROOVE state graph:

1. For each GROOVE node with an object id label, find the according object in the Ecore model. For each GROOVE node without an object id label, create an object of the according type within the Ecore model. Remember the mapping from GROOVE nodes to the existing or newly created Ecore objects.
2. Delete each Ecore object for which no corresponding GROOVE node could be found and which has also not been stored in the model containing the unused model elements, making use of the mapping computed in step 1.
3. For each outgoing reference edge of each GROOVE node:
    - If the reference has cardinality greater than 1, collect all referenced objects in a set, and then use that set as the value of the object's reference.
    - Else, directly use the referenced object as the object's reference's value.

The result will be an Ecore model which reflects the changes performed by the GROOVE rules on the start state graph and the intermediate graphs. Note that the translation of DMM `Quantifiers` will be dealed with in Sect. 6.3.3.

6.3.2.4 Rule Invocation

So far, all DMM constructs could be mapped to GROOVE contructs in a very natural way. For rule invocation however, this is not the case: The only means of control of rule execution GROOVE provides is that of rule priorities, but that doesn't help for implementing rule invocation.

As such, the approach suggested by Hausmann has been applied and extended:

- The GROOVE state graphs contain an explicit *invocation stack*.
- Bigstep rules can only match if the invocation stack is empty (and, of course, if their left-hand graph is contained in the state graph).
- In contrast, a smallstep rule can only match if an invocation corresponding to that rule is on top of the invocation stack.
- For property rules, the matching behavior can be configured: the `PropertyRule.matchAsBigstepRule` property dertermines whether a property rule can match always (i.e., whenever its left-hand graph is found in a state graph) or only if the invocation stack is empty.

[7]DMM yet has only very limited support for ordered references; see Sect. 6.3.6.

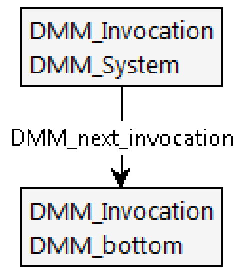

Figure 6.15: Empty invocation stack – only bigstep rules can match.

To achieve this very behavior, the GROOVE start state graph contains an empty invocation stack which is generated as part of the transformation. Such an invocation stack is depicted as Fig. 6.15. The node labeled `DMM_System` and `DMM_Invocation` represents the stack itself. It has a single outgoing edge labeled `DMM_next_invocation` pointing to the stack's first invocation node, which is in this case the special node representing the bottom of the stack (the node carrying the label `DMM_bottom`). As such, the stack is empty.

In the following, we will explain how the matching behavior of the DMM rules is achieved.

Bigstep Rules To make sure that a DMM bigstep rule only matches if the invocation stack is empty, the empty stack's structure is added to the bigstep rule. If the rule does not invoke any smallstep rules, this is the only change necessary.

Otherwise, the invocations are added to the stack by means of invocation nodes, and in the order being implied by the invocations' sequence numbers. For each invocation,

1. an invocation node is created which is additionally labeled with the invoked rule's name
2. the invocation's target node is marked with a GROOVE edge labeled `self` from the invocation node to the target node
3. the parameters of the invocation are marked with a GROOVE edge labeled `DMM_parameter` from the invocation node to the respective parameter node
4. to maintain the order of the parameters, `DMM_parameter_next` edges are used.

All the above constructs representing the invocations are additionally equipped with `new:` aspect edges. Finally, the invocation stack structure is modified such that the `DMM_next_invocation` node pointing to the stack's bottom is deleted, and new such edges are created from the stack to the first invocation node, and from the last invocation node to the stack's bottom invocation.

The procedure is illustrated in Figs. 6.16 and 6.17, which show a (very simple) bigstep rule and the GROOVE rule resulting from translating that bigstep rule. The bigstep rule contains one invocation of a rule execute().

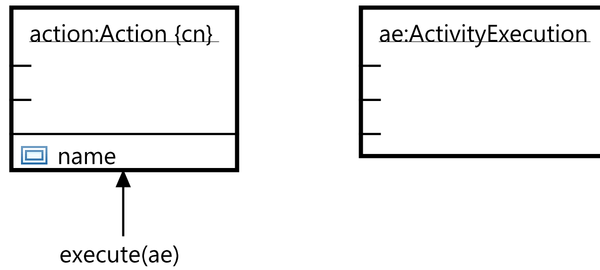

Figure 6.16: DMM bigstep rule with one invocation.

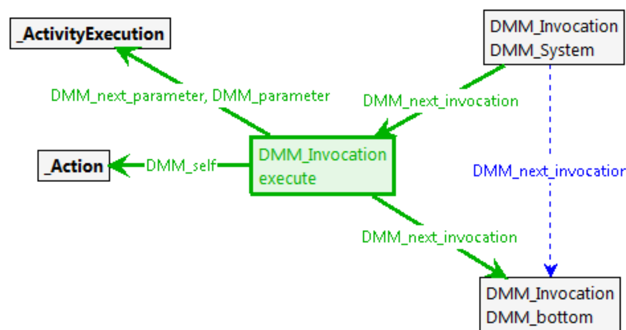

Figure 6.17: Invocation stack manipulation of bigstep rule with one invocation (resulting from the rule shown as Fig. 6.16).

In the resulting GROOVE rule, the invocation node representing that invocation is created and inserted into the invocation stack. After application of that rule, the invocation stack's structure will have changed: The stack's bottom invocation will not be the first invocation any more, and thus, no bigstep rule can match the containing state. The next section will show how that change is used to make only the invoked smallstep rules match in such a state.

Smallstep rules In case of smallstep rules, the translation to GROOVE needs to make sure that each smallstep rule only matches if it has been invoked. This is achieved by adding the invocation stack to the smallstep rule just as we did for bigstep rules as described above. The only difference is that the invocation stack is not empty, but has an invocation corresponding to the smallstep rule at its top.

Additionally, the context and parameter nodes of the smallstep rule need to be annotated. We have already seen in the last section that the target and parameter nodes of an invocation are annotated by dedicated edges (i.e., by an edge labeled `self` in the case of the target node, and by edges labeled `DMM_parameter` in the case of parameter nodes). If a smallstep rule is executed, its context and parameter nodes need to be bound to the target and parameter nodes of the invocation. This is achieved by according `self` and `DMM_parameter` edges which make sure that the rule's context node can only be bound to the node on which the invocation had been performed (since it is

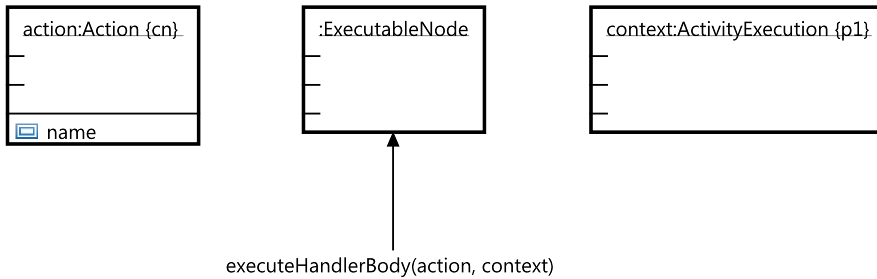

Figure 6.18: DMM smallstep rule with one invocation.

the only node to which a `self` edge is pointing) – for parameters, the idea is exactly the same.

Let us illustrate the above with an example: Fig. 6.18 shows a simple smallstep rule which contains a single invocation; the GROOVE rule resulting from translating that smallstep rule is depicted as Fig. 6.19. In the latter figure, we can see that the rule has an invocation node labeled `execute` in its context. This node refers to the containing rule itself. The invocation node is connected to the rule's context and parameter node via `self` and `DMM_parameter` edges. In the state graph, these nodes and edges have been created by the invoking rule (see e.g. the rule within Fig. 6.17 which creates exactly this structure). As such, if the smallstep rule from Fig. 6.19 matches a state graph, it is clear from the matching on which node the invocation has been performed by the invoking rule, and which nodes were passed as parameters. During execution of our smallstep rule, the according structure is deleted from the invocation stack, corresponding to the fact that the rule has indeed been executed.

Our smallstep rule additionally contains an own invocation. For this, an according structure is pushed on top of the stack (the green elements of Fig. 6.19). This works as explained for bigstep rules above, with one minor exception: In case of a bigstep rule, we want to make sure that the invocation stack is empty. As such, we have seen that a GROOVE rule resulting from a bigstep rule has the `bottom` invocation as its next invocation (meaning that there is no next invocation). For smallstep rules, the situation is slightly different: There might be an arbitrary number of invocations on the stack. Therefore, a smallstep rule has an arbitrary invocation as its next invocation (which can be seen at the bottom right of Fig. 6.19). At runtime, this invocation node might be mapped to an arbitrary invocation node of the state graph, including the `bottom` invocation node in case the smallstep rule is the last one in a sequence of invoked smallstep rules – the smallstep rule does not need to know what follows.

Finally, a smallstep rule can push new invocations on top of the stack, just as bigstep rules can do. The mechanism is exactly the same as for bigstep rules and will thus not be explained here again.

Property Rules In the case of property rules (which are not allowed to perform any changes, which includes the invocation of smallstep rules), the translation is simple: If the `matchAsBigstepRule` property is set to `true`, the structure of the empty invocation stack is added to the rule, making it only

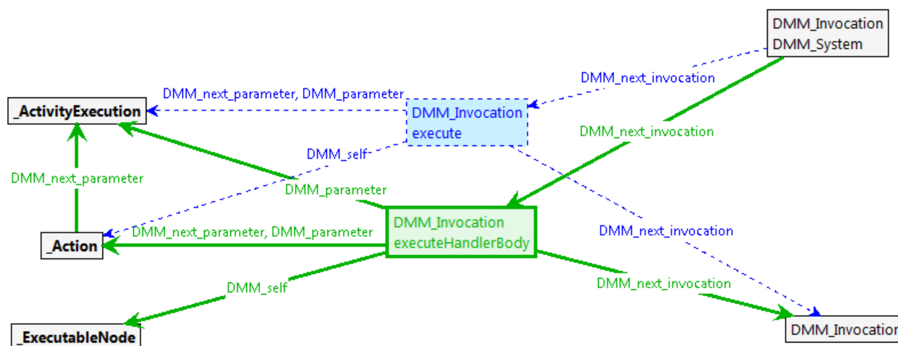

Figure 6.19: Invocation stack manipulation of smallstep rule with one invocation (resulting from the rule shown as Fig. 6.18).

match if the stack is indeed empty. Otherwise, no change is made to the rule. Thus, the rule matches as soon as the rule's structure is contained in a state graph; the rule's matching is not affected by the state of the invocation stack.

6.3.3 Universal Quantified Structures

The treatment of universal quantified structures (UQS) by Hausmann was hampered by the fact that GROOVE at time of writing his PhD thesis [96] did not support UQS in any way. Therefore, Hausmann fell back to the concept of *rule schemes* [198], where a scheme can be *unfolded* into a number of single rules, each treating a fixed number of elements the UQS is mapped to.

In the meantime, GROOVE has introduced a powerful notion of UQS: the concept of *nested graph transformation rules* introduces predicates $\forall$, $\forall^{>0}$, and $\exists$ into GROOVE rules. These can be used to express things like "If all incoming places of a transition carry a token, delete these tokens, and create one token on each outgoing place" within one rule.

As a result, we have reused this mechanism to implement UQS for DMM. For this, we have defined the `ElementRoles` `ElementRole::ZERO_TO_MANY`, `ElementRole::ONE_TO_MANY`, and `ElementRole::NESTED`, corresponding to the GROOVE predicates $\forall$, $\forall^{>0}$, and $\exists$.

Additionally, we have defined the notions of *UQS cluster* and *nested cluster*, meaning a set of nodes (transitively) being neighbors of each other and all having the same `ElementRole`. By adding the restriction that each nested cluster must be connected to exactly one UQS cluster (see Sect. 6.2.3.5), the translation is straight-forward:

- For each UQS or nested cluster within a DMM rule, create an according GROOVE predicate node in the resulting GROOVE rule, and connect all nodes of the cluster with that predicate node by an edge labeled `in`.
- For each nested cluster: Connect the according predicate node with the one predicate node representing the nested cluster's UQS cluster by an edge labeled `at`.

Let us demonstrate the above with a simple example: The rule action.start()# depicted as Fig. 6.20 contains a node with element role `ElementRole::ONE_TO`

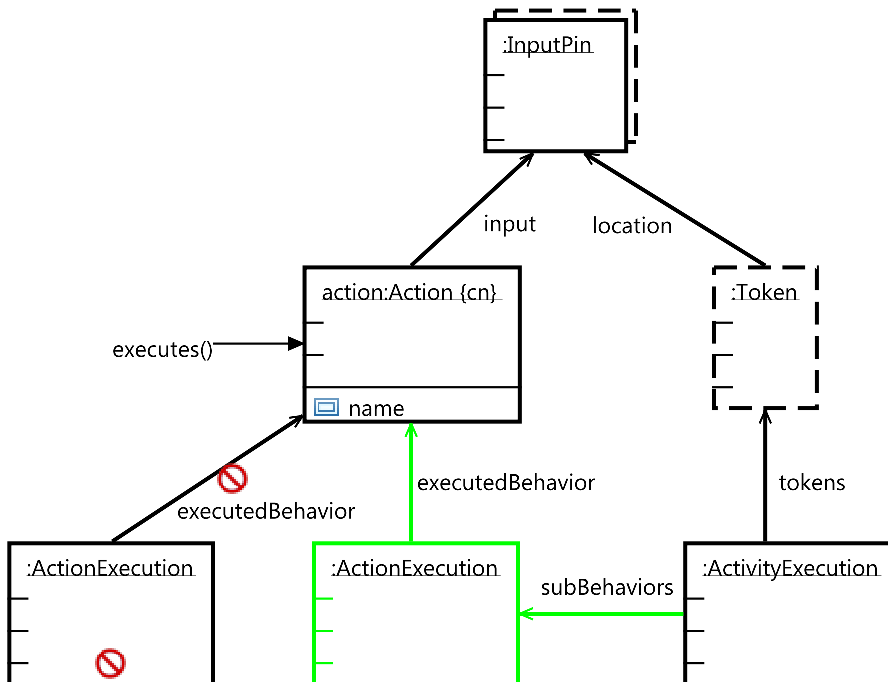

Figure 6.20: DMM rule with one UQS node and one nested node.

_MANY (the node typed InputPin) and one node with element role Element-Role::NESTED (the node typed Token). Here, the former node forms a UQS cluster, and the latter forms a nested cluster. The semantics of the rule is as follows: The rule matches if there exists an Action which has at least one InputPin, and if all InputPins of that Action carry at least one Token. Additionally, the Action must not yet be executed, i.e, it must not have an attached ActionExecution. Application of the rule will then create an ActionExecution node and invoke rule action.execute().

In the resulting GROOVE rule in Fig. 6.21, the UQS structure can be seen to the right: the InputPin and Token nodes are connected to the $\forall^{>0}$ and $\exists$ nodes, and the $\exists$ node is connected to the $\forall^{>0}$ node. The manipulation of the invocation stack can be seen to the top left of the rule, and the handling of the ActionExecution at the bottom left. Overall, the rule implements exactly the behavior described above.

6.3.3.1 Invocations on Universally Quantified Nodes

The intuition of invoking a smallstep rule with a UQS node as target node is that one invocation of the according rule is performed for each of the nodes of the state graph to which the UQS node has been mapped.

Note that this statement does not consider any order in which the invocations are performed on the nodes – the assumption is that the order of invocations does not matter. This must of course be respected by the language engineer when creating a DMM semantics specification.

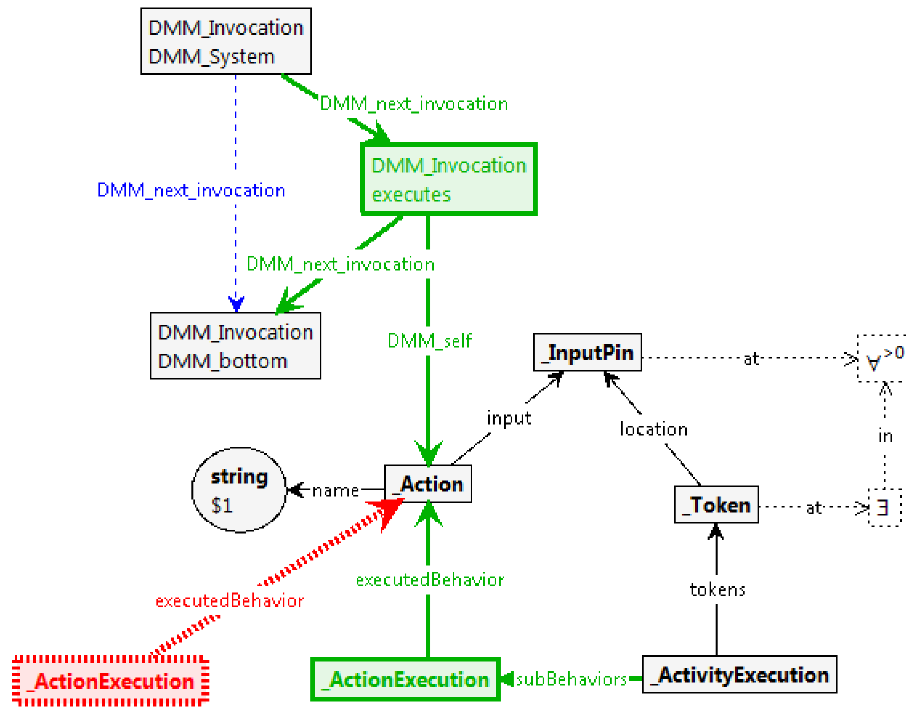

Figure 6.21: GROOVE rule resulting from the DMM rule in Fig. 6.20.

We have seen earlier how rule invocation is implemented on the GROOVE side by maintaining an invocation stack in which the invocations of rules are inserted. However, we face a problem here: The UQS mechanism of GROOVE does not allow to create a "linked list" structure out of a set of state nodes mapped to a UQS node, at least not within one rule.[8]

To cope with that, the transformation works as follows:

- The invoking rule is mapped to a GROOVE rule such that for each invocation on a UQS node n, one `UQSInvocation` node is inserted into the invocation stack. Additionally, the rule will create one `Invocation` node per state node mapped to n, and connect these `Invocation` nodes to the `UQSInvocation` node via a `DMM_handleInvocation` edge.
- Additionally, some generic helper rules exist. These rules have higher priority than all other GROOVE rules – they will match as soon as the invocation stack contains a `UQSInvocation`, and will insert the `Invocation` nodes created above into the invocation stack (and finally remove the `UQSInvocation`) node.
- Finally, the normal DMM invocation mechanism takes care of performing the invocations.

This solution has the following advantages: The GROOVE ruleset resulting from translating a DMM ruleset will have the same number of rules (i.e., a

[8]Note that Rensik has indeed considered to change this [39], but to our knowledge this is not implemented yet.

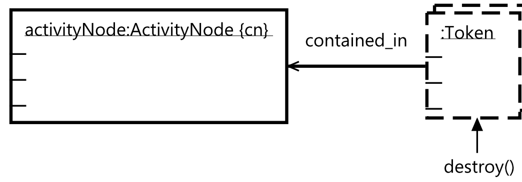

Figure 6.22: DMM smallstep rule with invocation on a universally quantified node.

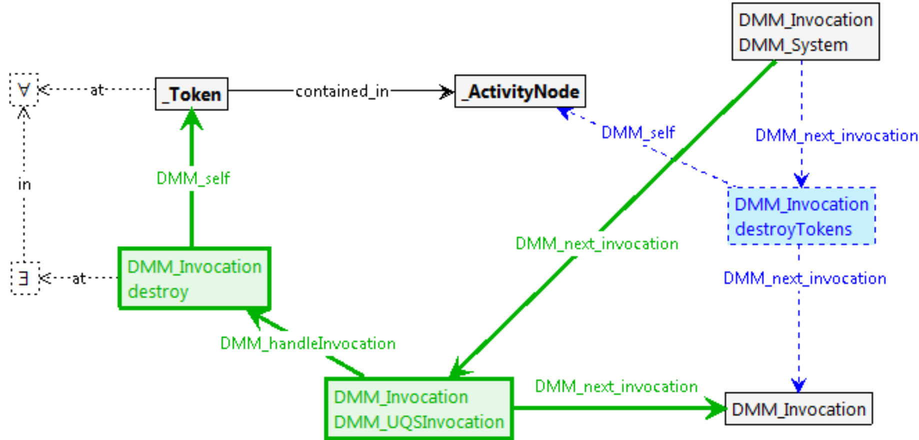

Figure 6.23: Invocation stack manipulation of smallstep rule with invocation on a universally quantified node (resulting from the rule shown as Fig. 6.18).

DMM ruleset of size n will result in a GROOVE ruleset of size $n + k$, where k is a constant number of rules). Additionally, since the order of invocations does not matter, the resulting transition system will contain a linear structure of applications of helper rules (in contrast to all possible orders of invocations in case order would matter).

Let us demonstrate the above with an example: The DMM smallstep rule shown as Fig. 6.22 contains an invocation on the `Token` UQS node. The GROOVE rule resulting from this rule is shown as Fig. 6.23. To the right, we see the insertion of the `DMM_UQSInvocation` node into the invocation stack. To the left, we can see that for each `Token` object, a `DMM_Invocation` node is created and connected to the `DMM_UQSInvocation` node.

The resulting structure will then be resolved by helper rules such as the ones shown as Fig. 6.24 and 6.25. The former rule inserts one of the invocation nodes into the invoation stack and removes the `DMM_handleInvocation` edge; this rule will be applied until all invocations are inserted into the stack. The latter rule then removes the `DMM_UQSInvocation` node from the invocation stack. Note that more helper rules exist which take care of bigger numbers of invocations at once to avoid extensive cluttering of the transition system. Note also that the helper rules will only be generated if the DMM ruleset to be transformed contains at least one invocation on a UQS node.

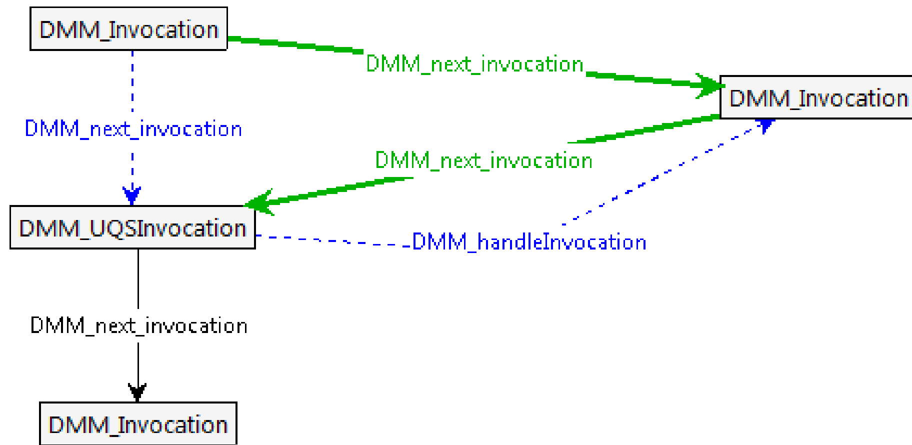

Figure 6.24: Helper rule inserting an invocation into the stack.

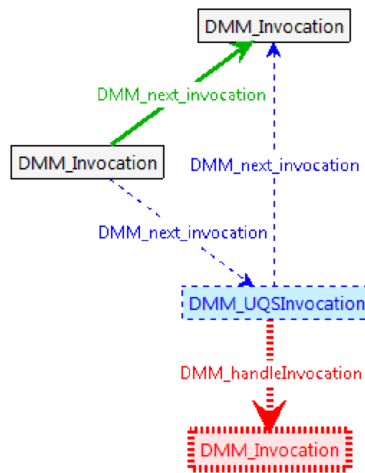

Figure 6.25: Helper rule removing the UQSInvocation from the stack.

6.3.4 Attributes

The treatment of attributes in DMM is two-fold: Attributes and their values have to be translated to and from GROOVE state graphs (as long as they are used by any of the DMM rules, see Sect. 6.3.2.1), and the DMM rule constructs `Condition`, `Assignment`, and `EmphasizedNodeAttribute` have to be dealt with.

6.3.4.1 State Translation

There are two basic kinds of datatypes for attributes supported by DMM: *enumerations* and *primitive datatypes*. The translation of these two kinds is rather different.

Since GROOVE does not support the notion of enumerations, these have to be mapped into common GROOVE constructs. This is done as follows:

- For each enumeration, a dedicated node is created within the GROOVE state graph and labeled with the enumeration's name.
- Additionally, for each literal of each enumeration, a node labeled with the literal's string representation is created and connected to the node representing the enumeration via a `DMM_Enum` edge.
- Then, if a node's type has an attribute of enumeration type, the node will have an edge to the enumeration literal representing the object's attribute value, and labeled with the attribute's name.

In contrast, attributes having primitive datatypes as types are mapped to GROOVE attributes. This is done according to Table 6.4. The shown mapping is implemented in class `de.upb.dmm.transformation._2groove.common.AttributeMapper` of the `de.upb.dmm.transformation._2groove.common` plug-in.

Fig. 6.26 shows an excerpt of a GROOVE state. The `PseudoState` node in the figure's top right represents an object of type `PseudoState`. This object has two attributes: `kind` of type `PseudoStateKind` (an enumeration) and `name` of type `string`. The graph structure representing the enumeration `PseudostateKind` fills the main part of Fig. 6.26. The `PseudoState` in the graph has kind `Initial` and name "InitialState" – the latter is expressed by the attribute node to the top right of the figure, which is connected to the `PseudoState` node by an edge labeled with the attribute's name (which is `name` in this case).

The translation from GROOVE graphs back to Ecore models is straightforward: For each attribute, the attribute value is received from the state graph (either by identifying the literal value in case of an enumeration datatype, or by mapping the primitive value back to the according Java value). Using the attribute's name (which is received from the edge connecting the object node with its attribute value), the value is then set on the Ecore object.

6.3.4.2 Conditions

We have seen earlier that Conditions are used to restrict the matching of rule nodes: Nodes can only be matched to nodes of the state graph which fulfill the

Table 6.4: Mapping between Java and GROOVE datatypes

Java type	GROOVE aspect
java.lang.String	string:
char	string:
java.lang.Character	string:
java.util.Date	string:
short	int:
java.lang.Short	int:
byte	int:
java.lang.Byte	int:
int	int:
java.lang.Integer	int:
long	int:
java.lang.Long	int:
java.math.BigInteger	int:
float	real:
java.lang.Float	real:
double	real:
java.lang.Double	real:
java.math.BigDecimal	real:
boolean	bool:
java.lang.Boolean	bool:

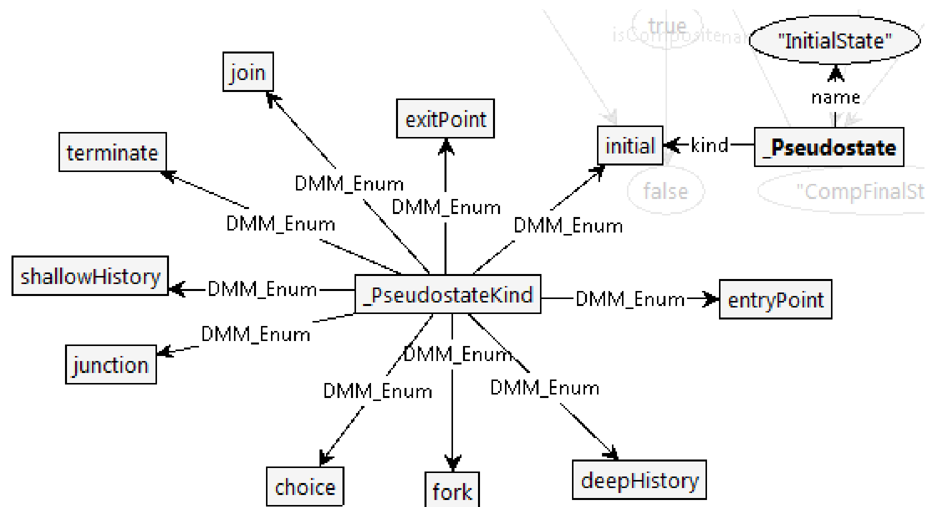

Figure 6.26: GROOVE state graph with a node having attributes of type EEnumeration and string.

Table 6.5: Mapping between DMM operators and GROOVE productions

DMM operator	GROOVE production
`&`	`and`
`|`	`or`
`!`	`not`
`+`	`add`
`-`	`sub`
`*`	`mul`
`/`	`div`
`%`	`mod`
`min`	`min`
`max`	`max`
`abs`	`abs`
`>`	`gt`
`>=`	`ge`
`==`	`eq`
`<=`	`le`
`<`	`lt`

nodes' conditions, where a condition is an expression over attribute values of a rule's nodes which evaluates to boolean.

To achieve this behavior in GROOVE, the following transformations are performed:

- State graph: Each GROOVE node is equipped with a *condition binder*, i.e., the boolean value `true` which is bound to its node by means of a `DMM_ConditionBinder` edge.
- Rules: Each condition's expression is translated into a tree of GROOVE nodes, where leafs are literal values or attribute values of rule nodes, and where the inner nodes are GROOVE productions. The translation of DMM expression elements is straight-forward: `LiteralExpressions` become GROOVE literal values, `AttributeExpressions` become references to attributes of rule nodes, and `OperationExpressions` become GROOVE productions. For a mapping of DMM operations to GROOVE productions see Table 6.5.
- Finally, the "outer" expression of the condition (which evaluates to boolean) is bound to the condition binders `true` attribute value via an `eq` production.

The resulting structure makes sure that a state node can only be mapped to a rule node if the rule node's conditions all evaluate to `true`, resulting in the desired behavior.

Note that the condition binder is necessary for one particular reason: Usually, a node's conditions will refer to attributes of that node, and thus, an expression tree resulting from a condition could be bound to the node through those attributes. However, this does not have to be the case. For instance, consider a custom Petri net semantics where a transition shall fire as soon as the sum of the weights of tokens on incoming places is bigger than the sum of

tokens of outgoing places. Clearly, the condition formalizing this requirement belongs to the transition.

However, the condition only refers to attributes of the incoming and outgoing places, but not of the transition itself. Transforming such a condition would result in an expression tree which would not be connected to the transition, but would instead affect the complete rule's matching. Therefore, every expression tree is bound to the owning node by means of the node's condition binder.

6.3.4.3 Assignments

The translation of `Assignments` into GROOVE structures is very similar to that of `Conditions` as described above: The assignment's expression is translated into an expression tree, just as we have seen for conditions. However, that expression tree (which of course does not have to evaluate to a boolean value) is not bound to the owning node by means of a condition binder, but through the attribute of the assignment's `AttributeExpression`.

More concretely, the translation deletes the edge pointing to the attribute's value node before rule application, and creates a new edge pointing to the result of the expression tree's evaluation and labeled with the attribute's name.

6.3.4.4 Node Creation

In case of DMM nodes with role `ElementRole::CREATE`, attributes need to be treated also. For such nodes, no conditions are allowed (see Sect. 6.2.3.5). However, assignments are allowed and can be used to compute the initial value of attributes of the newly created node. As such, the transformation ensures the following:

- A condition binder (see Sect. 6.3.4.2) has to be created. This is because otherwise, DMM nodes on which a condition is defined could not be mapped to a newly created node.
- The attribute values of the newly created node are either initialzed with the attribute's datatype's default value or with the result of an assignment.

6.3.4.5 Negative Application Conditions

In case of DMM nodes with role `ElementRole::NOT_EXISTS`, no assignments are allowed (see Sect. 6.2.3.5). However, conditions are allowed and can be used to further restrict the mapping of state nodes to the negative application condition. Since the conditions belong to the negative application condition, all nodes belonging to the conditions' expression trees are equipped with a GROOVE NAC edge.

6.3.4.6 Emphasized Attributes

The purpose of `EmphasizedNodeAttributes` is to see the value of certain attributes in the transition system at the time a rule is applied. For this, GROOVE has a dedicated construct: For each `EmphasizedNodeAttribute` of a rule, a special GROOVE node is created and labeled with `par=$<nr>:`, where `<nr>` is the index of the according `EmphasizedNodeAttribute` in the rule's list of `EmphasizedNodeAttributes` (this is why in the DMM

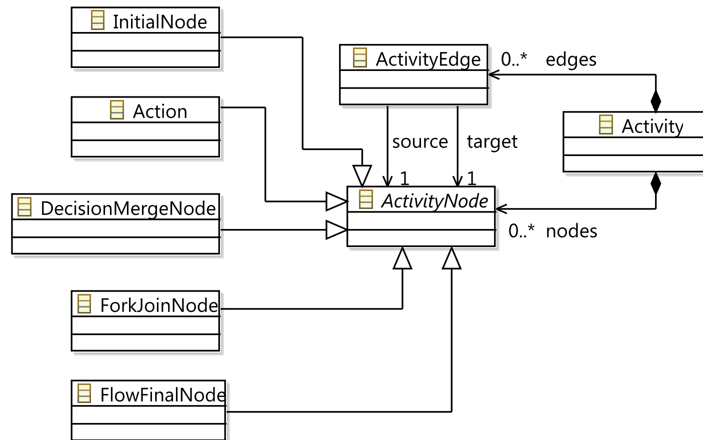

Figure 6.27: Simplified syntax metamodel of UML Activities.

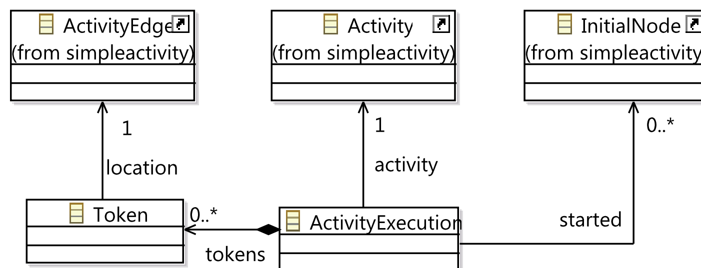

Figure 6.28: Runtime metamodel of UML Activities, referring to elements of the syntax metamodel depicted as Fig. 6.27.

metamodel, `EmphasizedNodeAttributes` are not owned by the node they are associated to, but by the according rule.).

Additionally, an edge is created from the DMM node being associated to (but not owning) the `EmphasizedNodeAttribute`, and that edge is labeled with the attribute's name. This tells GROOVE which attribute's value to display.

6.3.5 Rule Overriding

While working with DMM specifications, one recurring problem was the restriction that they can not be refined in the sense that some behavior (i.e., rules) of that specification can be changed. This hampered reusability of DMM specifications quite a bit. As a result, we decided to introduce a notion of *rule overriding* into the DMM language: An overriding relation allows to model the overriding of rules. Since DMM specifications can import other DMM specifications, the refining of existing DMM specifications can be achieved.

Since rule overriding [62, 192] is one of the main contributions of this thesis, we show the definition and implementation of it in detail in this section, which is based on [192].

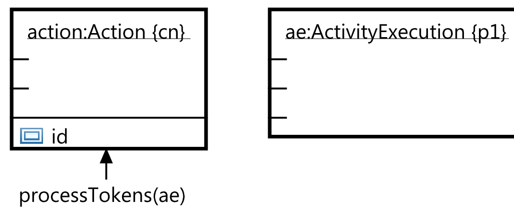

Figure 6.29: DMM rule action.execute(ActivityExecution)#.

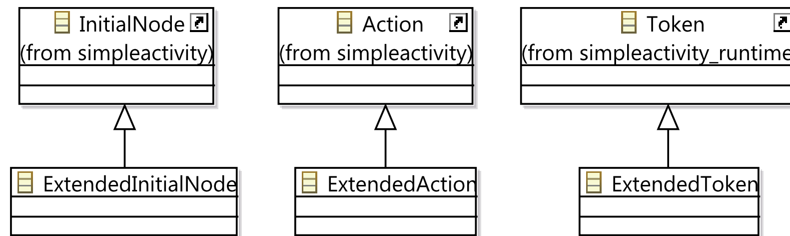

Figure 6.30: Metamodel extending the syntax and runtime metamodels (Fig. 6.27 and 6.28).

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

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

In other words: Our extending metamodel contains three classes which are referring to and inheriting from classes from the syntax and the runtime metamodel seen in Fig. 6.27 and 6.28 (note that it would be conceptually cleaner to again separate into an extending syntax and runtime metamodel). The resulting metamodel is depicted as Fig. 6.30.

Let us now define the behavior of the new elements. The rules extendedInitialNode.flow()# and extendedAction.execute(ActivityExecution) are shown as Fig. 6.31 and 6.32. The task of the former is to create `ExtendedTokens` instead of just `Tokens`; despite that, the rule is similar to rule initialNode.flow()# from the original ruleset. Rule extendedAction.execute(ActivityExecution) is more complex than its counterpart, rule action.execute(ActivityExecution): The rule requires that at least one of the `ExtendedAction`'s incoming `ActivityEdges` is carrying an `ExtendedToken` owned by the passed `ActivityExecution`

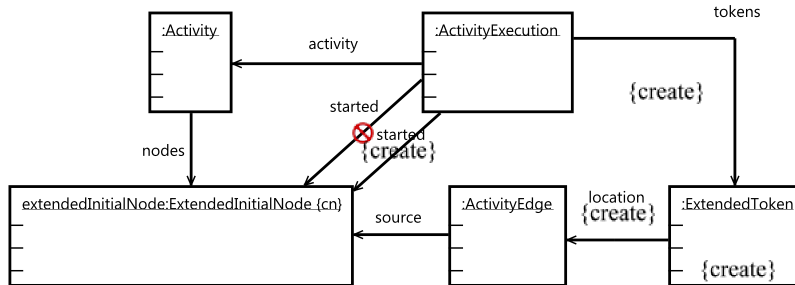

Figure 6.31: DMM rule extendedInitialNode.flow()#.

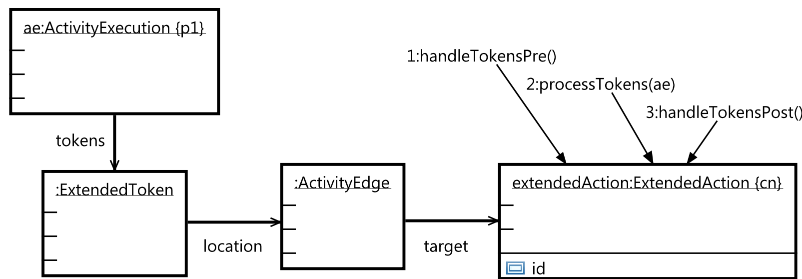

Figure 6.32: DMM rule extendedAction.execute(ActivityExecution).

node. If this is the case, the rule will perform some pre- and postprocessing on the flowing tokens (we do not show the corresponding rules here).

Unfortunately, it is not that easy. As we have concluded in Sect. 5.3, DMM in its current state only allows to add rules to an existing ruleset. These added rules do not influence the application of the original rules, though: If one of the old rules as well as one of the newly added rules matches a state, both of them will be applied when computing a transition system, therefore leading to a branch.

This might be the desired behavior, but in some cases it is not. For instance, what does that mean for our new rules extendedInitialNode.flow()# and extendedAction.execute(ActivityExecution)? It is easy to see that rule initialNode.flow()# matches whenever rule extendedInitialNode.flow()# matches, and the same holds for rules action.execute(ActivityExecution) and extendedAction.execute(ActivityExecution). This is due to the fact that the left-hand graph of, e.g., initialNode.flow()# basically is a subgraph of the other rule's left-hand graph. The only exception is the typing: In the new rule, some node's type is not the same type but a subtype of the old rule's node's type. Referring to Def. 6, the above follows.

In a transition system, a state where rule extendedInitialNode.flow()# matches will therefore give rise to (at least) two new states. One is derived by applying rule initialNode.flow()#; in this state, a simple `Token` has been created. The other state is the result of an application of rule extendedInitialNode.flow()# and does contain a newly created `ExtendedToken`. In other words: If our

model contains an `ExtendedInitialNode`, the resulting transition system will contain an undesired state where not an `ExtendedToken` but a `Token` has been created.

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

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

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

6.3.5.1 Prerequisites

In the previous chapters, we have not formalized the matching of rules. This was on purpose: In Sect. 6.1, we have explained that a compiler semantics is more appropriate for applying DMM specifications then a direct formalization of DMM's semantics. However, for the sake of explaining rule overriding in a compact yet precise way, we will now capture the definitions of DMM rules and rule matching in two definitions, which will then be referred from later parts of this section.

We start with the definition of a DMM rule:

Definition 5 (DMM Rule) *A DMM rule is a tuple* $R = (name, G_L, G_R, NACs, contextNode, params, invocations)$ *where* G_L *and* G_R *are graphs typed over a metamodel* M*,* $NACs$ *is the set of negative application conditions,* $contextNode \in N_{G_L}$ *is the context node of the rule (*N_{G_L} *is the set of nodes of* G_L*),* $params \in N_{G_L} \times \cdots \times N_{G_L}$ *is the (possibly empty) list of parameter nodes, and invocations is the list of invocations of other DMM rules (which are pushed on the invocation stack after application of the invoking rule).*

The above definition of DMM rules basically transfers the information contained in the DMM metamodel (see Sect. 6.2.2.2 on page 63) into the world of set theory. Next, we use this definition to explain rule matching:

Definition 6 (Rule matching) *Let* G *be a graph typed over a metamodel* M*, let* R *be a DMM rule typed over the same metamodel.* R matches G *if the following conditions hold:*

1. *The invocation stack is either empty if R is a bigstep rule or has an according invocation on its top if R is a smallstep rule.*

2. *A morphism m from G_L to G can be found such that the types of the matched nodes in G are of the same type or a subtype of the matching nodes in G_L.*

3. *m can not be extended to m' such that m' is a morphism from any of the rule's $NACs$ to G.*

Again, this definition captures (and formalizes to some extent) what we have explained earlier. We are now ready to explain the relations between rules participating in an overriding relation in the next section.

6.3.5.2 Relation of Overriding Rules

First of all, the names of two rules participating in an *overrides* relation must be equal. Then, the context node of the overriding rule must be a subtype of the overridden rule's context node. This is because we want to mimic overriding as it can be found in object-oriented languages (recall from Sec. 6.2.2.2 that the context node can be seen as owning the rule, similar to a class owning its methods). For the same reason, the parameter types of the two rules must be the same or subtypes of the overridden rule's parameters, i.e., the first parameter of the overriding rule needs to have the same type or a subtype of the first parameter of the overridden rule and so on.

So far, our arguments have been based on similarity with well-known object-oriented concepts. There is one important difference between a method and a DMM rule, though: Correct invocation of a method only relies on syntactical constraints (and can therefore be checked by a compiler). For a DMM rule, the situation is different: Recall from Sect. 6.3.2.4 that the invocation of a rule might fail. This problem is not directly related to overriding at all. As we have seen before, a DMM specification which—when executed—gives rise to an intermediate state where a smallstep rule is invoked but cannot match is considered to be incorrect.

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

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

The following definition collects all requirements identified in this section:

Definition 7 *Let R, R' be DMM rules as defined in Sect. 5. R can only override R' if the following requirements are fulfilled:*

1. *R and R' have the same name.*

2. R's context node has a type which is a subtype of the context node of R'.

3. R has the same number of parameters as R', and the parameters' types are the same or are subtypes of the types of the parameters of R'.

4. Let G be an arbitrary graph typed over the same metamodel as R and R'. It must then be the case that $R\, matches\, G \Longrightarrow R'\, matches\, G$.

6.3.5.3 Implicit and Explicit Overriding

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

In DMM, the situation is different: As we have seen in Sect. 6.3.2.4, several rules having the same signature can exist. These rules will often have different left-hand graphs, but this does not even have to be the case. To achieve maximum flexibility, a rule therefore needs to explicitly declare the rules it overrides. This leads to the following modified definition of a DMM rule:

Definition 8 (Overriding DMM Rule) *Let R be a DMM rule as defined in Sect. 5. An* overriding DMM rule *is a tuple $R_O = (R, overrides)$ where overrides is the set of DMM rules overridden by this rule, such that all rules in overrides fulfill the requirements formulated in Def. 7.*

Note that the directed graph consisting of rules as nodes and overriding relations as edges must not contain cycles. Note also that we will later use the notation $R_O\, overrides\, R'_O$ if R'_O is contained in the set of overridden rules of R_O (formally: $R_O\, overrides\, R'_O :\Leftrightarrow R_O = (R, overrides) \wedge R'_O \in overrides$).

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

6.3.5.4 Complete Overriding

The idea of the first alternative is that an overridden rule can only match if the node the rule's context node is mapped to does not have an actual type for which an overriding rule exists. Definition 6 is then modified as follows:

Definition 9 (Rule matching (complete overriding)) *Let G be a typed graph, let R be an overriding DMM rule as defined in Def. 5.* R matches G *if the conditions listed in Def. 6 hold, and additionally:*

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

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

Having rule initialNode.flow()# not match anymore solves our problem of two rules being applied (leading to an unwanted branch in the transition system), but only partly: Assume that rule extendedAction.execute(ActivityExecution) completely overrides rule action.execute(ActivityExecution). Now, if only usual `Tokens` arrive at an `ExtendedAction`, rule extendedAction.execute(ActivityExecution) will not match (since it requires an `ExtendedToken` on at least one of its incoming edges).

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

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

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

6.3.5.5 Soft Overriding

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

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

Definition 10 (Rule matching (soft overriding)) *Let G be a typed graph, let R be an overriding DMM rule as defined in Def. 5. R* matches *G if the conditions listed in Def. 6 hold, and additionally:*

4) No overriding DMM smallstep rule R' exists such that $R' \in overridden_R$ and R' matches G

The main difference is that in this definition it does not suffice for rule R' to exist to prevent rule R from matching G – additionally, R' itself needs to match

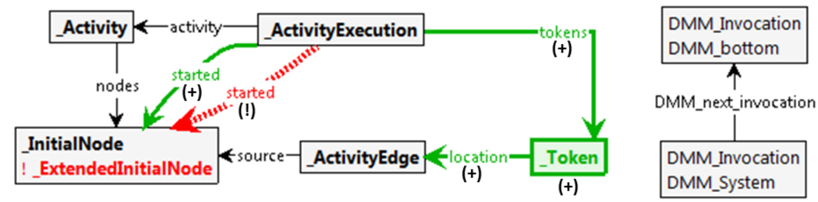

Figure 6.33: GROOVE rule resulting from translating the overridden rule initialNode.flow()#.

G. It is easy to see that this indeed solves our problem from the last section: In case that at least one `ExtendedToken` arrives at an `ExtendedAction`, rule extendedAction.execute(ActivityExecution) will match and be applied. Because of this, rule action.execute(ActivityExecution) will not match. However, if only `Tokens` arrive at `ExtendedAction`, rule extendedAction.execute(ActivityExecution) will not match. This "activates" rule action.execute(ActivityExecution); the behavior of the `ExtendedAction` falls back to that of the `Action`.

There is another, more subtle difference between definitions 9 and 10: Only smallstep rules can participate in a soft overriding relation. The reason for this lies in the way soft overriding is translated into corresponding GROOVE structures – we will explain this in the next section. However, this is actually no restriction: If the soft overriding mechanism is needed in the context of a bigstep rule, that rule can be changed such that its content is copied into a smallstep rule, which is then invoked by the bigstep rule. The created smallstep rule can now be softly overridden.

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

6.3.5.6 Translation into GROOVE Rules

The implementation of matching with complete overriding as introduced in Def. 9 is straightforward to implement: While generating the GROOVE rules to represent the DMM rules, the transformation tool needs to keep track of complete rule overriding relations. For every rule which is completely overridden, the transformer identifies the types of the context nodes of the overriding rules. For each of those collected types, it then adds a negative application condition to the context node of the overridden rule, preventing it from matching in a context where an overriding rule exists.

The new translation of the overridden rule initialNode.flow()# is depicted as Fig. 6.33. The mentioned negative application condition can be seen at the bottom of the node typed `InitialNode` – as desired, it prevents the rule from being applied in the context of an `ExtendedInitialNode`. Note that the translation of rule extendedInitialNode.flow()# does not change.

The implementation of matching with soft overriding as defined in Def. 10 is more difficult, as the actual rule that is to be executed has to be identified

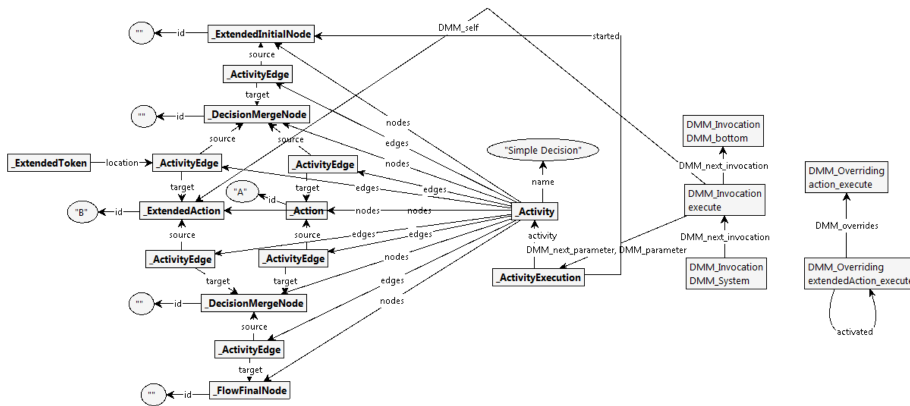

Figure 6.34: GROOVE state after application of rule action.flow()#.

dynamically at runtime. The basic idea is to equip rules participating in a soft overriding relation with additional structures which enforce that the rules can only match if they are "activated". An additional helper rule makes sure that the rules are activated one after the other (according to their participation in the overrides relation) until the most specialized *and* matching rule is found and applied.

Let us investigate this in more detail: While transforming the DMM ruleset, the transformer builds up a *rule hierarchy graph*, where nodes correspond to rules, and edges correspond to *overrides* relations between the rules. This graph will be part of the start graph. Figure 6.34 shows the state after application of rule action.flow()#; the rule hierarchy graph resulting from our rules action.execute(ActivityExecution) and extendedAction.execute(ActivityExecution) can be seen on the right side of the figure (compared to the start state, it has not changed yet). Note that in the start graph, the node(s) corresponding to the most specialized rule(s) carry `activated` edges – this is where the search for an applicable rule will start in case an overriding smallstep rule is invoked.

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

Figure 6.35 shows the GROOVE rule resulting from translating rule action-.execute(ActivityExecution). Its semantics is as follows: On the left side, the nodes resulting from the actual DMM rule we have shown as Fig. 6.29 can be seen. The structure to the right makes sure that the rule indeed matches as desired: First, the `action_execute` node carries an `activated` edge which is to be deleted; in other words, the rule can only match if the rule is activated within the rule hierarchy graph as described above. Note that we will explain the NAC below that node in the next section. Above to the right, the `extendedAction_execute` node gets a new `activated` edge, corresponding to the fact that the most specialized rules are activated again (in

Figure 6.35: GROOVE rule action.execute(ActivityExecution) with structure realizing soft rule overriding.

Figure 6.36: GROOVE rule moving the `activated` edge to the next level.

a more complex setting, there would probably be more rules to be activated). The part below ensures that all other `activated` edges are deleted as well.

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

The helper rule is depicted as Fig. 6.36. Its semantics is as follows: The left part of the rule makes sure that it only matches if the state contains at least one `DMM_Overrides` node carrying an `activated` edge. If this is the case, it will delete all `activated` edges. The right part is responsible for moving the `activated` edge to the next level: Every `DMM_Overrides` node which has an overriding and activated `DMM_Overrides` node gets an `activated` edge. The rule is generic in the sense that its nodes do not carry any rule labels. As a result, the rule can take care of all parts of the rule hierarchy graph, independent of the concrete rule names carried by the nodes of the rule hierarchy graph.

It now becomes clear why bigstep rules can only participate in complete overriding relations: We have seen that the realization of soft overriding requires a kind of dynamic binding. If such a rule is overridden, it will often result in several applications of the helper rule from Fig. 6.36 until a matching smallstep rule is found (we will see an example in Sect. 6.3.5.7). However, only rules

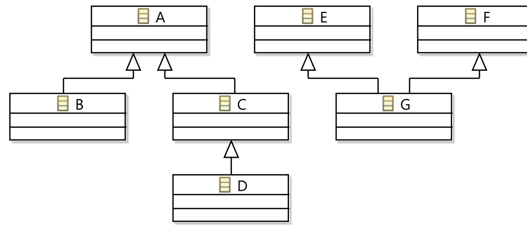

Figure 6.37: Metamodel of the example.

having the according signatures have to be taken into consideration.

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

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

6.3.5.7 Complex Rule Overriding

The presented soft rule overriding scenario serves well to demonstrate the general idea, but is rather simple. In this section, we want to investigate a more complex example. It consists of three parts: in Fig. 6.37, a metamodel is shown, Fig. 6.38 shows a sequence of states of the rule hierarchy, and in Fig. 6.39, a labeled transition system can be seen.

The metamodel in Fig. 6.37 consists of several classes, some of which are in an inheritance relation. To keep the example compact, we have chosen abstract class names such as `A`. The example contains multiple inheritance: class `G` inherits from classes `E` and `F`.

Let us now investigate the sequence of states in Fig. 6.38. They depict the rule hierarchy: The graph at the top shows the initial state. It has been computed during the transformation from the model into the corresponding GROOVE state graph. Again to keep the example compact, we have omitted the `DMM_Overriding` labels from the nodes of the rule hierarchy graphs, and we use the label `o` instead of `DMM_overrides` and `a` instead of `activated`.

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

To demonstrate the soft overriding mechanism, we have created all these rules as dummy rules, i.e., despite the rule hierarchy graph, they do not change the state (we have even removed the invocation stack). Each rule matches if

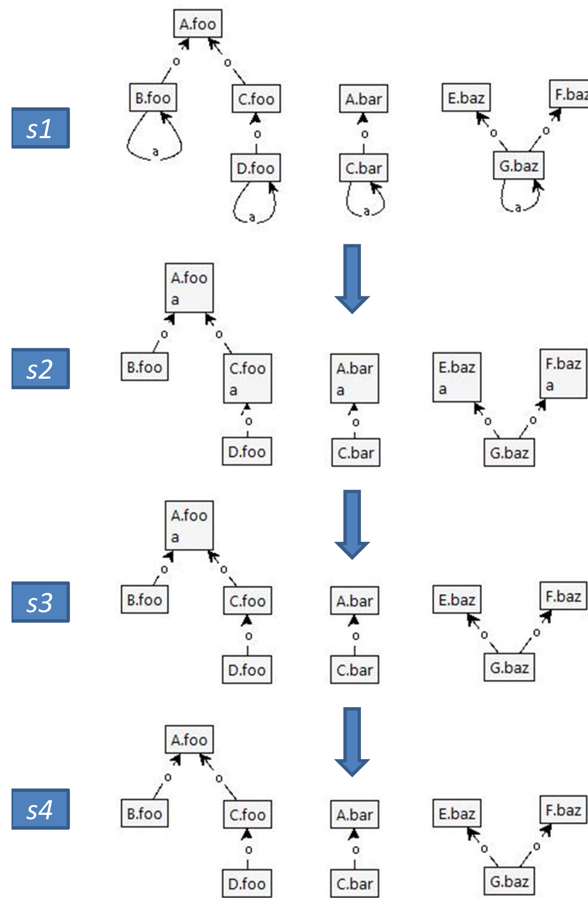

Figure 6.38: Sequence of states of the rule hierarchy graph.

the corresponding node carries an activating edge. It then deletes all activating edges and creates new edges on the nodes corresponding to the most specialized rules (as the right part of rule action.execute(ActivityExecution) in Fig. 6.35 does).

Obviously, to implement soft rule overriding within our example, we need a helper rule to move the a edges. We do not show this rule, since it is very similar to the rule shown in Fig. 6.36 (only the labels have been changed as described above). The sequence of state graphs in Fig. 6.38 shows the effect of the helper rule: With each rule application, all nodes having an activated predecessor node are themselves activated by means of an a edge, and all existing a edges are deleted. Please note that GROOVE draws self edges of nodes as node labels (states 2 to 4).

With this knowledge, we are ready to discuss the transition system depicted as Fig. 6.39. Its purpose is to show the activated rules in each state of the rule hierarchy graph. For this, we have adjusted the rule priority of the helper rule to be equal to those of the other DMM rules (recall from Sect. 6.3.5 that the helper rule normally has lower priority than the "normal" DMM rules, making sure that the rules on the next level are activated only if no smallstep rule of the current level matches).

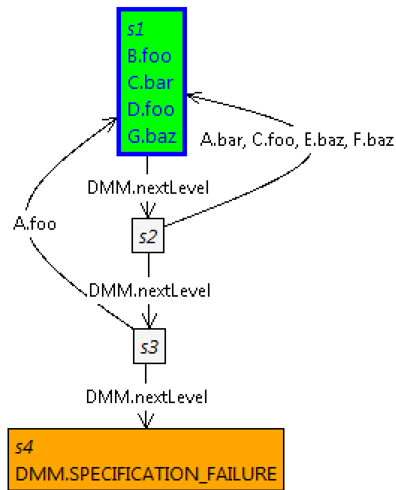

Figure 6.39: Labeled transition system showing the matching of the rules.

The first state is state `s1` at the top of Fig. 6.39. Besides the `s1` label, the state carries four more labels referring to rules. These labels are in fact self transitions: We have seen that the application of one of our example rules deletes all a edges and creates a edges at the most specialized rules. Therefore, rules B.foo() etc. do not change the state at all. They return to the initial state (as all rules in this example do).

Application of the helper rule brings us to state `s2`. In this state, the node corresponding to rule A.foo() is activated. However, the node of rule C.foo() is also activated. Now we can explain the reason for the NAC at the bottom of the `action.execute(ActivityExecution)` rule in Fig. 6.35: It makes sure that the rule only matches if there is no other, (transitively) overriding rule which is still activated. This can be modeled with GROOVE by means of a regular expression about edge labels: label `DMM_overrides+` of rule `action.execute(ActivityExecution)` in Fig. 6.35 means that the NAC is matched if there exists a node which is connected through a positive number of edges whose sequence of labels conform to the regular expression. Therefore, the rule does not match if an activated node is connected through at least one `DMM_overrides` edge. This is the reason why in the transition system, rule A.foo() does not match in state `s2`. However, the other activated rules do match; application of the rules again brings us back to the initial state.

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

6.3.5.8 Applications of Rule Overriding

In the last sections, we have seen by means of simple examples how rule overriding has been integrated into the DMM framework. In this section, we want to point to some more sophisticated uses of rule overriding we have performed so far: We have implemented the semantics of Activities [105], State machines [149], and Interactions [180]. While creating the specifications, we have used rule overriding at two places:

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

6.3.6 Restrictions

In the previous sections, we have seen how Ecore language elements are translated into GROOVE constructs, and how DMM rules are typed over Ecore metamodels. However, there are a number of Ecore features which are not yet supported by DMM.

First, attributes are only supported if they have cardinality 0..1 or 1. In other words, an `EAttribute a` such that `a.isMany()` is `true` will be ignored during transforming a DMM ruleset or Ecore model into a GROOVE grammar or start graph.

The next restriction concerns `EReferences` with cardinality greater than 1. Such references are generally supported (as we have seen in Sect. 6.3.2.3). However, there is another limitation: Ordered references are only supported in a limited way. The current treatment of ordered references provides the following functionality:

- The order of objects in an ordered reference will be preserved, i.e., if an Ecore model is transformed into a GROOVE state and back again, the order of objects in the references of the the resulting Ecore model will be the same as in the source Ecore model.
- When an object is deleted from an ordered reference, the ordering structure is preserved.
- Objects added to an ordered reference are appended to the end of that reference.

However, there is currently no way to manipulate an ordered reference, e.g. by inserting an object at a certain position, or by changing the position of

an object within an ordered reference. Additionally, the support for ordered references introduces quite a number of graph elements on the GROOVE side, i.e., the size of the rule graphs as well as state graphs is increased. Therefore, the support for ordered references is disabled by default in the DMM transformation, resulting in ordered references being treated as unordered ones. For the same reason, we have not detailed the treatment of ordered references in Sect. 6.3.

Finally, a number of more sophisticated Ecore language elements is not supported at all, mostly since the concepts do not make sense in the context of DMM specifications.

- No support for `EOperations` is provided. In Ecore, these can be used to define the signatures of operations which will then be generated by EMF, but have to be implemented manually[9]. However, behavior is described by rules in the case of DMM.
- Several datatypes provided by Ecore are not supported, in particular `EByteArray`, `EEList`, `EFeatureMap`, `EFeatureMapEntry`, `EJavaClass`, `EJavaObject`, `EMap`, `EResource`, `EResourceSet`, and finally, `ETreeIterator`. Metamodels making use of these datatypes can be used with DMM, but the according attributes can not be referenced (they will be ignored by the transformations).
- With version 2.3 of EMF, the metaclass `EGenericType` has been introduced for the sake of supporting Java generics (see e.g. [148]). Since there is no equivalent concept to generics in DMM, `EGenericTypes` are ignored during the DMM transformations.

6.4 Related Work

In this section, we present related work to DMM and semantics specification in general and to our approach of rule overriding in particular. For the former, one important approach to defining a language's semantics is *code generation*, where a code generator receives a model or a set of models as input and generates executable code in a certain programming language. For instance, the MDD tool *Enterprise Architect* [193] includes several code templates for UML constructs such as `Behavior`, `Action`, or `Trigger` which are then instantiated according to the given UML model, resulting in executable code. Other examples include [1, 140, 161].

DMM differs from such approaches in that it focuses on the analysis of behavioral models at design time. However, it could be used in conjunction with code generation approaches, using e.g. techniques as [179] for ensuring equivalence of model and code behavior. This would allow to formally reason about the models' behavioral quality at design time and efficient execution at runtime.

For the formalization of a language's semantics, many different approaches exist, some of them focusing on particular problems (and thus usually being quite specialized), some of them trying to provide more general applicability. Hausmann [96] and especially O'Keefe [160] have investigated different ways

[9] Alternatively, one can use OCL [155] or the recently developed Xcore [53] to specify the operations' behavior.

to provide behavioral semantics for metamodel-based languages such as the UML. O'Keefe concludes that graph transformations are the most appropriate target formalism for this task, and emphasizes DMM as one promising candidate because of its formal, but still understandable nature. We do not repeat these discussions here.

Instead, we focus on a few approaches which are particularly comparable to DMM in that they are also formally based on graph transformations. Where appropriate, we will point out further related work in the appropriate sections later in this thesis.

Fujaba (From UML to Java and back again) [150] has started as a software forward and reverse engineering tool, but has evolved into an extensible platform for software engineering researchers. Several plug-ins add functionality such as MOF-based metamodeling, reverse engineering with detection of design patterns and anti-patterns [211], architectual reengineering [208], and model transformation by means of triple-graph grammars [88]. One particular focus of the Fujaba community have been real-time systems [25].

Fujaba allows to model a software system by means of class diagrams and a special notion of activity diagrams called *story diagrams*; the latter are capable of modifying models and thus represent the system's behavior, and their semantics is defined by means of graph transformations. From a system model, an executable prototype implementation of the system can be generated.

Fujaba has many similarities with DMM since it allows to modify instances of metamodels by means of (graph transformation based) story diagrams. However, the focus of Fujaba is code generation, whereas DMM's main goal is model analysis, e.g. by means of state space exploration.

AGG [199] is a graph transformation toolset implementing the algebraic approach to graph transformation. AGG graph grammars can be attributed with Java objects and Java code, allowing to modify Java data structures; as such, AGG can be used within high-level Java applications as a graph transformation engine.

The algebraic approach to graph transformation allows to apply the theory of *category theory* to graph grammars. AGG makes use of this fact by offering sophisticated, theoretically backed analysis options for graph grammars, e.g. critical pair analysis. This allows to analyze graph grammars for confluence [98] or termination [56].

Eclipse Henshin [3] provides a model transformation language and tool support based on the Eclipse platform. The transformation language is once more visual, and its semantics is backed by graph transformations. Henshin focuses on model transformations, but has an option to explore a Henshin transformation's state space with model checking techniques. As such, it could likely have served as DMM's graph transformation engine (and thus replaced GROOVE) due to its comparable functionality. However, Eclipse Henshin was not yet available when selecting the engine.

Next, we will discuss work related rule overriding, i.e., we will investigate work which provides some support for reusability, such as modularization, prioritization, or support for inheritance. The first area of interest we discuss is notions of inheritance.

In [38], de Lara et al. show how to integrate attributed graph transformations with node type inheritance, therefore allowing to formulate *abstract* graph

transformation rules (i.e., rules which contain abstract nodes). The resulting specifications tend to be more compact, since a rule containing abstract nodes might replace several rules which would otherwise have to be defined for each of the concrete subtypes. The resulting formalism does not provide support for the refinement of rules (and is therefore comparable with the expressiveness of the state of DMM as presented by Hausmann [96]).

Ferreira et al. [71, 132] develop a notion of typed graph transformations which supports several object-oriented features, including inheritance and polymorphism. They focus on delivering a framework which is as close to object-oriented systems described by e.g. Java code as possible, whereas the models targeted by DMM are expected to have a less complex semantics, since they usually will be abstractions of concrete systems described by code. As such, DMM emphasizes on keeping the specification language simple.

Another tool to help with reusing existing semantics specifications is prioritization, since it gives additional control over the application of rules. The AGG toolset [199] supports the concept of *rule layers*: First, all (matching) rules of layer 0 will be applied followed by all rules of layer 1 and so on. This allows the implementation of a simple control flow of graph transformation. This mechanism could probably be used to realize the concept of rule overriding, but this has to be done “by hand”, i.e., the modeler has to manually add according structures to the rules (which finally result in the desired overriding behavior). This is not necessary for DMM, since that structure will automatically be built during the transformation process of a DMM specification into a GROOVE grammar (as we have seen in Sect. 6.3.5).

Another area of related work is modularization. In [123], Kreowski et al. introduce the concept of *graph transformation units* (GTUs) as a way to structure large graph transformation systems. In a nutshell, a GTU consists of a set of graph transformation rules, an optional import of other GTUs, and control conditions. Its semantics is defined by means of the interleaving semantics of all (own and imported) graph transformation rules. The control condition is used to reduce nondeterminism of the resulting graph transformation system, since only rule applications consistent with that condition are considered. GTUs obviously are a powerful means to define the semantics of modeling languages in a reusable way; however, we believe that in comparison to DMM, a considerable larger amount of knowledge of the language engineer is needed to benefit from its expressiveness, partly because the control conditions have to be delivered in addition to the rules.

A completely different approach has been suggested by Legros et al. [129]: They introduce *generic and reflective graph transformation rules* into the Fujaba/MOFLON [2] framework. These rules can combine elements of the model, the metamodel, and the metametamodel level, allowing to define rules which reflectively inspect a corresponding graph and act with respect to the available meta-information. The approach does not support reusability explicitly. However, it might give rise to specifications which are so general that modifying them is not even needed, although this was not the intention of the authors and is most likely not always possible.

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

For instance, ATL [110] and ETL [120] allow to define model transforma-

tions based on the source and target language's metamodels; they are both very similar to the *Relations* part of *Query/View/Transformation* (QVT) specification [156] provided by the OMG. Each rule describes the transformation of a particular language element.

In the context of rule overriding, the most interesting feature of both languages is that a rule can *extend* another rule. However, in the case of ATL, the rule extension mechanism is a static one: The language's compiler makes sure that the precondition of an extending rule becomes the union of the preconditions of all (transitively) extended rules, and that the extending rule's behavior is an aggregation of all extended rules' behaviors. ETL works very similar to ATL with respect to the rule extension semantics. Note that the semantics of the QVT rule extension mechanism is not precisely defined by the QVT specification (see e.g. [86]).

Summary of Part II

In this chapter, we have introduced our re-defined notion of Dynamic Meta Modeling called DMM++. The main differences to the original DMM definition by Hausmann [96] are that we defined DMM's syntax by means of a metamodel and OCL constraints and its semantics by means of a translation into GROOVE grammars, allowing for better tool support and therefore for the practical application of the DMM approach. Additionally, we have improved and extended the existing DMM language, in particular by making attribute support much more powerful, enhancing the DMM mechanism for specifying universal quantification, and by introducing language elements which allow a DMM rule to override another DMM rule.

The syntax of DMM has been defined in Sect. 6.2; we have provided a documentation following the style of the UML specification, explaining each metaclass and providing its static semantics through OCL expressions. In particular, we have introduced new language constructs such as the attribute expression language, which allows to formulate rule application conditions against attribute values as well as manipulating those attributes, and we have introduced metaclasses for modeling the rule overriding relations mentioned above.

We have then specified the semantics of DMM by describing the transformation of arbitrary EMF models and DMM specifications into GROOVE grammars in Sect. 6.3. Since the resulting GROOVE grammars will only work if the transformation of models and DMM specifications is consistent, we have explained the two transformations in parallel where appropriate. In particular, we have put the focus of our explanations on new language features such as attribute handling and rule overriding.

The results of this chapter are

- an extension of DMM which allows to specify rule overriding and improves reusability of DMM specifications,
- an Ecore metamodel of the DMM language defining its syntax,
- OCL constraints to be evaluated against instances of the DMM metamodel and specifying the DMM language's static semantics,
- and a Java-based transformation of Ecore models and DMM specifications into GROOVE grammars, specifying the DMM language's semantics.

This foundation allows to implement tool support for systematically creating DMM specifications, as we will see in the next part, and the formulation and analysis of functional as well as non-functional requirements, as we will see in Part IV of this thesis.

Part III

Quality of DMM++ Specifications

7 Creating DMM Specifications

Obviously, the first task towards using Dynamic Meta Modeling is to create a DMM semantics specification for the language at hand. This chapter will introduce the necessary concepts for this task, which basically consists of three steps:

1. The *runtime metamodel* has to be created from the syntax metamodel.
2. A transformation from instances of the syntax metamodel into instances of the runtime metamodel (which are then ready to be executed by applying DMM rules to them) has to be defined.
3. Finally, the DMM ruleset has to be defined, typing it over the runtime metamodel defined in step 1.

The first step of the above process is motivated by the fact that—as we have explained in Chapter 5 on page 37—for many languages, only the syntax is defined formally; the semantics is then explained by means of natural language. However, our goal is to formalize the language's behavioral semantics, which means that we have to formally introduce runtime concepts allowing for this task.

For instance, in the case of UML activities, we have seen that the semantics is based on *token flow*. Thus, tokens and their locations have to be added to the syntax metamodel. Other languages will require other runtime concepts, e.g., program counters. Therefore, we do not discuss the creation of runtime metamodels here.

The second step results in a transformation which can be used to automatically translate syntax models into runtime models. As such, the tranformation's goal is to make the necessary changes on the syntax model such that it can be executed by means of DMM rules. For instance, in the case of UML activities, an `ActivityExecution` object is added to the syntax model, and that object is then used by the DMM rules to realize the semantics of the language (in this case, the `ActivityExecution` will store the tokens which are flowing through an activity as mentioned in the last paragraph). There are two general approaches of defining such a transformation, both having their own advantages and disadvantages. Both approaches will be introduced and discussed in Sect. 7.2. Before we do that, we will discuss the usage of DMM for model transformations in Sect. 7.1.

The last step of the above process—creating the actual DMM ruleset—is usually the most complex one. For this, we have developed rich tool support,

which we will show in Sect. 7.3.

Note that the above process will often be performed in an iterative way: The runtime metamodel will be built up step by step until it contains all concepts necessary to express the complete language's semantics (and each step will only introduce concepts needed to realize the currently treated part of the behavioral semantics). Consequently, in each step the transformation from syntax to runtime metamodel is refined to take the newly introduced runtime concepts into account, and the rules realizing the semantics part are created. In fact, we will see such an iteative process of creating a DMM specification in Chapter 8 on page 147.

7.1 DMM and Model Transformations

As stated above, the second step of creating a DMM semantics specification is to define a *model transformation* which translates instances of the syntax metamodel in instances of the runtime metamodel (which can the be executed by DMM rules). But what is a model transformation? An (in our opinion) appropriate definition is provided by Kleppe et al. [118, p. 24]:

> A transformation is the automatic generation of a target model from a source model, according to a transformation definition. A transformation definition is a set of transformation rules that together describe how a model in the source language can be transformed into a model in the target language. A transformation rule is a description of how one or more constructs in the source language can be transformed into one or more constructs in the target language.

Note that the above definition does not express any contraints on source and target language. In fact, one way to categorize model transformations is to distinguish between *endogenous* and *exogenous* transformations: The former refers to transformations where source and target language are the same, the latter to transformations where the languages differ.[1]

We immediately notice that the above definition uses terminology which is closely related to DMM's terminology: a "set of transformation rules" is comparable to a DMM ruleset, and "transformation rule" to a DMM rule. In fact, we will now see that DMM can be used to specify model transformations in a simple way. The idea is to create a set of DMM rules which receive the input model as input; the rules will then be applied one after the other until no more DMM rule matches the current model state. The final model state represents the result of the transformation.

DMM can be used to specify endogenous as well as exogenous model transformations: For the former, the transforming DMM ruleset is (only) typed over the metamodel of the one language; the latter implies that the DMM ruleset is typed over both metamodels of source and target language, meaning that rules of such a DMM ruleset can contain objects from both metamodels. Another difference is that in the case of exogenous model transformations, the input model is usually not needed any more. In this case, the transforming DMM ruleset will contain a number of rules which—after creation of the target model—will

[1] See [34] and [139] for classifications of different model transformation types.

remove all language elements of the input language from the resulting model state.

However, there is an important difference between using DMM for semantics specification and for model transformation: In the former case, one is interested in the complete semantics of a model. Thus, the complete transition system will be created, which contains all possible executions of the model at hand, and that transition system will be subject to verification by model checking. In the latter case, we are not interested in the transition system, but in the result of the model transformation.

This implies that a DMM ruleset used for model transformation needs to ensure that a) the resulting transition system contains a final state, and that b) we receive one and only one final state. Otherwise, which state would represent the transformation's result?

In other words, a DMM ruleset realizing a model transformation needs to fulfill two requirements: *termination* [207] and *confluence* [98].

Termination refers to the fact that for each valid input model (i.e., each model being consistent with its syntax and static semantics definition), applying the rules will result in a final state, i.e., a state where no more rules match. Additionally, the set of rules needs to be confluent, meaning that the order of application of the ruleset's rules does not matter; the transformation will always result in the same single final state.

For a DMM ruleset fulfilling these two requirements, one thus does not have to compute the complete transition system implied by the ruleset and the input model. Instead, it suffices to apply one matching rule to each state reached during the model transformation. The result is that using DMM for model transformations does not suffer from the *state explosion problem* as introduced in Chapter 4.

However, there are many dedicated languages for specifying model transformations (see Sect. 7.4). Thus, why should we use DMM for such tasks? The answer depends on the concrete model transformation to be realized. In the next section, we will see why DMM is a good solution for specifying a model transformation from the syntax to runtime metamodel.

7.2 From Syntax Metamodel to Runtime Metamodel

We have seen earlier that the syntax definition of many languages does not allow to express runtime states, and that DMM copes with this situation by introducing a so-called *runtime metamodel*: States of execution of a model are instances of the model's runtime metamodel. In the case of UML activities, the runtime metamodel contains concepts such as `Tokens`, and a state of execution of an activity is determined by the location(s) of the activity's token(s).

Therefore, the syntactical representation of models needs to be translated into an instance of the language's runtime metamodel. In this section, we will introduce two approaches for defining a runtime metamodel, und for deriving an executable transformation which takes a syntax model as input and outputs the according runtime model. In other words: we will see in this section how the *semantic mapping* seen in the overview figure 5.1 on page 39 will be realized technically.

The most important criterion for choosing one of the approaches is the similarity between the syntax and runtime metamodels.

Section 7.2.1 introduces the so-called *"From Scratch"* approach: The runtime metamodel will be built completely from scratch and does not contain any elements of the syntax metamodel, but elements which represent syntax metamodel elements. This has the advantage that all information irrelevant for executing a model will not be contained in the runtime states of that model. This approach has been applied by Hausmann in [96]; here, we show how this approach can be supported by automatically creating a base model transformation which only has to be refined by the language engineer.

In contrast, the *"Decorator"* approach we will introduce in Sect. 7.2.2 directly refers to the syntax metamodel and only enhances it with runtime information. Here, the main advantage is that the actual model does not need to be changed; instead, it will be "embedded" into a decorating part which contains all the runtime information. As we will see in Chapter 10, this allows to re-use existing visual editors for the sake of animating a model's behavior.

7.2.1 The "From Scratch" Approach

Sometimes only a part of a language's syntax is relevant for the language's behavioral semantics. For instance, in the case of the UML, the metamodel contains all classes of the complete UML.[2] However, the runtime metamodel for executing UML activities neither needs to contain syntax elements of, say, state machines, nor should it: Since DMM rulesets can be imported into other DMM rulesets (see Sect. 6.2.1.2), it has several advantages to maintain the semantics of each sub language in its own ruleset, e.g., easier maintainability and improved scalability (the latter because if only a UML activity shall be executed—and not a complex UML model containing different kinds of behavioral models which even might invoke each others behavior—, the graph transformation engine has to consider less rules for each state of the resulting transition system).

As such, Hausmann [96] suggested to create a runtime metamodel from scratch which only contains the language elements relevant for the language's behavioral semantics, and to provide a mapping between syntax and runtime metamodel by means of meta relations (see Sect. 5.2.1.2).

However, this has an important implication: To execute a model, it has to be translated into an instance of the runtime metamodel. Unfortunately, we have seen in Sect. 5.2.1.2 that meta relations are not executable. Thus, the language engineer needs to come up with a model transformation for this task.

The created runtime metamodel will most likely be smaller than the syntax metamodel, but it will still contain many elements which directly represent elements of the language metamodel. This fact can be used as follows: Given a syntax metamodel, we automatically generate a runtime metamodel which is an exact copy of the syntax metamodel (except the root package's name, which needs to be unique to be able to distinguish between the metamodels). Additionally, we generate a set of DMM rules which translate each syntax metamodel instance into the equivalent runtime metamodel instance, i.e., an exact copy of the syntax model.

[2] Of course, the scope of the UML can be narrowed, e.g. by means of compliance levels [158, p. 2] or the usage of UML profiles [158, p. 669].

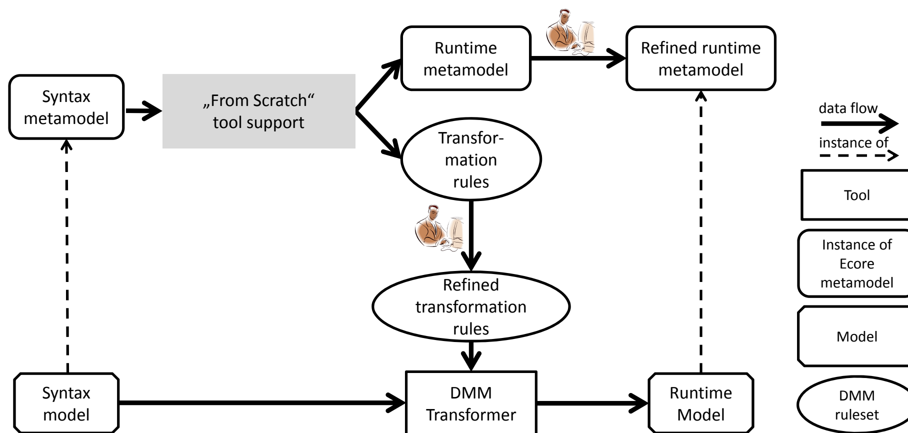

Figure 7.1: Illustration of the creation of runtime metamodel and transformation in the "From Scratch" approach.

The language engineer's job is then reduced to refining these two artifacts, e.g. by removing metaclasses irrelevant for the semantics, and adding metaclasses needed to realize the language's semantics (such as a `Token` class). In parallel, the language engineer refines the generated ruleset.

The result will be a DMM ruleset which serves as a model transformation as follows: The transition system will be generated using a given syntax metamodel instance as start state. However, the transition system will be explored linearly as discussed earlier: For each state, only one outgoing transition will actually be followed. The final state of this linear, labeled transition system then is an instance of the runtime metamodel. It serves as start state of computation of the actual semantics of the model by means of the DMM semantics specification typed over the runtime metamodel.

An overview of the process is illustrated in Fig. 7.1. A syntax metamodel serves as input for the support framework, which copies the metamodel and generates the "copying" DMM transformation rules. The runtime metamodel as well as the transformation rules can then be customized by the language engineer. The final transformation rules serve—together with an instance model—as input for the DMM transformator which will create the according runtime model.

Note that the human intervention by means of adjusting the generated metamodel and DMM rules is only needed once – afterwards, the DMM transformator at the bottom can be executed for every possible input syntax model.

The copying of the metamodel is done using EMF Java API and is thus not explained any more. In the next section, we investigate the generation rules which create the DMM transformation rules to be modified by the language engineer. This section is based on the master's thesis of Hendrik Schreiber [183].

7.2.1.1 Generating the DMM Transformation Rules

As mentioned in the last section, the DMM "From scratch" tool support generates a set of DMM rules for transforming syntax models into runtime models.

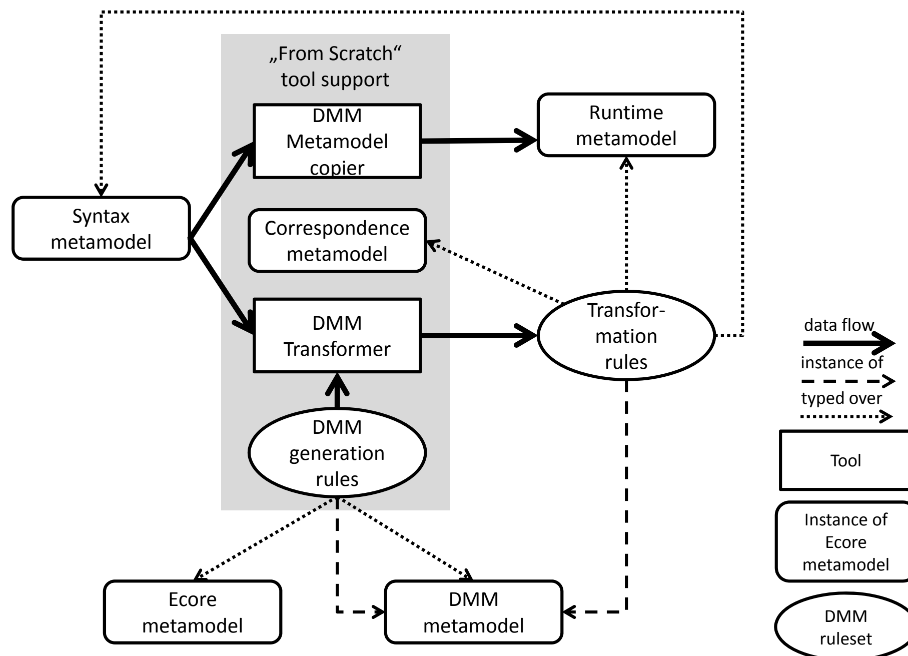

Figure 7.2: "From Scratch" approach: Detailed view of tool support and involved artifacts.

The generation is itself performed by a set of DMM rules. Fig. 7.2 shows a detailed view of the metamodels and rulesets involved in this task.

The DMM Transformer gets the generation rules and a syntax metamodel as input and generates the transformation rules. To realize this exogenous transformation, the generation rules are typed over the Ecore metamodel as well as the DMM metamodel. Additionally, the "From scratch" support includes a so-called *correspondence metamodel*, which will be used by the generated transformation rules to create correspondences between syntax and runtime model elements. Therefore, the transformation rules (which again realize an exogenous model transformation) are typed over three metamodels: The syntax metamodel, the (generated) runtime metamodel, and that correspondence metamodel.

With this brief idea of how the transformation generation works, let us now investigate the generation ruleset in more detail, an overview of which is depicted as Fig. 7.3.

The generation has two major parts: the actual rule generation and a clean up phase. The latter is performed by the rules epackage.DestroyRuntimeMetamodelEPackage()#, epackage.DestroyMetamodelEPackage()#, and ruleset.DestroyEFactory()#, which remove the artifacts not needed in the generated ruleset.

The interesting part is the actual generation, which is started by rule epackage.EPackage2BigstepRule()# depicted as Fig. 7.4. The rule has the syntax and runtime metamodels as `EPackages` as prerequisite. It creates the `Ruleset` with a single `Package` which will contain the generated rules. Additionally, it creates a reference to a *correspondence metamodel* which will be used in the transformation to keep track of the correspondences between the elements of

Figure 7.3: Overview of the transformation generation ruleset.

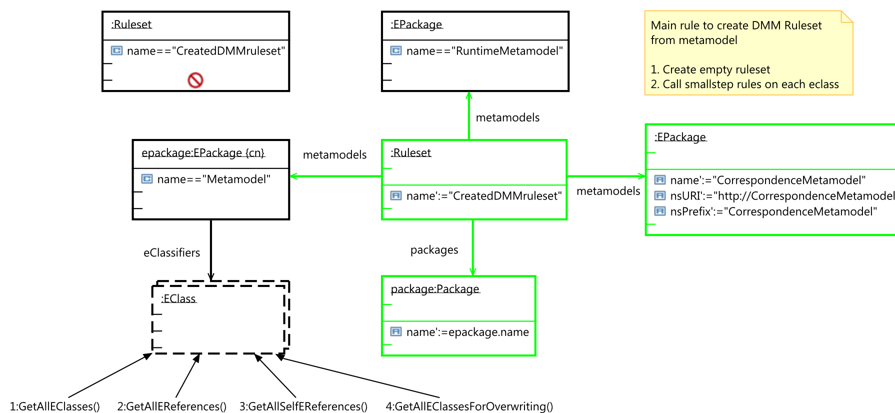

Figure 7.4: Main transformation generation rule.

syntax and runtime model.

The rule then starts the generation of the rules which will copy the elements of the syntax model into the runtime model. For each `EClass`, four steps are performed:

1. A bigstep rule is generated which copies elements of the `EClass`'s type.
2. A bigstep rule is created for each outgoing `EReference` of the `EClass` which is not a self-reference.
3. A bigstep rule is created for each outgoing `EReference` of the `EClass` which is a self-reference. Self-references need special treatment because of the fact that DMM rules match injectively – thus, the rules generated in step 2 can not take care of objects referencing themselves.
4. The rules generated so far have a problem: A rule which is targeted for a

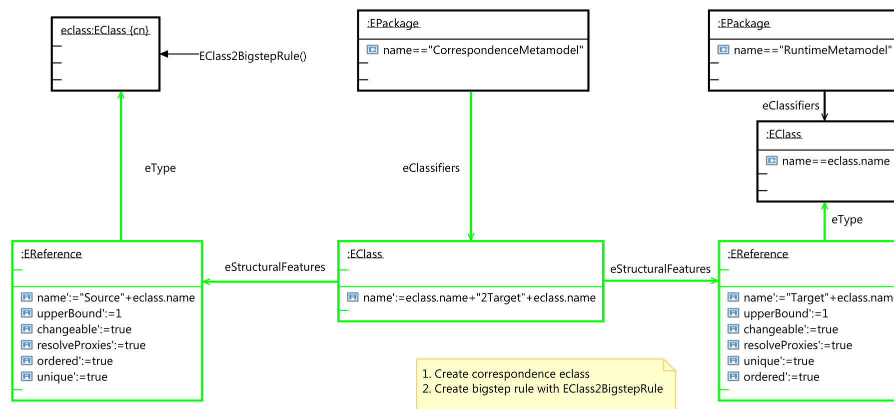

Figure 7.5: Transformation generation rule creating correspondence structure of an EClass.

metaclass A will also match for all subclasses of A. Therefore, in the last step rule overriding relations will be inserted into the ruleset which make sure that each rule only matches in the context of its particular type.

As an example, let us investigate rule eclass.GetAllEClasses() which is depicted as Fig.7.5. The rule creates the correspondence structure between the EClasses of syntax and runtime metamodel and then delegates the rule creation to rule eclass.EClass2BigstepRule().[3]

Note that the correspondence metamodel is created using Ecpre: The source and target eclasses' (instances of which are to be copied) are connected by the EClass at the bottom and to EReference instances; the name of the EClass will later be used for finding the right correspondence.

Finally, rule eclass.EClass2BigstepRule(), which is depicted as Fig. 7.6, creates the actual bigstep rule which will copy elements of the rule's type into the runtime model. The creation of the bigstep rule can be seen at the top of the rule; note that we have hidden the references between the Rule node and the rule's Node and *Edge* nodes to not clutter the figure. The actual node is created to the top right of the rule – its type is set as the EClass of the runtime metamodel which corresponds to the syntax metamodel's EClass. Additionally, the rule creates a correspondence structure which will during transformation be used to identify the target node of an EReference within the runtime model. Finally, the rule adds a negative application condition to the generated rule which makes sure that during transformation, the rule can only be applied once per object of the syntax model.

Note the treatment of correspondence at the bottom of the rule: the correct correspondence class is found using a condition over the class's name. The nodes and edges of the created rule reference the correspondence parts.

As soon as all invocations of rule eclass.GetAllEClasses() are finished and one copy rule has been generated per EClass f the syntax metamodel, the generation continues with the rules for copying EReferences, which can now

[3]Note that most of the above rules are separated into the two phases of creating a correspondence structure and creating the actual target elements.

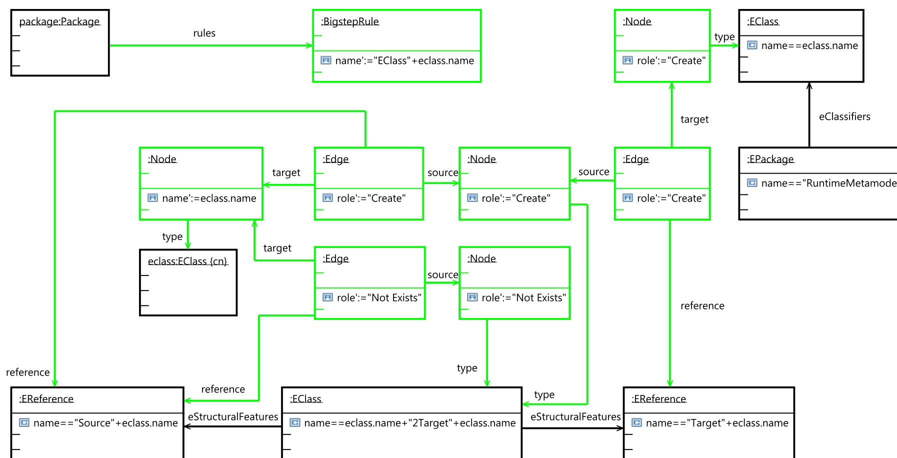

Figure 7.6: Transformation generation rule creating the rule for copying elements of type `EClass`.

rely on the fact that for each object of the syntax model, a corresponding object has been created in the runtime model.

We do not show the other rules of the generation ruleset here – the interested reader can study the complete generation ruleset within plug-in `de.upb.dmm.transformation.metamodel2runtime`. Instead, we show some generated transformation rules within the next section as an example.

7.2.1.2 Example DMM Model Transformation

As an example for generating a DMM model transformation, let us again consider the simplified version of UML activities we had introduced in Sect. 6.3.5 on page 105, the metamodel of which is depicted as Fig. 6.27 on page 105. The language contains a subset of the language elements of UML activities: It does not distinguish between `ControlFlows` and `ObjectFlows`, and it only supports a small subset of the UML's `ActivityNodes`, i.e., the subclasses contained in the UML's packages `FundamentalActivities`, `BasicActivities`, and `IntermediateActivities`.

Feeding the language's metamodel and the generation rules we have seen in the last section into the DMM "From Scratch" support tooling produces two results: A copy of the syntax metamodel and a DMM ruleset which copies instances of that metamodel into instances of the metamodel's copy. Let us first investigate the generated ruleset in more detail; it is depicted as Fig. 7.7.

The ruleset's single package contains a number of bigstep rules, some of which completely overriding each other. Each of these rules copies a certain language element: rules with a signature of the form *.EClass* copy objects of the `EClass`'s type, and the other rules copy references.

Fig. 7.8 shows the rule Activity.EClassActivity()#, which copies objects of type `Activity`. For each `Activity` of the syntax model, an `Activity` is created in the runtime model. Additionally, a correspondence node is created which keeps track of the correspondence between the two `Activity` nodes – this is an instance of the according correspondence metaclass and thus has a descrip-

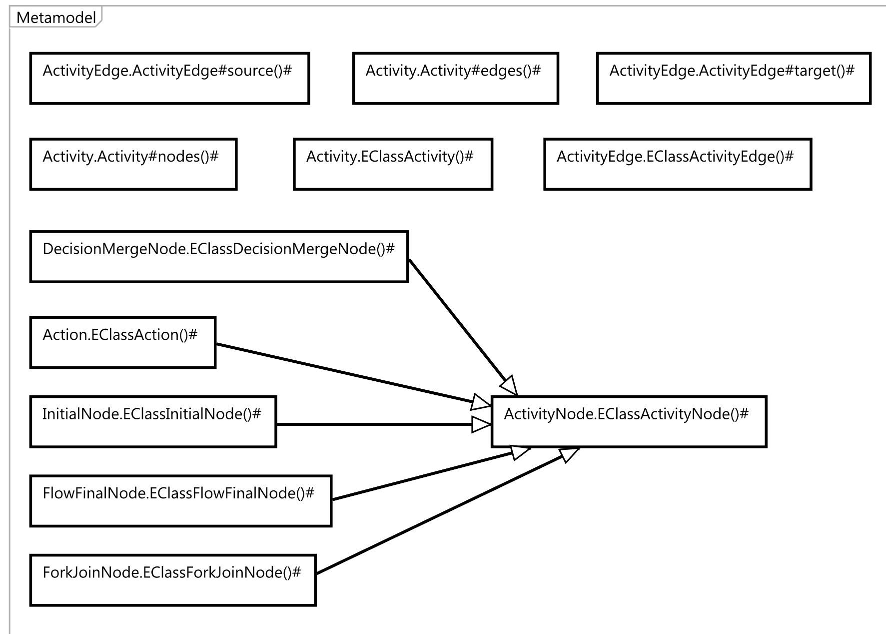

Figure 7.7: Generated ruleset for transforming syntax models into runtime models.

tive type name. Finally, the negative application condition makes sure that the rule can only match once for each `Activity` object. Note that due to a restriction of the visual DMM editor, the two `Activity` types can not be visually distinguished in the rule view, but the nodes are indeed typed over the two different `Activity` metaclasses of syntax and runtime metamodel.

The correspondence structure built by the object-copying rules is then used in the reference-copying rules to make sure that the right objects are connected via the references. Let us investigate rule Activity.Activity#nodes()# as an example, which is depicted as Fig. 7.9. It matches for each `nodes` reference between an `Activity` and an `ActivityNode`. The correspondence nodes in the middle of the rule make sure that a) the rule does not match before the `Activity`'s as well as the `ActivityNode`'s objects have been copied, and b) that the `nodes` reference is created between the right object copies. Again, the negative application condition makes sure that each `nodes` reference is copied exactly once.

Now, the language engineer has to perform three tasks:

1. Refine the runtime metamodel as desired, e.g., add runtime elements and remove elements not relevant for the behavioral semantics.

2. Refine the generated transformation ruleset, e.g., add runtime elements to the according rules or remove rules which deal with elements removed in step 1.

3. Create the actual DMM semantics specification realizing the language's semantics.

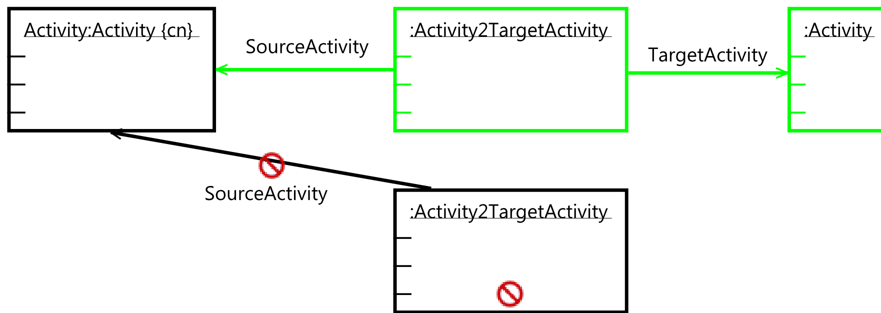

Figure 7.8: Generated rule Activity.EClassActivity()# copying nodes of type `Activity`.

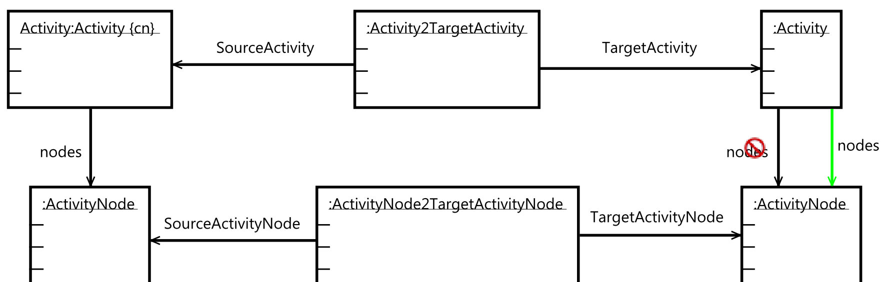

Figure 7.9: Generated rule Activity.Activity#nodes()# copying the `nodes` reference between `Activities` and their `ActivityNodes`.

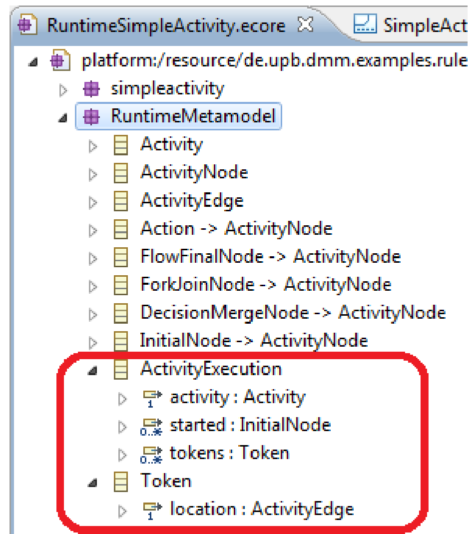

Figure 7.10: Modified runtime metamodel – the runtime part is marked with a red box.

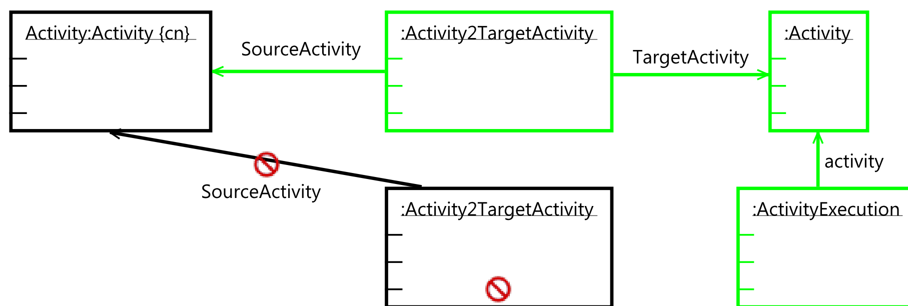

Figure 7.11: Refined rule Activity.EClassActivity()# for metaclass `Activity`.

For our simplified activity language, steps 1 and 2 are straight-forward – the modifications are depicted as Figs. 7.10 and 7.11. The runtime metamodel is enhanced with two metaclasses which can be seen at the bottom of Fig. 7.10: The `Token` metaclass represents the locus of control of an executed `Activity` and is located at an `ActivityEdge`;[4] the `ActivityExecution` owns the `Tokens` flowing through the executed `Activity` and keeps track of the `InitialNodes` which have already produced a `Token` at activity start time.

Finally, the only change which has to be performed on the generated model transformation is on rule Activity.EClassActivity()#. This rule is refined such that for each `Activity` node, an `ActivityExecution` node is created and associated with the `Activity` node. The resulting rule is depicted as Fig. 7.11.

The final transformation ruleset translates instances of the syntax metamodel into instances of the runtime metamodel, adding `ActivityExecu-`

[4]Note that the semantics of the simplified activity language does not support traverse-to-completion (see Sect. 3.3).

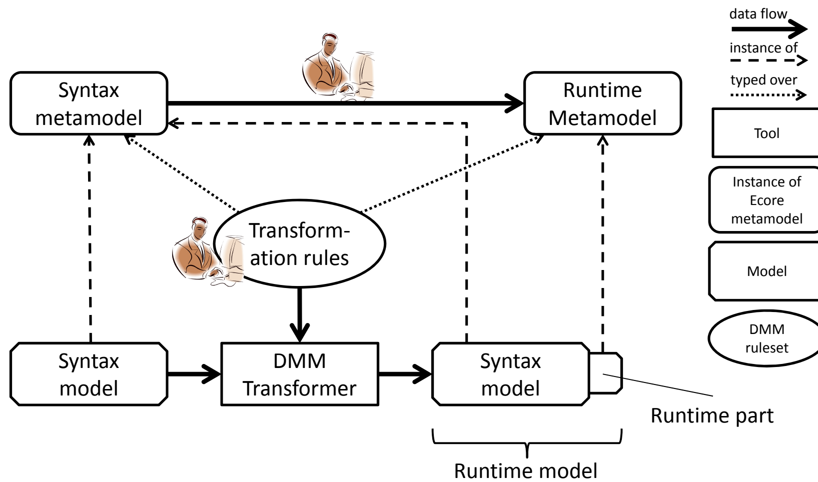

Figure 7.12: Illustration of the creation of runtime metamodel and transformation in the "Decorator" approach.

`tions` which make the models executable by the accompanying semantics specification, as desired. The language engineer's part of creating the transformation ruleset and the metamodel comes down to a minimum and can be performed in a couple of minutes.

We have now seen that the "From Scratch" approach makes creating runtime metamodel and transformation ruleset rather easy. However, replacing the complete syntax metamodel with a different (although rather similar) metamodel has one drawback: Any tooling based on the language's syntax metamodel can not be used for runtime models. In the next section, we will introduce the "Decorator" approach which deals with exactly this problem.

7.2.2 The Decorator Approach

The difference between the syntax and runtime metamodel is—as we have seen above—that the latter contains language elements needed to express states of execution of a model – information which is not contained in the syntax metamodel and has thus to be added.

To achieve this, another approach can be applied: The "Decorator" approach is inspired by the decorator pattern [78, p. 175]. The idea is to store the runtime language elements within their own metamodel, which then reference elements from the syntax metamodel. An overview of the decorator approach is depicted as Fig. 7.12.

The advantage is that the syntax metamodel does not have to be changed at all – thus, tooling based on the syntax metamodel can be used on runtime instances to some extent (we will make use of this in Sect. 10.1). Another benefit is that there is no need to transform the syntax model into a runtime model – only the runtime elements have to be created and configured such that they reference the according elements of the syntax model.

Let us apply the "Decorator" approach to our simplified activity language again, of which we have seen the metamodel in Fig. 6.27 on page 105. The deco-

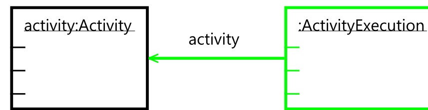

Figure 7.13: The single DMM rule realizing the runtime transformation.

rating runtime metamodel has been depicted as Fig. 6.28 on page 105. The runtime elements `ActivityExecution` and `Token` are contained in the runtime metamodel and reference types from the syntax metamodel (i.e., `Activity` and `InitialNode`). The referenced syntax metamodel elements can be seen at the top of Fig. 6.28.

The transformation which adds runtime elements has to be created completely in the "Decorator" approach, but is of course much easier than in the "From Scratch" approach, since it does not need to take care of the syntax part. In fact, for our example it consists of a single DMM rule which adds the `ActivityExecution` – it is depicted as Fig. 7.13.

One could think that a drawback of the decorator approach is that the DMM ruleset realizing the language's semantics specification now has to be typed over the complete syntax metamodel, instead of a metamodel containing only the semantically relevant language elements as is the case with the "From Scratch" approach. However, the translation of DMM rulesets into GROOVE grammars makes sure that this is not true in general: Elements not relevant for the language's semantics will probably not be referenced by any of the DMM ruleset's rules and will thus be filtered out during translation of models into GROOVE state graphs (and will be re-added when translating graphs back to models), as we have seen in Sect. 6.3.2. The only exception are elements which are not semantically relevant, but still need to be referenced by some semantics rules, for instance because they connect two semantically relevant elements.

Having defined a runtime metamodel as well as a transformation from syntax models to runtime models, the next step is to create the actual DMM ruleset realizing the language's semantics. This is done with help of the DMM Editor, which is presented in the next section.

7.3 Creating DMM Rulesets

One of the main advantages of DMM is that the resulting semantics specifications are formal yet relatively easy to understand, the latter because of their visual appearance which should be familiar to all advanced language users, i.e., users who know the language's metamodel. Obviously, to make use of this advantage, appropriate tool support is needed.

The core of this support is the DMM Editor, a set of Eclipse plug-ins which allow to

- create DMM semantics specifications in a visual, intuitive manner,
- guarantee the syntactical correctness of the created specifications,
- check the static semantics of the specifications, and

- execute DMM functionality such as transforming DMM rulesets and models into GROOVE grammars, performing model checking etc.

An early version of the DMM Editor has been developed by Malte Röhs within [174]. Despite the author's work on the DMM Editor, Eduard Bauer and Nils Bandener have contributed within [16, 11].

The DMM Editor is based on the Eclipse Graphical Modeling Framework (GMF) [50], a framework for generating rich visual editors out of models provided in a framework-specific domain-specific language. The core functionality of the DMM Editor is contained in four plug-ins:

- `de.upb.dmm.editor.diagram.ruleset`
  Generated visual editor for DMM rulesets, allowing to view and edit DMM rulesets on the ruleset level.
- `de.upb.dmm.editor.diagram.rule`
  Generated visual editor for DMM rules, allowing to view and edit a ruleset's rules.
- `de.upb.dmm.editor.diagram.custom`
  Customizations of the generated editors, e.g., visual appearance of some elements, drag and drop functionality etc.
- `de.upb.dmm.editor.parser`
  Support for DMM's expression language (see Sect. 6.2.4).

The visual editors generated by the GMF framework guarantee that the resulting models are syntactically correct with respect to the metamodel of DMM. Additionally, GMF allows to provide constraints for defining the static semantics of a language; these constraints—which are implemented by means of OCL—can be validated against a created DMM ruleset, and violations are communicated by visually annotating elements within the DMM Editor as well as by listing them in Eclipse's problems view. So far, the DMM Editor checks 66 such constraints – DMM rulesets not giving rise to any validation errors are expected to be translatable to GROOVE grammars without problems.

Screenshots of the DMM tooling are shown as Figs. 7.14 and 7.15. The former shows the visual DMM ruleset editor to the right. Left to that editor, the tree-based DMM ruleset editor (generated from the EMF framework and customized) can be seen. To the bottom left, the outline of the ruleset under consideration can be seen – the ruleset being edited is a rather large one with more than 200 rules. Right to the outline, the Eclipse problem view is located and shows a couple of errors and warnings for the edited ruleset - two of these warnings are also visualized in the visual ruleset editor by a yellow warning sign annotating the affected rule. Hovering over the annotation reveals the reason of the warning. Finally, to the bottom right, the Eclipse properties view can be seen which shows detailed (and editable) properties of the currently selected element.

Fig. 7.15 shows the visual DMM rule editor. To its left, the language's runtime metamodel is opened within the Ecore tree editor. The two editors are integrated such that if one drags a metaclass from the Ecore editor and drops it onto the rule editor's canvas, a DMM node will be created and typed over that metaclass, allowing for a simple and quick creation of DMM nodes.

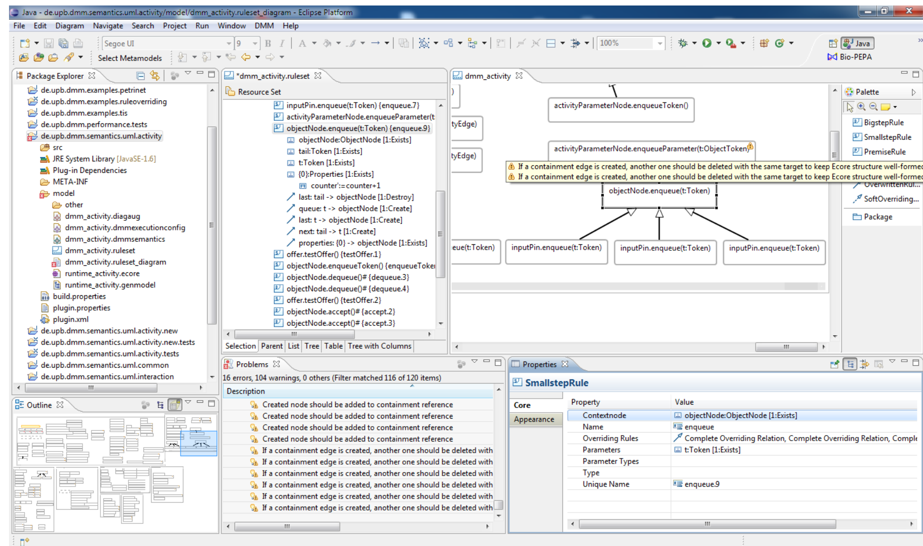

Figure 7.14: DMM Workbench with tree ruleset editor, visual ruleset editor, outline, problems view, and property view.

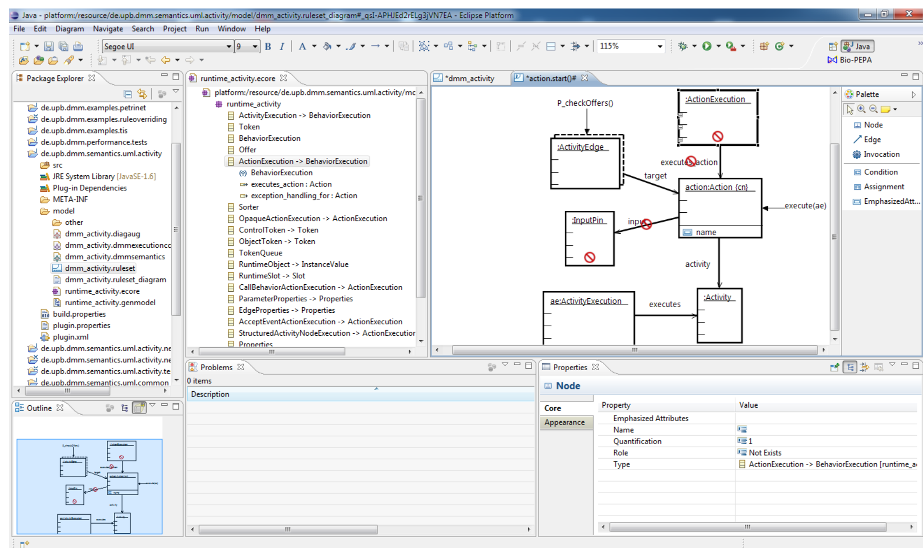

Figure 7.15: DMM Workbench with Ecore metamodel editor, visual rule editor, outline, problems view, and property view.

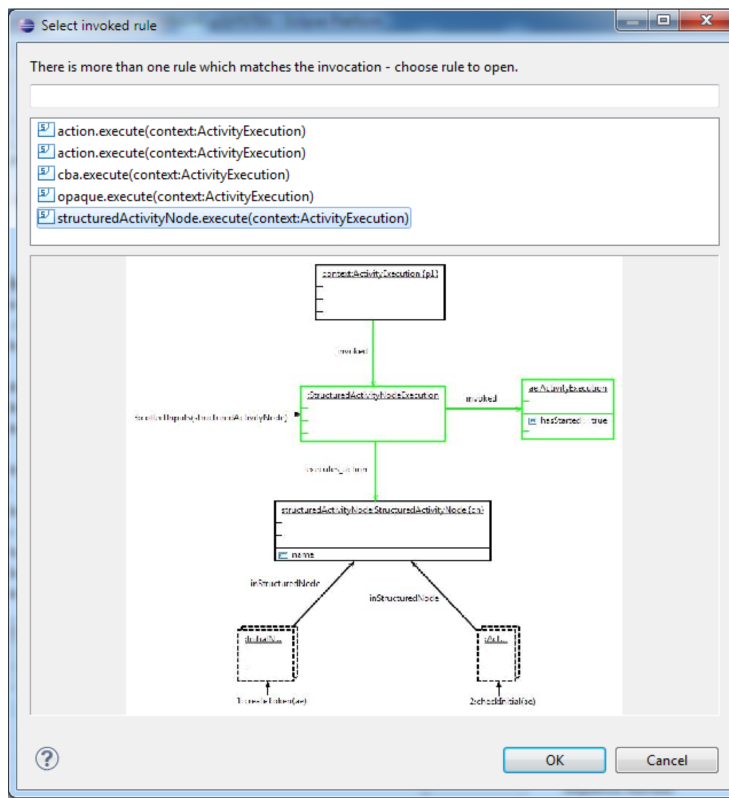

Figure 7.16: Dialog allowing to select one of the potentially invoked rules of an invocation.

The DMM tooling supports the language engineer in creating DMM semantics specifications in several ways: For instance, if an edge is created between two DMM nodes, a dialog opens and offers all available `EReferences` over which that edge can—depending on the types of source and target node—be typed. Another example is the “Open invoked rule” action contained in all invocations’ context menus: Recall from Sect. 6.2 that many smallstep rules may exist which realize a certain invocation; the action will display a dialog showing all smallstep rules which might match if an invocation is performed. It is depicted as Fig. 7.16.

Overall, the DMM tooling has proven to enable the language engineer to create and maintain DMM rulesets. In particular, [105], [180], and [149] would not have been possible without reasonable tool support.

In the remaining thesis, we will point out other features of the DMM tooling where appropriate.

7.4 Related Work

The scientific work related to this chapter can be divided into two categories: Model transformation in general and the automatic generation of model transformations. For the former, a huge number of approaches exist – providing a

comprehensive overview is out of scope of this thesis. Mens et al. [139] present a taxomomy of model transformation approaches; overviews of such approaches can be found in [35] or —more recently—in [135]. Stevens focuses on bidirectional model transformation approaches in [195].

Of particular interest are model transformation approaches which are backed by graph transformations (as DMM is). An overview of different approaches is given in [59]; Ehrig et al. [55] investigate the formal concepts of graph transformation in the context of model transformation and define correctness criteria such as *functional behavior*, which includes termination and confluence.

For algebraic graph transformation approaches, termination is undecidable in general [164]. However, termination criteria have been defined for graph transformation systems. For instance, one can show that the number of certain kinds of nodes and edges decreases with every rule application [7]. A more general approach is followed by Bottoni et al. [23], where *high-level replacement systems* serve as a generalization for graph transformation systems; the authors develop termination criteria for such systems, which are then applicable to graph transformation systems as well.

The graph transformation community has also delivered tools which allow to practically apply the above concepts. For instance, AGG [199] and Eclipse Henshin [3] both allow for the analysis of model transformations with respect to termination and confluence, thus enabling the modeler to prove at design time that her model transformation indeed realizes functional behavior.

In contrast, DMM has not been designed with performing model transformations in mind. Therefore, it does not provide sophisticated analysis techniques as the approaches discussed above. However, it is an interesting research question to which extend such approaches could be transferred into the DMM world.

Finally, we want to mention triple-graph grammars (TGGs) [184, 185], which follow a slightly different approach: triple-graph transformation rules are typed graph transformation rules which are not only typed over the source and target metamodels, but also a correspondence metamodel; instances of that metamodel's metaclasses are used to keep traces between the source model's elements and the elements created on the target side.

Triple-graph grammars have one advantage: They can be applied bidirectionally. As such, triple-graph grammars have been used for model transformation (see e.g. [127, 122, 89]) in general and model synchronization (see e.g. [83, 99]) in particular. DMM does not make use of triple graph grammars; however, the correspondence metamodel used when generating the DMM transformation rules in the From Scratch approach as seen in Sect. 7.2.1 is inspired by the correspondence metamodel used in triple graph grammars.

Let us now consider the generation of model transformations. One scenario that has been investigated is the automatic generation of "converting" model transformations, i.e., model transformations which convert one data model into a different, but similar data model. Roser and Bauer [176] use ontology matching techniques to create mappings between source and target metamodels of the desired transformation, and then automatically generate the transformations. Falleri et al. [70] follow a similar approach: Given a source and target metamodel, they use the *Similarity Flooding algorithm* to compute an alignment model between the two metamodels, from which they generate the final model transformation.

The above approaches are more general to ours in that source and target metamodel can be different (which does not have to be the case in our scenario), but rely on the difficult task of ontology matching and metamodel aligning (consequently, both approaches consider manual intervention during these steps). They could, however, be applied in the DMM setting by using the language's syntax metamodel as both source and target metamodel, and by using identity as the ontology matching/meta model alignment.

A whole class of work on model transformation generation is dedicated to the field of *model transformation by example*. The idea is to provide a couple of transformation samples, i.e., pairs of a model of the source language and the model which shall result from the target model transformation. The samples will then be analyzed, and a model transformation will be generated which performs the necessary transformation automatically.

Kappel et al. [112] provide an overview of current approaches of model transformation by example. Varró et al. [10] use inductive logic programming to gather knowledge about the target transformation, taking the sample models into account. The transformation is then generated as graph transformation rules. In [206], Varró suggests a different approach: He assumes that a prototype mapping model exists (created manually, and pointing out only the most critical relations between elements of source and target metamodel). That mapping model is then analyzed together with the transformation samples, and a model transformation is generated by means of graph transformation rules.

Other approaches do not generate graph transformation rules, but rules of the transformation language *ATL* [110]. For instance, Wimmer et al. [197] provide languages for manually relating source and target metamodels as well as algorithms for deriving an according ATL transformation. Langer et al. [125] show how to generate ATL rules from simple sample models demonstrating the correspondence of source and target metamodel elements.

All above approaches could probably be used for creating a transformation between syntax and runtime metamodels, although they usually need manual intervention when defining the relations between source and target metamodel. Since in our scenario of copying the metamodel, that relation is very simple, the above approaches could probably be tweaked such that no manual intervention is necessary. However, the From Scratch approach still has a minor advantage: The resulting model transformation is specified be means of a DMM ruleset (as is—or will be—the semantics specification of the language). Thus, the language engineer does not have to learn additional languages such as ATL.

8

Test-driven Semantics Specification

Our final goal will be to reason about the quality of models whose language is equipped with a DMM specification. This has an important implication: If the DMM specification itself contains flaws, the analysis results of the models will be pretty much useless. For instance, analysis of a model might reveal that that model's execution gives rise to potential deadlock situations; however, we need to be (reasonably) sure that the cause of those deadlocks lies indeed in the model (and not in the fact that erronously implemented DMM rules cause the deadlock). Therefore, we have developed the approach of *test-driven semantics specification* [191] which will be introduced in Sect. 8.1.

However, even if we develop DMM specifications in a test-driven way, this will only result in higher quality if the tests themselves are reasonable. One way to measure the quality of tests within software development is *test coverage*. In Sect. 8.2 we will present our work [5] on transferring the concept of coverage into the DMM world.

8.1 Test-Driven Semantics Specification

Given a formal specification, the first and obvious idea is to define a notion of correctness by means of requirements the specification shall fulfill, and then to *prove* that this is indeed the case. Unfortunately, the experiences from software development seem to imply that proving the correctness of a reasonable complex system is often just not feasible; as such, the most important technique in software quality assurance is *testing*.

Therefore, we have developed a pragmatic approach to help creating high-quality semantics, which is inspired by the well-known approach of *Test-Driven Development* [19]. This is motivated by the fact that a semantics specification basically follows the Input-Process-Output (IPO) model, where a certain model can be seen as the input, and the semantics of that model is the output (e.g., represented as a transition system).

Figure 8.1 shows our approach and its relation to the testing of software systems. In the latter case, a test case consists of some input for the software system and the system's expected result. The test succeeds if the actual output of the system is equal to the expected result.

In contrast to that, we want to test a semantics specification. Therefore, a test consists of an example model and its expected behavior. From that model and the semantics specification, a transition system can be computed

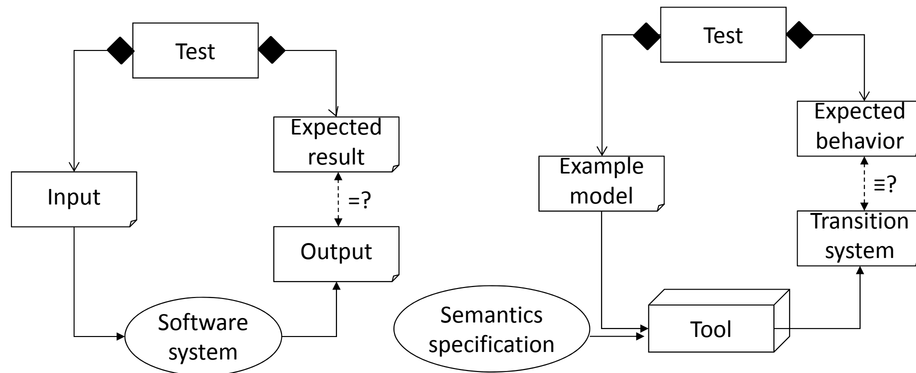

Figure 8.1: Comparison of testing of software systems (left) and semantics specifications (right); the test subject is depicted as an oval.

which represents the model's behavior. The test succeeds if the actual behavior contains the expected behavior (and only that behavior). There is only one requirement on the semantics specification technique used: the behavior of a model must be represented as a transition system which can be model checked for certain *execution events*, i.e., events occuring when a model is executed.

In a nutshell, our approach works as follows: In a first step, a set of example models will be created which demonstrate the constructs of the language under consideration. Additionally, the expected behavior of each example will be identified and fixed in terms of *traces of execution events*. In the second step, the actual semantics specification is performed and tested continously, using a formalization of the traces identified in step 1.

The result is a semantics specification which realizes the expected behavior of the example models. Additionally, the language engineer has a set of examples at hand which can be used e.g. for documentation purposes.

In the following, we will explain our approach of test-driven semantics specification in detail. Section 8.1.1 will show how to systematically create example models, and how to describe the expected behavior of those models in terms of traces of execution events. To ease the creation of the traces, we have developed a DSL which we will introduce in Sect. 8.1.1.5. Section 8.1.2 will then show how to derive test cases from the example models and the associated traces, and how to use these test cases to ensure that the semantics specification indeed works as expected.

8.1.1 Creating Example Models

Obviously, one has has to figure out the exact meaning of the language constructs before their behavior can be formalized. This is where example models come into play: if they are chosen appropriately, they can serve as a good base for discussion of the meaning of the example's language elements.

But what means "appropriate" in this case? The example models should

- concentrate only on a few language elements and their meanings,
- all together cover all elements of the language under consideration, and

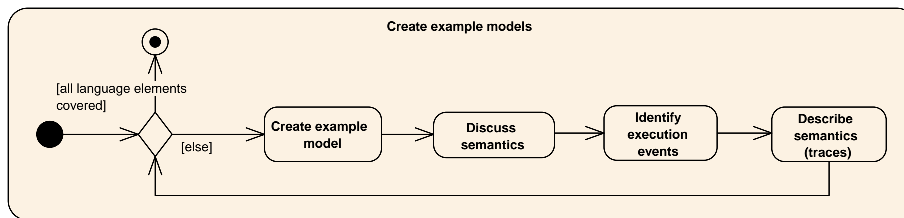

Figure 8.2: Process of creating example models.

- each give rise to a finite transition system.

Section 8.1.2 shows how the last requirement is needed to reuse an example model as a test case. In this section, we will describe how to systematically create appropriate example models, and we will show how to precisely but informally describe their meanings. The steps described within this section are shown in Fig. 8.2.

The starting point is the abstract syntax of the language under consideration. It defines all language elements and their relations with each other. In the case of DMM, the abstract syntax must be given as a metamodel, but other descriptions could be used here, e.g. some kind of grammar. Based on the abstract syntax, the example models should be created step by step, systematically going from the most basic to more complex language constructs[1].

8.1.1.1 A Very Simple Example

Create example model: The very first step is the creation of an example model which should be as simple as possible. Let us investigate this in the case of UML Activities. The UML metamodel is structured into packages which depend on each other, and which indeed start with the most fundamental language elements (contained in the package `FundamentalActivities`) up to the sophisticated language elements contained in package `ExtraStructured-Activities`. This is helpful for our task of systematically creating example models.

In fact, the package `FundamentalActivities` only allows to create Activities containing `Actions` which can be grouped using `ActivityGroups`[2]. Therefore, the first example model we create only contains one `Action`; it is depicted as Fig. 8.3.

Discuss semantics: The next step will be to figure out the supposed behavior of our newly created example. For our simple Activity containing only one `Action`, this is not difficult: the UML specification states that "when an activity starts, a control token is placed at each action or structured node that has no incoming edges" [158, p. 389]. Therefore, if the Activity is started, the only occuring event is that the contained `Action` is executed.

Identify execution events: Now that the semantics of the example model is reasonably clear, we want to describe it precisely. For this, we first have

[1] The example models can of course be created using existing editors and the language's concrete syntax.

[2] Note that according to the UML specification, `ActivityGroups` "have no inherent semantics" and are therefore not used in our examples.

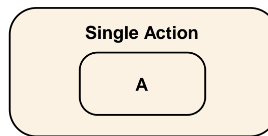

Figure 8.3: Example Activity "Action" containing only one `Action` having name "A".

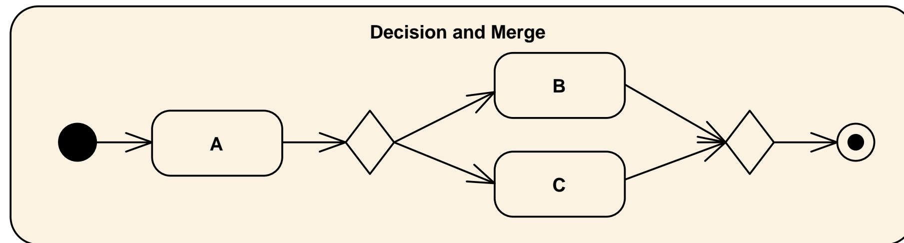

Figure 8.4: Example Activity containing a simple `DecisionNode`/`MergeNode` structure.

to identify important *execution events*, i.e., events which will occur during the execution of our example model, and which will describe what happens at a certain point in execution time. Again, this is not difficult for our example model: the only event is that the contained `Action` is executed. We therefore define an execution event ActionExecutes. Since we will later refer to more than one `Action`, we parameterize that execution event with the `Action`'s name.

Describe semantics: The last step of treating the current example model is to actually describe the model's semantics. We do that in terms of *traces of execution events*: Here, a trace is just a possible sequence of events as identified above. Our example model only has one possible trace, which we can describe as

ActionExecutes("A")

The example presented is very simple, but it serves well to demonstrate the overall approach. The next step would now be to proceed to more complex examples, taking the package structure of the UML metamodel into account. It turns out that the concept of `ActivityEdges` is introduced in package `BasicActivities`, together with concepts like `InitialNode` (which produces a token when the Activity starts) and `ActivityFinalNode` (which consumes tokens). Therefore, the next example model might consist of a sequence of two `Actions`, connected by an `ActivityEdge`, with an according trace consisting of the execution of the two `Actions` in the according order. We skip that example model and proceed to a more complex one in the next section.

8.1.1.2 Example with two Traces

Let us now turn to a (slightly) more complex example model, which is depicted as Fig. 8.4. Its purpose is to demonstrate the semantics of the `DecisionNode` and `MergeNode`.

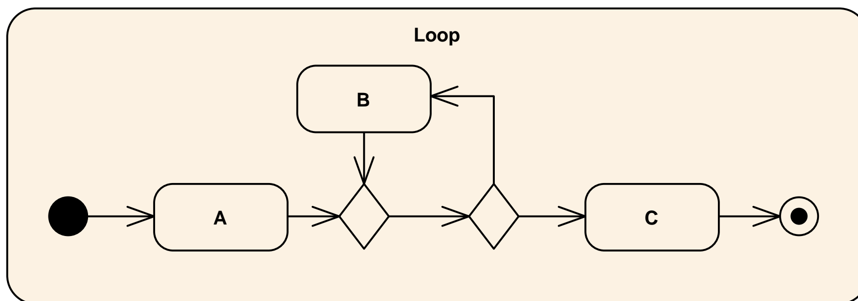

Figure 8.5: Example Activity containing a loop.

This example is interesting because of the fact that it allows for more than one possible execution: a token flowing through the Activity will—as soon as it has passed `Action` "A"—be routed either to `Action` "B" *or* to `Action` "C". Therefore, we will describe the model's behavior by two traces of execution events:

ActionExecutes("A") ActionExecutes("B")

and

ActionExecutes("A") ActionExecutes("C")

We decided to reduce the semantics of Activities to the possible orders of execution of `Actions`, since the `Actions` are the places where the actual work will be performed. However, it would also be possible to use more fine-grained traces like InitialNode() ActionExecutes("A") DecisionNode() ActionExecutes("B") MergeNode() ActivityFinalNode().

In fact, some execution events (e.g., when a token traverses an edge) might become important only when investigating more complex examples at a later stage. If this is the case, an additional execution event can (and should) of course be used to describe the complex model's behavior. Note that this does not render the traces of the simpler examples useless: if such an execution event does occur in a simpler model, too, but has not been used to describe that model's behavior, it is not important for that behavior; otherwise, it would already have been added to the traces of the simpler model when its behavior was investigated. In other words: there is no need to refine the traces of a simple model at a later stage.

8.1.1.3 Example with Loop

The last example model we want to consider here shows how we deal with loops; it is depicted as Fig. 8.5.

Obviously, that Activity gives rise to infinitely many traces; they only differ in the number of times `Action` "B" is executed. However, the transition system is still finite: The execution of "B" always corresponds to the same runtime state (i.e., the state where the only token is sitting on `Action` "B"). As a result, we are still able to use the Activity from Fig. 8.5 as an example model, and to later derive a test case from it.

For this, we use the three traces where "B" is executed zero times, once, and twice, covering the following three situations:

- The loop is not executed at all.
- The loop is executed the very first time, i.e., the `Actions` executed before the loop is entered are lying outside the loop.
- The loop is executed more than once, i.e., the `Actions` executed before the loop is entered (again) are lying inside the loop.

If all three situations are handled properly by the involved rules, we have some confidence that rules are indeed correct with respect to the loop's language elements.

8.1.1.4 Guidelines for Creating Example Models

We have seen how to systematically create example models, and how to precisely but informally describe their behavior. Before we continue with the actual semantics specification and derivation of test cases from the examples, we will outline a few more guidelines for the creation of the examples.

Existing Examples: In the case of the UML, the starting point for semantics specification is the existing but informal specification provided by the OMG. That specification already contains many example models, which should be reused for two reasons: first, these models have been developed by the creators of the UML and are therefore expected to be relevant. Second, the examples are well-known to other users of the UML; these users—if in doubt about the precise meaning of one of the examples—can use our traces of execution events as a reference.

Difficult Semantics: Some language element's semantics will probably be more difficult to understand than others, most likely leading to a more difficult to implement semantics specification (leading to a higher probability of introducing flaws into the semantics specification). Such elements will be identified when discussing their precise meaning. In this case, more example models containing these elements should be created, and each of these examples should concentrate on one or more of the identified difficulties.

Language creation: In the case of DSLs, a new language has to be created from scratch, having certain target users in mind (e.g., business analysts). Creation of the new language should involve these users, and the easiest way to do this is through the discussion of example models, including their precise meanings. In other words: the example models should be created in parallel with the actual language. The examples can later be reused for documentation of the new language.

8.1.1.5 Describing Traces of Execution Events

Providing traces of execution events can be a cumbersome task, especially if the example model whose semantics is to be described contains concurrency. For instance, consider the UML activity shown as Fig. 10.6 on page 223: a huge number of traces of execution events are needed to describe the model's semantics. Creating these traces manually is cumbersone and—even worse—error-prone. Therefore, we have developed a simple textual DSL called *Traces* which can be used to describe traces of execution events in a compact, efficient manner. In this section, we introduce this language.

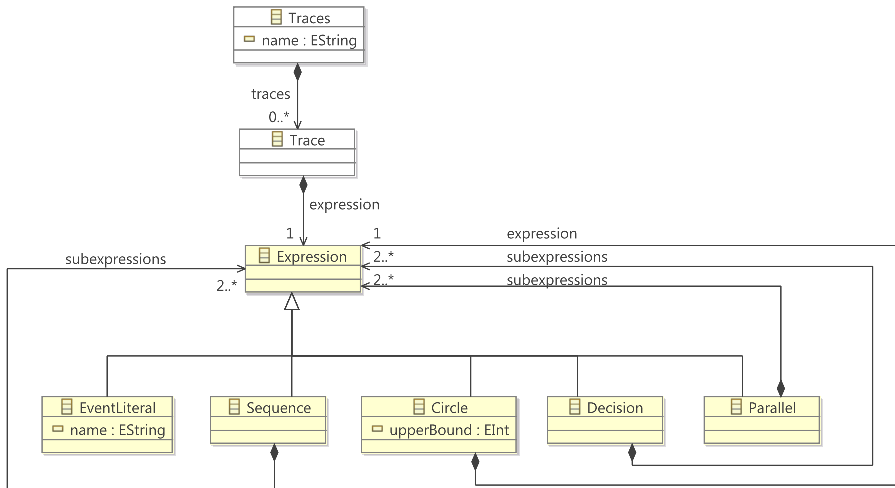

Figure 8.6: Metamodel of the *Traces* language.

The metamodel of the Traces language is depicted as Fig. 8.6. The `Traces` metaclass represents a complete set of traces of execution events describing a certain model's semantics. It has an association `traces` to the `Trace` metaclass representing a single trace. Additionally, each set of traces has a name (which will usually be the name of the model the traces belong to).

A `Trace` contains a single `Expression`, of which 5 kinds are available: an `EventLiteral` represents an actual execution event, and that event is stored in the classes' `name` attribute. The other expressions describe combinations of expressions (and therefore all have a `subExpressions`) association to the `Expression` metaclass). The `Sequence` describes a sequence of expressions. The `Circle` models that the contained expression is to be repeated 0 times, 1 times, ... `upperBound` times (where the latter is the according attribute of the `Circle` metaclass). A `Decision` represents different alternatives, and finally, the `Parallel` models concurrency.

Each `Trace` instance can then be unfolded into a number of simple sequences of `EventLiterals`, i.e., into a set of traces of execution events as one would otherwise have created manually. The unfolding is done recursively by unfolding the most inner elements – the recursion ends as soon as the resulting traces do not contain any composite constructs any more. To make this approach clearer, let us investigate an example `Traces` model,[3] the concrete syntax of which is depicted as Fig. 8.7.

The example is unfolded by first dealing with the `Parallel` and `Circle` expressions inside the `Decision` – the former evaluates to all permutations of the execution events such that the order of the sequences is preserved, i.e., the traces `BCDE`, `BDCE`,`BDEC`,`DEBC`,`DBEC`, and `DBCE`. The latter evaluates to repetitions of `F`, i.e., the empty string, `F`,`FF`, and `FFF`.

[3]Note that again we do not provide the complete formalization of our *Traces* language; the interested reader can investigate the language's definition within plug-ins `de.upb.dmm-.tests.common.traces` and `de.upb.dmm.tests.common.traces.ui`, and the unfolding of traces is implemented in class `de.upb.dmm.tests.common.AbstractTemporalFormu-laGenerator` of plug-in `de.upb.dmm.tests.common`.

```
Traces for test 'TraceDemo'
traces: begin
  A-><< [[B->C||D->E]] ?? ((F::3)) >>->G
traces: end
```

Figure 8.7: Example model of the Traces language, showing all language constructs.

```
ABCDEG
ABDCEG
ABDECG
ADEBCG
ADBECG
ADBCEG
AG
AFG
AFFG
AFFFG
```

Figure 8.8: Evaluation of the Traces model depicted as Fig. 8.7.

Then, the `Decision` is evaluated by unifying the results of the above evaluations (since it models that either one or the other trace of execution events is expected). Finally, each of the traces is prefixed with an `A` and postfixed with a `G`. The unfolding steps as well as the final traces are depicted as Fig. 8.8

In the next section, we will see how models and their expected behavior by means of traces of execution events are translated into executable test cases.

8.1.2 Creating the Semantics Specification and Deriving Test Cases

We have already argued in Sect. 8.1.1 that formal semantics specification is a difficult task. Therefore, we have described how to first gain an understanding of the semantics to be created by investigating example models, and by precisely describing the examples' behavior by means of traces of execution events. In this section, we will perform the actual semantics specification, and we will test that specification using the example models and their behavior. The overall process is depicted as Fig. 8.9.

8.1.2.1 Creating the Semantics Specification

Recall from chapter 6 that DMM uses operational rules to describe behavior: a DMM rule has a precondition and a postcondition, formulated in terms of typed graphs. If a state fulfills the precondition (i.e., if the precondition's graph can be found within the current state graph), the rule will be applied, leading to a new state which fulfills the postcondition (i.e., the precondition's graph will be replaced by the postcondition's graph within the current state, leading to a new state).

This means that in principle, one ore more DMM rules have to be defined for each language construct. Naturally, one starts with defining DMM rules for the

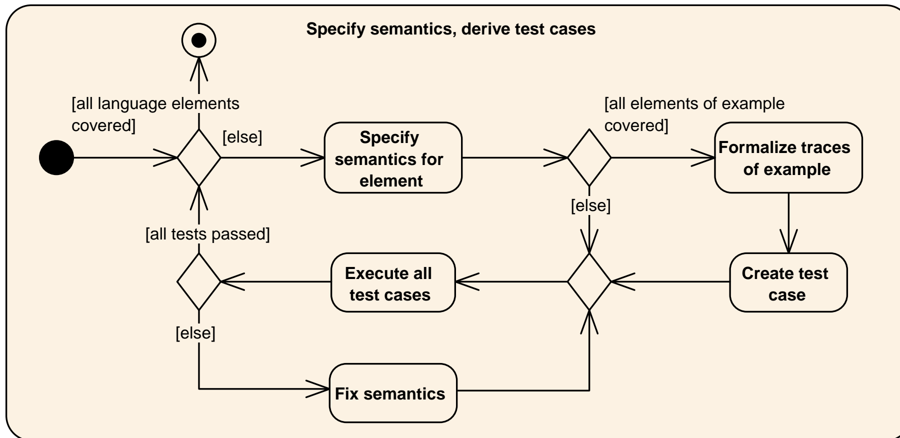

Figure 8.9: Specify semantics, create test cases from example models.

more simple language constructs and adds rules for more complex constructs step by step. Now, the idea is that as soon as all language constructs a particular example model consists of are covered, that example is executable, and its execution should result in a behavior similar to the one identified when creating the example (and described by traces of execution events).

In other words: the example model and the current state of the semantics specification should give rise to a transition system, and that transition system should contain the traces of execution events (and only those traces). This puts a requirement on the DMM specification we are creating: in Chapter 4, we have seen that the GROOVE grammar produced by a DMM specification and a model can be model checked against LTL expressions about the application of GROOVE rules. Since we want to check the transition systems for traces of execution events, we must make sure that for each such event, one or more corresponding DMM rules exists (which will then be transformed into GROOVE rule as seen in Sect. 6.3.2). An occurence of such a DMM rule is then equivalent to an occurence of the according execution event.

Note that this is not a restriction, but a benefit of our approach: one of the goals of creating a formal semantics specification is to check the behavioral quality of models, i.e., to check whether certain behavioral properties hold for the model under consideration. These properties need to be expressed in terms of execution events. To put it another way: our approach makes sure that the resulting semantics specification indeed allows for the verification of such properties.

8.1.2.2 Translating Traces into LTL formulas

But how to convert the traces of execution events into properties which can be checked against a transition system? This is in fact quite simple; let us demonstrate our approach using the more complex example introduced in Sect. 8.1.1.2. The trace

ActionExecutes(“A”) ActionExecutes(“B”)

can also be read as follows: the transition system representing the model's behavior must contain a trace where at some point in time, `Action` "A" must be executed. From that point on, there must be a "subtrace" such that `Action` "B" is executed at some point in time. Using the temporal logic dialect LTL [134], this can be expressed as follows: As we have seen in Sect. 4.3.1 on page 31, the LTL formula

$$\mathbf{F}(r)$$

expresses the fact that **F**inally[4], property r holds. Since we model check against the application of DMM rules, r will be a such a rule (and reveals information about the state the rule is applied to: it must be the case that the precondition of r holds for that state).

Now, our DMM specification will contain a rule action.start(action.name), corresponding to the execution event ActionExecutes(name) as described earlier. Therefore, the trace shown above can be translated into the following LTL formula, which can then be checked against the transition system:

$$p_1 := \mathbf{F}(\text{action.start(“A”)} \wedge \mathbf{XF}(\text{action.start(“B”)}))$$

However, if p_1 holds for a transition system, then this means that all traces of that transition system fulfill p_1. This is not what we want to express (and would of course not be true for our example): We want to know if the transition system contains an according trace, but it may (and probably will) contain other traces not fulfilling p_1. We can deal with this by negating p_1, expressing that on all traces of the transition system, p_1 does *not* hold. If the model checker finds out that $\neg p_1$ indeed does *not* hold, we know that the transition system contains the trace as desired.

Checking our transition system as described above ensures that it indeed contains the trace as desired. There is one remaining problem, though: Up to now, we only know that there are traces such that `Actions` "A" and "B" are executed, but we do not know what happens before "A", between "A" and "B", and after "B".

To make our LTL formula more precise with respect to that problem, we have to dive deeper into LTL: we have to make use of the **U**ntil operator, the ne**X**t operator, and the **G**lobally operator. To explain the new formula, we first define some helper constructs.

First, to be able to use a more compact representation, we will write action.start("A") as a_A (a_B, a_C accordingly). Now, let $R = \{a_A, a_B, a_C\}$ be the set of all rules corresponding to execution events relevant for the model under consideration. Finally, we define the predicate $\hat{R}$ as $\wedge_{r \in R}(\neg r)$.

We will now construct the formula step by step. The first part looks as follows:

$$P_1 := \hat{R}\,\mathbf{U}\,X_A$$

The intuition is that we want to find the first occurrence of rule a_A on some path; therefore, we require that none of the rules contained in R occurs **U**ntil a_A occurs (which will be part of X_A). The definition of X_A reads as

$$X_A := a_A \wedge \mathbf{X}(\hat{R}\,\mathbf{U} X_B)$$

[4]"Finally" must be understood as "at some point in time" here.

This is the most important part of the formula to construct. The idea is that since we have found the first occurrence of a_A, we want to make sure eventually in the future X_B will hold, and before that, no other rules out of R will occur. Note that the $\mathbf{X}$ is needed since we have to look at the next state, because in the current state, a_A holds, so $\hat{R}$ can never be true. Now for X_B:

$$X_B := a_B \wedge \mathbf{XG}(\hat{R})$$

This formula completes our definition of P_1. It expresses the fact that after a_B has occurred, no other rule from R will ever occur again.

All together, $\neg P_1$ expresses exactly the desired property of our transition system: it is false iff the transition system contains a trace such that a_A and a_B occur in the desired order, and there are no other occurrences of rules from R at other places. Additionally, it is easy to see that the above construction can be extended to traces of arbitrary length by using several expressions similar to X_A, where a_A is replaced by the rule to be checked, and by nesting them as above.

Note that the above could be expressed more easily by using *property specification patterns* [42], a collection of temporal logic formulas which can be used to express properties which are used frequently. We still decided to provide the translation using basic LTL contructs, since we expect them to be more common to most readers than the property specification patterns.

8.1.2.3 Creating Test Cases

In the last section, we have seen how to translate a trace of execution events into an LTL formula, which can then be model checked against the transition system. It is now straight-forward to create a test case from a model and a set of such traces.

First, all the traces belonging to the example model under consideration have to be translated into LTL formulas as explained above. In the case of our more complex UML Activity, this will result in two LTL formulas P_1 and P_2 (we have seen P_1 in the last section). Then, a model checker can be used to verify whether all these properties hold. If this is the case, we know that the expected behavior is contained within the transition system; this means that our semantics specification so far produces the behavior as desired. Otherwise, we know which trace of execution events is not contained in the resulting behavior, and we can use that information to fix the semantics specification.

It remains to show whether this is the only behavior produced by our semantics specification: there might be other traces which do not fulfill P_1 or P_2, i.e., some undesired behavior is going on. Therefore, we check one more property which ensures that the transition system indeed only contains the desired behavior:

$$P_1 \vee P_2$$

The above formula holds iff for all traces through the transition system, either P_1 or P_2 hold. Its verification will fail if the transition system contains undesired behavior. In this case, the model checker will provide a counter example, i.e., a trace which does not belong to the expected ones. That counter example can then be used to fix the semantics specification (in Chapter 10 on page 215, we introduce some basic toling for supporting the language engineer in this task, which might—even with a counter example—still be quite difficult).

8.1.2.4 Automatic Execution of Test Cases

To support the creation of DMM semantics specifications, we have implemented a Java framework which enables the automatic execution of test cases as described above. For this, we have used JUnit [115], which provides convenient ways to execute our test cases, including a GUI showing which tests passed or failed for which reasons. An execution of a test case works as follows:

First, the example model under consideration is translated into a GROOVE graph, which serves as the start state for the transition system to be computed. Next, the traces of execution events—written in the language we have introduced in Sect. 8.1.1.5—are translated into LTL formulas as described in Sect. 8.1.2.2. Then, the generation of the transition system is started, using the current state of the DMM specification to be built. Finally, the LTL formulas are verified one by one; if a verification fails, the according JUnit test will fail, providing a message which points at the trace not being contained in the transition system.

8.2 Coverage Criteria for Tests of DMM Specifications

In the last section, we have seen how a DMM semantics specification can be tested by means of example models and their expected behavior formalized as traces of execution events. If all our tests pass, we have some confidence that the semantics specification indeed realizes the language's semantics as desired. That confidence of course depends on the tests themselves. For instance, we will naturally trust our semantics specification more if we have many test cases (instead of only a few ones which e.g. test just the main features of our semantics specification). However, the number of tests is not a very helpful quality criterion for a test suite, since all tests could concentrate on testing only a small part of the software system, leaving other parts completely untested.

Therefore, it would be beneficial for our approach of test-driven semantics specification if we would have a way to measure the quality of our tests – the higher their quality, the higher our trust in the semantics specification.

In software engineering, the most important criterion for measuring the quality of a software system's tests is *test coverage*. The International Software Testing Qualifications Board (ISTQB) defines coverage as "the degree, expressed as a percentage, to which a specified coverage item has been exercised by a test suite", where a coverage item is defined as "an entity or property used as a basis for test coverage, e.g. equivalence partitions or code statements" [108, p. 16].

In this section, we will define notions of test coverage for DMM semantics specifications. Before we do that, in the next section we will discuss the notion of coverage in general, and we will investigate what kinds of coverage items to define for DMM specification. Section 8.2.2 will then introduce a data structure to be covered, and Sect. 8.2.3 will introduce several coverage criteria for DMM. Finally, Sect. 8.2.4 will discuss how our coverage criteria relate to each other in terms of expressiveness.

This section is based on [5] and [4]. The concepts described here are implemented in the plug-ins `de.upb.dmm.tests.coverage.rule` and `de.-upb.dmm.tests.coverage.edge`.

8.2.1 Covering DMM Specifications

In software engineering, the most important criterion for choosing a coverage item is the kind of test: in black-box testing, we do not have access to the source code of the system under test (SUT) – the system in treated as a black box. As a result, coverage items can only be defined by taking the test's input data into account. For instance, an important technique in black-box testing is *equivalence partioning*, where the test input is devided into classes, each of which shall produce the same result for all members of a class. Coverage can then be defined as the number of classes for which test cases do exist.

In white-box testing, the SUT is transparent to the tester – the source code (and all other artifacts) of the SUT can be accessed. This allows to define much more fine-grained coverage criteria. The idea is to derive a data structure called *control-flow graph*[5] from the source code, where nodes of the graph are statements of the source code, and there is an edge between two statements s_1, s_2 if there is a possible execution of the program such that s_2 is executed immediately after s_1. Such a data structure can then be covered: For instance, *statement coverage* is defined as the number of statements which are executed by our test suite, divided by the number of all statements. Other coverage criteria for control-flow graphs include branch coverage (where the outgoing edges of each node are covered) or path coverage (where the possible pathes through the control-flow graph are covered).[6] See e.g. [202, 85] for more information on coverage criteria and their definitions.

But what to cover in DMM? In contrast to software engineering, where the flow of control is basically given by the order of program statements and constructs such as loops and `if-then-else`, graph transformation rules can in general be applied in any order: At every state, any rule can potentially match and be applied. However, in DMM the situation is sightly different: We have seen in DMM's definition that if a DMM rule contains an invocation, the next rule to be applied must be an invoked one; if no such rule has been specified, the DMM specification is considered to be erroneous.

Let us illustrate this with an example: Assume that bigstep rules A and B both contain a single invocation of a rule S. In this case, our tests should make sure that in both cases, S can be applied after the previous rule (A or B). For instance, A and B must make sure that S's application context is available.

As such, test coverage of a DMM specification can be defined as the degree to which the potential orders of invoked rules has been exercised by our test models.

In the next section, we will introduce a data structure called *invocation graph* which models all potential orders of invoked rules.

8.2.2 Invocation Graph

In the case of common graph transformation rules, all rules must be checked for matching in every state, since there is no concept which restricts the order of

[5]Other data structures used in covering white-box testing e.g. describe the flow of data (data-flow graph).

[6]Note that in most cases, a path coverage of 100% can not be achieved since the control-flow graph contains an infinite number of pathes; however, path coverage is often used as the *theoretical optimum* to which other, actually applicable coverage criteria are compared.

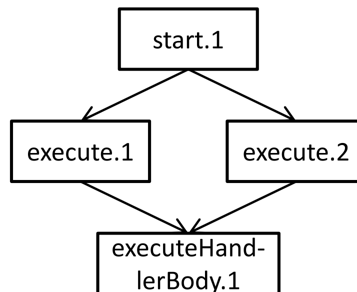

Figure 8.10: Example invocation graph for bigstep rule action.start()# with unique name start.1.

application of those rules.[7] In contrast, we have seen in chapter 6 that DMM invocations allow to implement some flow of control: If a DMM rule contains an invocation, only rules compatible to that invocation can match and be applied.

This allows to compute potential orders of execution of DMM rules by only evaluating the rules' invocations. Invocations graphs are the result of such a computation and reflect exactly those potential sequences of applications of DMM rules: Nodes of an invocation graph correspond to DMM rules, and a rule r_1 can be followed by a rule r_2 if and only if there is an edge between the corresponding nodes. Note that an invocation graph might (and usually will) contain edges corresponding to orders of rule applications which are not actually possible, e.g. since r_1 contains an invocation compatible to r_2 but creates object structures which prevent r_2 from matching.

Since bigstep rules can not be invoked (and can therefore match as soon as the invocation stack is empty, see Sect. 6.3.2.4), each bigstep rule gives rise to a single invocation graph, from which a sequence of applications of smallstep rules might follow – the bigstep rule corresponds to the root node of its invocation graph, from which all potential sequences of rule applications are rolled out.

Let us illustrate the above with an example from the UML activity semantics created by Hornkamp [105]. The invocation graph depicted as Fig. 8.10 reflects the following situation: the bigstep rule with unique name start.1 contains an invocation, and two smallstep rules (with unique names execute.1 and execute.2) are compatible to that invocation. Both of these rules contain an invocation themselves, for which only one compatible rule exists (the rule with unique name executeHandlerBody.1). Note that the invocation graph would look the same if start.1 would contain both invocations in the according order (and rules execute.1 and execute.2 would contain no invocation).

Figure 8.11 describes a slightly different situation: Rule flow.1 again contains a single invocation for which two compatible rules exist: accept.1 and accept.2, and both rules contain two invocation for which the single compatible rules notifySpawns.1 and moveOffers.1 exist. The difference to the above example is that rule accept.1 additionally contains a third invocation with another single compatible rule finish.1. In this case, two different nodes are needed which both

[7] Usually, graph transformation rules are designed to (at least partly) match in a specific order, e.g. by using *trigger nodes* [91]. However, it is the responsibility of the rule creator to make sure that the rules indeed match in the desired order. DMM invocations take that burden from the creator's back.

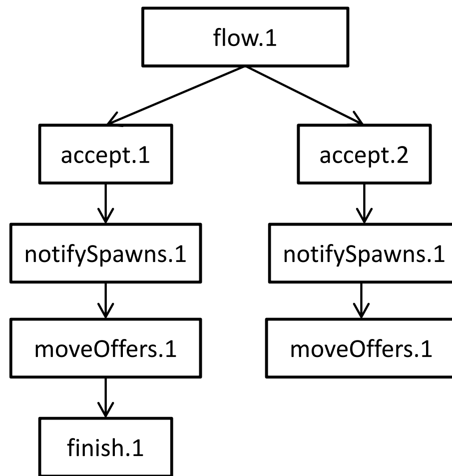

Figure 8.11: Example invocation graph for bigstep rule forkNode.flow()# with unique name flow.1.

correspond to the DMM rule with unique name notifySpawns.1 (and the same holds for rule moveOffers.1): If the rules would be reflected by a single node (as in the above example), there would be paths through the invocation graph which would *not* correspond to possible sequences of applications of DMM rules (namely, the invocation graph would allow for the sequence flow.1 → accept.2 → notifySpawns.1 → moveOffers.1 → finish.1, which is not possible since neither rule accept.2 nor rules notifySpawns.1 or moveOffers.1 contain an invocation to which rule finish.1 is compatible).

To put this more generally: two nodes corresponding to the same DMM rule are merged if at the point of execution of that rule, the invocation stack is the same in both cases. This is true for our example in Fig. 8.10: After the executions of rules execute.1 and execute.2, the invocation stack contains the only rule executeHandlerBody.1. Since the invocation stack is equal in both cases, only one node is needed. In contrast, in the example of Fig. 8.11, after execution of rule accept.1, the invocation stack contains three rules, after execution of rule accept.2, the invocation stack contains only two rules. Since the two stacks differ, two nodes are needed.

The final example of this section is depicted as Fig. 8.12. Here, rule supplyStreamingToken.1 contains a single invocation, for which two compatible rules exist: rules destroy.1 and destroy.2. These two rules realize a recursion, where destroy.1 contains an invocation for which both the rule itself and rule destroy.2 are compatible. The recursive call is modeled by the self edge of the destroy.1 node, and the end of the recursion is reached as soon as rule destory.2 matches and is applied.

Having a good intuition on how invocation graphs are reflecting a DMM ruleset's invocation structure, let us now investigate the algorithm used to compute invocation graphs, which is depicted as Listing 8.1. Lines 1–6 define a data structure called `RuleNode` which—as the name suggests— represents DMM rules as nodes in the invocation graph. For this, a `RuleNode` stores its rule's unique name as well as a list of invocations which are to be processed at the

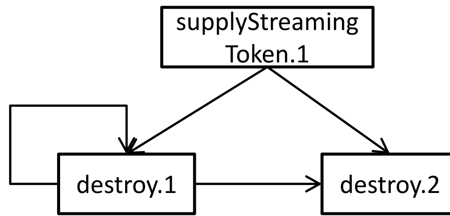

Figure 8.12: Example invocation graph for bigstep rule inputPin.supplyStreamingToken()# with unique name supplyStreamingToken.1.

time the rule is applied. Two RuleNodes are equal if both their unique names and list of open invocations are the same.

Listing 8.1: Algorithm for computing the invocation graph of a bigstep rule

```
RuleNode                                                              1
    String uniqueName                                                 2
    List openInvocations                                              3
                                                                      4
    boolean equals(RuleNode other)                                    5
      return uniqueName = other.uniqueName and openInvocations.equals(  6
          other.openInvocations)
                                                                      7
DirectedGraph computeInvocationGraph(BigstepRule bigstepRule)         8
    DirectedGraph invocationGraph := new DirectedGraph()              9
    RuleNode rootNode := new RuleNode(bigstepRule.uniqueName,         10
        bigstepRule.invocations)
    invocationGraph.addVertex(rootNode)                               11
    exploreNode(invocationGraph, rootNode)                            12
    return invocationGraph                                            13
                                                                      14
void exploreNode(DirectedGraph invocationGraph, RuleNode node)        15
     if (not node.openInvocations.isEmpty())                          16
      Invocation invocation := node.openInvocations.removeFirstElement  17
          ()
      for each (Rule rule in getCompatibleRules(invocation))          18
          RuleNode ruleNode := new RuleNode()                         19
          ruleNode.uniqueName := rule.uniqueName                      20
          ruleNode.openInvocations.appendAll(rule.invocations)        21
          ruleNode.openInvocations.appendAll(node.openInvocations)    22
          if (invocationGraph.containsVertex(ruleNode))               23
              invocationGraph.addEdge(node, ruleNode)                 24
          else                                                        25
              invocationGraph.addVertex(ruleNode)                     26
              invocationGraph.addEdge(node, ruleNode)                 27
              exploreNode(invocationGraph, ruleNode)                  28
```

The operation shown in lines 8–13 then starts the computation of an invocation graph; it gets a bigstep rule as input and returns the rule's invocation graph. The operation creates the graph object and a RuleNode corresponding to the bigstep rule. As such, the RuleNode receives the bigstep rule's unique name as well as all the rule's invocations (reflecting the fact that after the bigstep rule has been applied, all the rule's invocations are still to be processed). The operation then adds the RuleNode to the invocation graph and starts the main part of computing the invocation graph by invoking the operation exploreNode, passing the invocation graph and the bigstep rule's RuleNode

as parameters.

Operation `exploreNode` does the actual work. Its idea is as follows: First, if the `RuleNode` to be processed (with name `name`) does not contain any invocations, the operation immediately returns – we have reached a leaf node of the invocation graph. Otherwise, the very first invocation to be processed is taken from the `node`'s list of open invocations (line 17). Then, the operation loops over all rules compatible to the invocation (see Sect. 6.3.2.4 for the definition of compatibility between an invocation and a smallstep rule). For each compatible rule, a `RuleNode` object is created and receives the rule's unique name and invocations. However, at point of time of applying the current rule, there might still be invocations from previously processed rules to be executed at a later stage – these invocations are appended to the list of open invocations in line 22.

Finally, it is checked whether the invocation graph already contains a `RuleNode` similar to the one just created. If this is the case, we only need to add an edge from the `node` to the `ruleNode`. No further exploration of `ruleNode` is necessary: Since the invocation graph's node has already been explored, and since that node contains exactly the same open invocations than our newly computed `ruleNode`, exploring the `ruleNode` would result in exactly the same structure we have already found when exploring the invocation graph's node. Otherwise, we add the `ruleNode` and an edge between the invoking `node` and the invoked `ruleNode` to the invocation graph and recursively continue with the exploration of `ruleNode`.

The algorithm described above results in an invocation graph which contains exactly one `RuleNode` for each (transitively) invoked rule and state of open invocations, and it contains an edge between two `RuleNodes` if and only if one rule might follow another rule. However, the algorithm does not always terminate. The reason for this is that some DMM specifications give rise to infinitely large invocation graphs. As a simple example, consider a DMM specification containing a smallstep rule which contains two invocations: The first one is a recursive invocation (i.e., the rule invokes itself), and the second one is an arbitrary invocation. This situation will result in an infinite invocation graph, where nodes are added over and over, and each of these nodes has one more invocation on its invocation stack than the previous node. It is the language engineer's responsibility to be aware of such situations (as it is her responsibility to create example models which give rise to finite transition systems).

Now that we have defined the invocation graphs of a ruleset, in the next section we will introduce several ways to cover these graphs.

8.2.3 Coverage Criteria

As we have explained above, the invocation graphs formalize the order of potential executions of smallstep rules. Now, the idea is to cover that data structure by means of executing the test models and marking nodes and edges of the invocation graphs as covered.

The invocation graphs of a ruleset contain two kinds of elements to be covered: nodes and edges. Therefore, we have developed two categories of coverage criteria: In Sect. 8.2.3.2, we will introduce three notions of *rule coverage*, followed by the definition of three notions of *edge coverage* in Sect. 8.2.3.3. Before we do that, we explain the general approach of computing coverage in the next section.

8.2.3.1 General Coverage Computation

In the following, the coverage criteria are described as follows: First, a general `CoverageCalculator` is introduced which relies on two functions to be provided by the different coverage criteria: The first function computes the coverage items to be covered, and the second function decides whether a coverage item is covered by a given transition system.

Listing 8.2 shows the `CoverageCalculator` as pseudo code. The algorithm receives a DMM ruleset and a set of transition systems as input and returns the ruleset's test coverage in percent.

Listing 8.2: General algorithm for coverage computation

```
double computeCoverage(Ruleset ruleset, Set transitionSystems)                  1
  Map invocationGraphs := new Map()                                              2
  for each (Rule rule in ruleset.rules)                                          3
    if (rule is BigstepRule)                                                     4
      invocationGraphs.put(rule, computeInvocationGraph(rule))                   5
  Set coverageItems := computeCoverageItems(ruleset, invocationGraphs)           6
  Set coveredItems := new Set()                                                  7
  for each (GraphTransitionSystem gts in transitionSystems)                      8
    performCoverageAnalysis(ruleset, invocationGraphs, coverageItems,            9
        coveredItems, gts)
  return coveredItems.size() / coverageItems.size()                              10
                                                                                 11
void performCoverageAnalysis(Ruleset ruleset, Map invocationGraphs, Set          12
     coverageItems, Set coveredItems, GTS gts)
  Map name2transitions := computeMap(ruleset, gts)                               13
  for each (CoverageItem coverageItem in coverageItems)                          14
    if (not coveredItems.contains(coverageItem) and isCovered(gts,               15
        invocationGraphs, name2transitions, coverageItem))
      coveredItems.add(coverageItem)                                             16
                                                                                 17
Map computeMap(Ruleset ruleset, GraphTransitionSystem gts)                       18
  Map result := new Map()                                                        19
  for each (Rule rule in ruleset.rules)                                          20
    if (rule is BigstepRule or rule is SmallstepRule)                            21
      Set transitions := new Set()                                               22
      for each (GraphTransition graphTransition in gts.transitions)              23
        if (matches(graphTransition, rule.uniqueName))                           24
          transitions.add(graphTransition)                                       25
      map.put(rule.uniqueName, transitions)                                      26
  return result                                                                  27
```

The algorithm starts with computing the invocation graphs of all bigstep rules contained in the given ruleset (lines 2–5). It then receives the coverage items for the given coverage criterion in line 6 – function `computeCoverageItems()` is the first function to be provided by concrete coverage realizations. Then, the algorithm iterates over all transition systems and performs coverage analysis for each of them. Finally, the coverage is computed by dividing the number of covered items by the number of all items (line 10).

The actual coverage analysis is shown in lines 12–16. First, a map is computed, mapping each DMM rule to a set of corresponding transitions from the transition system currently analyzed. Then, the algorithm iterates over the coverage items and—given that an item is not yet covered—uses the `isCovered()` function to check whether the given coverage item is indeed covered by the transition system at hand.

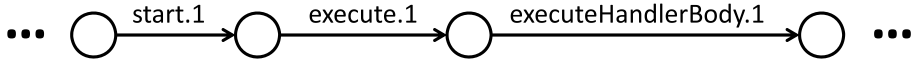

Figure 8.13: Excerpt of a test model's transition system.

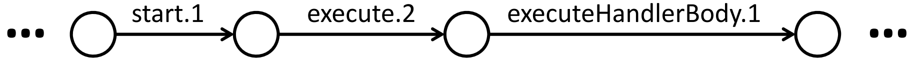

Figure 8.14: Excerpt of a test model's transition system.

In the following, we will introduce the different coverage criteria by means of their realizations of the functions `computeCoverageItems()` and `isCovered()`.

8.2.3.2 Rule Coverage Criteria

As stated above, the first kind of elements to be covered are nodes of the invocation graphs, which correspond to the execution of DMM rules. In the following, we will introduce three kinds of *rule coverage*.

Rule Coverage Our goal is to make sure that our test models make use of the DMM rules forming the language's semantics in as many ways as possible. However, the first (and most basic) requirement is that each rule of our semantics specification should be used during the execution of at least one of our test models. Therefore, our first coverage criterion is called *rule coverage*.

Let us illustrate rule coverage with an example: We will cover the invocation graph depicted as Fig. 8.10. Assume that we have a single test model, the transition system of which contains the sequence of states and transitions as shown in Fig. 8.13. It is easy to see that three of the four rules occuring in the invocation graph are contained in the transition system. As such, our test model would result in a rule coverage of 75%.

Note that for computing rule coverage, it would not be necessary to compute the invocation graphs – it would suffice to just traverse the example models' transition systems, and to mark each rule of the DMM specification as used which occurs in at least one of our transition systems. However, we still decided to define rule coverage on base of the invocation graph data structure, since this allows for a better understanding of the differences between the coverage criteria we are presenting.

To increase rule coverage, we would have to add a second test model,[8] the execution of which would result in the application of the rule with unique name execute.2. For instance, if our test model's transition system would contain the sequence of transitions as depicted in Fig. 8.14, we would have achieved a rule coverage of 100%.

Let us now define rule coverage by means of pseudo code, which is depicted as Listing 8.3. In the case of rule coverage, it suffices to describe the coverage items by means of a rule's uniqueName – two coverage items are equal if their respective unique names are equal (lines 1–5). Consequently, the computation

[8]Of course, we could also modify our existing test model. However, it is recommended to keep the test models as simple and small as possible.

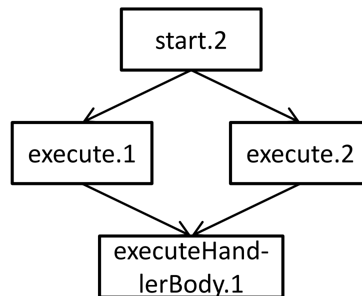

Figure 8.15: Example invocation graph for bigstep rule action.start()# with unique name start.2.

of the coverage items comes down to collecting all different unique names. Note that in lines 11–13 a new CoverageItem is only added to the result set if that set does not yet contain a coverage item equal to the one to be added; thus, the final set will contain one coverage item per rule of the ruleset.

Listing 8.3: Subroutines for computing rule coverage

```
CoverageItem                                                          1
  String uniqueName                                                   2
                                                                      3
  boolean equals(CoverageItem other)                                  4
    return uniqueName = other.uniqueName                              5
                                                                      6
Set computeCoverageItems(Ruleset ruleset, Map invocationGraphs)       7
  Set result := new Set()                                             8
  for each (DirectedGraph invocationGraph in invocationGraphs.values) 9
    for each (RuleNode ruleNode in invocationGraph.vertices)          10
      CoverageItem item = new CoverageItem(ruleNode.uniqueName)       11
      if (not result.contains(item))                                  12
        result.add(item)                                              13
  return result                                                       14
                                                                      15
boolean isCovered(GraphTransitionSystem gts, Map invocationGraphs, Map 16
    name2transitions, CoverageItem coverageItem)
  Set transitions := name2transitions.get(coverageItem.uniqueName)    17
  return transitions.size() > 0                                       18
```

Using the map computed by the generic coverage algorithm we introduced in Sect. 8.2.3.1, it can now be decided whether a coverage item is covered by checking whether the number of transition corresponding to the given rule is greater than 0 – if this is the case, we know that the rule is executed at least once by the example model giving rise to the given transition system.

Rule Coverage Plus Now, let us assume that we have a second invocation graph, which is depicted as Fig. 8.15. This invocation graph differs from the one in Fig. 8.10 only by the root bigstep rule (which is start.2 in this case). As such, we have five different rules, and our two test models would result in a rule coverage of 80%.

The smallstep rules execute.1, execute.2 and executeHandlerBody.1 now do not only need to work correctly when (transitively) invoked by bigstep rule

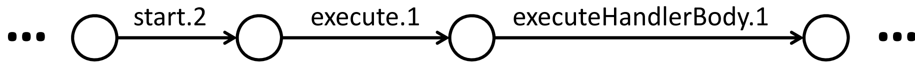

Figure 8.16: Excerpt of a test model's transition system.

start.1, but also when invoked by bigstep rule start.2. However, rule coverage does not take this into consideration. Therefore, we have defined the notion of *rule coverage plus*: For this coverage criterion, each smallstep rule must be covered separately for all bigstep rules it is transitively invoked by.

As an example, the number of nodes to be covered in rule coverage plus is eight for the given two invocation graphs, and our two example models would result in a coverage of 50%. If we add a third example model containg the sequence of transitions as depicted in Fig. 8.16, the coverage increases to $\frac{7}{8}$=87.5%: not only is bigstep rule start.2 now covered, but both rules execute.1 and executeHandlerBody.1 are executed in the context of rule start.2 and are therefore covered with respect to rule coverage plus.

Now for the definition of rule coverage plus, the pseudo code of which is depicted as Listing 8.4. Since we want to track whether a rule is executed while being invoked by a certain bigstep rule, that bigstep rule is also stored within our coverage items – two items are now equal if both their bigstep rules and unique names are equal (lines 1–6).

Listing 8.4: Subroutines for computing rule coverage plus

```
CoverageItem                                                                1
  BigstepRule bigstepRule                                                   2
  String uniqueName                                                         3
                                                                            4
  boolean equals(CoverageItem other)                                        5
    return uniqueName = other.uniqueName and bigstepRule.uniqueName =       6
        other.bigstepRule.uniqueName
                                                                            7
Set computeCoverageItems(Ruleset ruleset, Map invocationGraphs)             8
  Set result := new Set()                                                   9
  for each (Rule rule in ruleset.rules)                                     10
    if (rule is BigstepRule)                                                11
      DirectedGraph invocationGraph := invocationGraphs.get(rule)           12
      for each (RuleNode ruleNode in invocationGraph.vertices)              13
        CoverageItem item = new CoverageItem(rule, ruleNode.uniqueName)     14
        if (not result.contains(item))                                      15
          result.add(item)                                                  16
  return result                                                             17
                                                                            18
boolean isCovered(GraphTransitionSystem gts, Map invocationGraphs, Map      19
    name2transitions, CoverageItem coverageItem)
  Set transitions := name2transitions.get(coverageItem.uniqueName)          20
  for each (GraphTransition graphTransition in transitions)                 21
    Set invokingBigstepRules := new Set()                                   22
    collectInvokingBigstepRules(gts, graphTransition,                       23
        invokingBigstepRules)
    for each (String invokingBigstepRule in invokingBigstepRules)           24
      if (matches(invokingBigstepRule, coverageItem.bigstepRule.            25
          uniqueName))
        return true                                                         26
  return false                                                              27
```

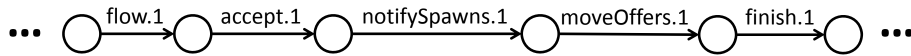

Figure 8.17: Excerpt of a test model's transition system.

Computing the coverage items is done by traversing all invocation graphs, creating a coverage item for each graph's nodes, and adding the created coverage items to the result set to be returned (lines 8–17). Note that as always, a coverage item will only end up in the result set if that set does not already contain a coverage item being equal to the one to be added.

Finally, before we explain how coverage of a certain item is checked, let us quickly introduce a helper function, which is depicted as Listing 8.5. The function `collectInvokingBigstepRules()` computes the set of all bigstep rules which might have (transitively) invoked the current rule. This is realized by the algorithm of Fig. 8.5 as follows: It receives a transition system, a transition and a set as input – after execution of the algorithm, the set will contain the invoking bigstep rules. Starting with the given transition's source state, the algorithm then goes backwards through the transition system in a recursive way. The recursion stops as soon as a bigstep rule is found (and added to the result set).

Listing 8.5: Routines to be reused by the coverage algorithms

```
void collectInvokingBigstepRules(GraphTransitionSystem gts,                1
    GraphTransition graphTransition, Set invokingBigstepRules)
  Set incomingTransitions := gts.getIncomingTransitions(graphTransition    2
      .source)
  for each (GraphTransition transition in incomingTransitions)             3
    if (transition represents BigstepRule)                                 4
      invokingBigstepRules.add(transition.label)                           5
    else                                                                   6
      collectInvokingBigstepRule(gts, transition, invokingBigstepRules)    7
```

It is now straight-forward to check whether one of our coverage items is indeed covered – function `isCovered()` works as follows: Using the map computed by the generic coverage algorithm, we first receive the set of transitions corresponding to the given rule (line 20). We then iterate over all the transitions found; for each, we compute the set of invoking bigstep rules as defined above (lines 22–23). Finally, we iterate over these bigstep rules – if we find the coverage item's bigstep rule within the set, the coverage item is indeed covered, and we return `true` (lines 24–26).

Rule Coverage Plus Plus Finally, let us add the invocation graph depicted as Fig. 8.11 as the third invocation graph, and let us also add another example model, the transition system of which shall contain the sequence of transitions as shown in Fig. 8.17. The invocation graphs and test models result in a rule coverage of $\frac{10}{11}$=90.9% (the only uncovered rule is accept.2) and a rule coverage plus of $\frac{12}{14}$=85.7% (the uncovered rules are rule execute.2 in the context of bigstep rule start.2 and rule accept.2 in the context of bigstep rule flow.1).

Now, the difference between rule coverage and rule coverage plus was that some rules were occuring in more than one invocation graph. In the invocation graph of Fig. 8.11, we have another situation: rules notifySpawns.1 and

moveOffers.1 occur more than once in a single invocation graph. The situation is different because the rules are invoked by both rules accept.1 and accept.2 and therefore need to work correctly in both cases. However, rule coverage plus does not distinguish between nodes within the same invocation graph which correspond to the same rule.

The solution is rule coverage plus plus, which means that every single node of our invocation graphs has to be covered by an example model. For the situation described above, we therefore have a coverage of $\frac{12}{15}$=80%.

The definition of rule coverage plus plus as pseudo code is depicted as Listing 8.6. The definition of the coverage item can once more be seen at the top of that listing (lines 1–6): Since we now need to distinguish nodes within invocation graphs, it is not enough any more to store a rule's unique name (since an invocation graph might contain several nodes which all correspond to the same rule). Thus, for two coverage items to be considered equal, they need to have the same bigstep rule and rulenode.

Listing 8.6: Subroutines for computing rule coverage plus plus

```
CoverageItem                                                              1
  BigstepRule bigstepRule                                                 2
  RuleNode ruleNode                                                       3
                                                                          4
  boolean equals(CoverageItem other)                                      5
    return bigstepRule.uniqueName = other.bigstepRule.uniqueName and      6
        ruleNode.equals(other.ruleNode)
                                                                          7
Set computeCoverageItems(Ruleset ruleset, Map invocationGraphs)           8
  Set result := new Set()                                                 9
  for each ((BigstepRule rule, DirectedGraph invocationGraph) in          10
      invocationGraphs)
    for each (RuleNode ruleNode in invocationGraph.vertices)              11
      CoverageItem item = new CoverageItem(rule, ruleNode)                12
      if (not result.contains(item))                                      13
        result.add(item)                                                  14
  return result                                                           15
                                                                          16
boolean isCovered(GraphTransitionSystem gts, Map invocationGraphs, Map    17
    name2transitions, CoverageItem coverageItem)
  DirectedGraph invocationGraph := invocationGraphs.get(coverageItem.     18
      bigstepRule)
  Set edgeCandidates := name2transitions.get(coverageItem.ruleNode.       19
      uniqueName)
  for each (GraphTransition edgeCandidate in edgeCandidates) {            20
    if (isCovered(gts, invocationGraph, coverageItem.bigstepRule,         21
        coverageItem.ruleNode, edgeCandidate, new Set()))
      return true                                                         22
  return false                                                            23
                                                                          24
boolean isCovered(GraphTransitionSystem gts, DirectedGraph                25
    invocationGraph, BigstepRule bigstepRule, RuleNode ruleNode,
    GraphTransition transition, Set seenInvocationEdges)
  if (ruleNode.uniqueName = bigstepRule.uniqueName and matches(           26
      transition, bigstepRule.uniqueName))
    return true                                                           27
  for each (Edge invocationEdge in invocationGraph.incomingEdgesOf(       28
      ruleNode))
    if (seenInvocationEdges.contains(invocationEdge))                     29
      return false                                                        30
    seenInvocationEdges.add(invocationEdge)                               31
```

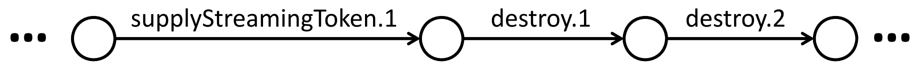

Figure 8.18: Excerpt of a test model's transition system.

```
  RuleNode invocationSourceNode := invocationEdge.source                32
  for each (GraphTransition sourceTransition in gts.                    33
      getIncomingTransitions(transition.source))
    if (matches(sourceTransition, invocationSourceNode.uniqueName))     34
      if (isCovered(gts, invocationGraph, bigstepRule,                  35
          invocationSourceNode, sourceTransition, new Set(
          seenInvocationEdges)))
        return true                                                     36
return false                                                            37
```

But how to find out which concrete rule node is covered by a certain transition? We need to make sure that the transition system contains a sequence of transitions corresponding to rules such that the sequences starts with the coverage item's bigstep rule, ends with the rule we are investigating for coverage, and has a corresponding path in the invocation graph which starts at the graph's root node and ends at the investigated rule node.

This is done as follows: the `computeCoverageItems()` method (lines 8–15) iterates over all invocation graphs and—within those graphs—over all graph nodes, and creates a coverage item for each of the nodes. The `isCovered()` method is more complex: it first receives the invocation graph belonging to the coverage item's bigstep rule as well as all edges which might cover the given node. It then delegates the actual coverage check to the recursive `isOrdered()` method which can be seen at lines 25–37.

The idea of this method is to walk the invocation graph as well as the transition system "backwards" in parallel until a bigstep rule is found – if this succeeds, we know that there are consecutive transitions $t_1, \ldots, t_n$ in the transition system such that the corresponding nodes form a path in the invocation graph that starts with the bigstep rule and ends up with the node to be covered.

Lines 26–27 check whether the current rule node corresponds to the bigstep rule of the rule node we are checking for coverage – if this is the case, we are done and return `true`. Otherwise, we iterate over all edges which have our current rule node as target. For each of those edges, we again iterate over all incoming transitions. If we find a matching pair of invocation graph edge and graph transition, we do perform a step "backwards" (line 35). Note that the set `seenInvocationEdges` makes sure that the algorithm terminates, since every edge is considered at most once.

8.2.3.3 Edge Coverage Criteria

To motivate the need for edge coverage criteria, let us extend our example once more by adding a fourth invocation graph, i.e., the graph depicted as Fig. 8.12, and let us add a test model the transition system of which contains the sequence of transitions shown in Fig. 8.18. It is easy to see that the test model fulfills all rule coverage criteria, since it covers all nodes of the invocation graph. As such, our example now has a rule coverage of $\frac{13}{14}$=92.9%, a rule coverage plus of $\frac{15}{17}$=88.2%, and a rule coverage plus plus of $\frac{15}{19}$=78.9%.

However, consider the invocation graph of Fig. 8.12: The graph is completely covered by the single test model of Fig. 8.18, but there are a number of situations which are still not executed: What if rule destroy.2 is immediately executed after rule supplyStreamingToken.1? The test model also does not give rise to situations where rule destroy.1 is executed more than once.

This is where edge coverage comes into play. Covering an edge means to execute two rules consecutively: The edge's source and target rules. This allows to define finer coverage criteria than rule coverage.

The approach to defining the coverage criteria is similar to the rule coverage criteria: Let us define two edges e_1, e_2 to be *similar* if e_1's source (target) node corresponds to the same rule as e_2's source (target) node. The first edge coverage criterion, edge coverage, requires that each similar edge is executed by at least one of the test models. Edge coverage plus requires that for each invocation graph, one of the (possibly contained) similar edges needs to be executed by a test model. Finally, edge coverage plus plus requires that each edge has to be executed by a test model (including similar edges within an invocation graph).

In the following, we will discuss each edge coverage criterion, and we will provide pseudo-code showing how each criterion is computed.

Edge Coverage Let us illustrate the edge coverage criterion with an example. First of all, we note that the invocation graph of Fig. 8.12 does not contain any similar edges (neither within the rule's invocation graph nor when taking the other invocation graphs into account). Therefore, to achieve an edge coverage of 100%, we need to make sure that our example models execute all edges of that invocation graph, including the self edge of node destroy.1 and the edge between nodes supplyStreamingToken.1 and destroy.2 (these two edges are not yet covered, even with a rule coverage plus plus of 100%). As such, we can e.g. change our example model of Fig. 8.18 such that rule destroy.1 is executed two times before rule destroy.2 is executed, and we could add a second example model the transition system of which would contain the execution of supplyStreamingToken.1 and destroy.2 without destroy.1 in between.

The computation of edge coverage is straight-forward; it is depicted as Listing 8.7. As expected, the coverage items are edges of the invocation graph, and two edges are considered to be equal if their source and target nodes refer to the same rule. As such, the set of coverage items is computed by iterating over the invocation graphs, and to iterate over each invocation graph's edges – for each edge found, a coverage item is added to the result set (lines 9–13). Note that as always, a coverage item only gets added to the result set if the set does not yet contain an edge equal to the one to be added.

Listing 8.7: Subroutines for computing edge coverage

```
CoverageItem                                                          1
  Edge edge                                                           2
                                                                      3
  boolean equals(CoverageItem other)                                  4
    return edge.source.uniqueName = other.edge.source.uniqueName and  5
        edge.target.uniqueName = other.edge.target.uniqueName
                                                                      6
Set computeCoverageItems(Ruleset ruleset, Map invocationGraphs)       7
  Set result := new Set()                                             8
  for each (DirectedGraph invocationGraph in invocationGraphs.values()) 9
```

```
    for each (Edge edge in invocationGraph.edges)                          10
      CoverageItem item = new CoverageItem(edge)                           11
      if (not result.contains(item))                                       12
        result.add(item)                                                   13
  return result                                                            14
                                                                           15
boolean isCovered(GraphTransitionSystem gts, Map invocationGraphs, Map     16
    name2transitions, String coverageItem)
  Set sourceTransitions := name2transitions.get(coverageItem.edge.         17
      source.uniqueName)
  Set targetTransitions := name2transitions.get(coverageItem.edge.         18
      target.uniqueName)
  for each (GraphTransition sourceTransition in sourceTransitions)         19
    for each (GraphTransition targetTransition in targetTransitions)       20
      if (sourceTransition.target = targetTransition.source)               21
        return true                                                        22
  return false                                                             23
```

Coverage computation is done is follows: First, using the map computed by the generic coverage algorithm, all transitions corresponding to the edge's source and target nodes are received (lines 18–19). The algorithm then iterates over these transitions (lines 20–21) and returns `true` as soon as it has found two consecutive transitions which correspond to the edge's source and target nodes. If such a pair of transitions can not be found, the algorithm returns `false`.

Edge Coverage Plus The next example refers to the invocation graphs of Fig. 8.10 and Fig. 8.15. Edge coverage would only require two example models such that the first executes the edge between nodes execute.1 and executeHandlerBody.1, and the second would execute the edge between nodes execute.1 and executeHandlerBody.1. It would not matter whether these executions would start with rule start.1 (as in Fig. 8.10) or start.2 (as in Fig. 8.15).

In contrast, edge coverage plus requires that these edges are covered within each invocation graph. As such, we have to add example models such that the edges are covered between the two invocation graphs of Figs. 8.10 and 8.15, making sure that the execution works in the context of both bigstep rules.

Consequently, the definition of edge coverage plus, which is depicted as Listing 8.8, is rather similar to the one of edge coverage: The coverage item additionally stores the bigstep rule which is also considered for the coverage item's `equals()` definition (lines 1–6). The set of coverage items is computed by iterating over all edges of each invocation graph, where an edge is added to the result set only if no similar edge is already contained in that set.

Listing 8.8: Subroutines for computing edge coverage plus

```
CoverageItem                                                               1
  BigstepRule bigstepRule                                                  2
  Edge edge                                                                3
                                                                           4
  boolean equals(CoverageItem other)                                       5
    return bigstepRule.uniqueName = other.bigstepRule.uniqueName and       6
        edge.source.uniqueName = other.edge.source.uniqueName and edge.
        target.uniqueName = other.edge.target.uniqueName
                                                                           7
public Set computeCoverageItems(Ruleset ruleset, Map invocationGraphs)     8
  Set result := new Set()                                                  9
```

```
  for each (Rule rule in ruleset.rules)                                  10
    if (rule is BigstepRule)                                             11
      DirectedGraph invocationGraph := invocationGraphs.get(rule)        12
      for each (Edge edge in invocationGraph.edges)                      13
        CoverageItem item = new CoverageItem(rule, edge)                 14
        if (not result.contains(item))                                   15
          result.add(item)                                               16
  return result                                                          17
                                                                         18
boolean isCovered(GraphTransitionSystem gts, Map invocationGraphs, Map   19
    name2transitions, CoverageItem coverageItem)
  Set sourceTransitions := name2transitions.get(coverageItem.edge.       20
      source.uniqueName)
  Set targetTransitions := name2transitions.get(coverageItem.edge.       21
      target.uniqueName)
  for (GraphTransition sourceTransition : sourceTransitions)             22
    for each (GraphTransition targetTransition in targetTransitions)     23
      if (sourceTransition.target = targetTransition.source)             24
        Set invokingBigstepRules := new Set()                            25
        collectInvokingBigstepRule(gts, sourceTransition,                26
            invokingBigstepRules)
        for each (String bigstepRuleUniqueName in invokingBigstepRules)  27
          if (matches(bigstepRuleUniqueName, coverageItem.bigstepRule.   28
              uniqueName))
            return true                                                  29
  return false                                                           30
```

Coverage computation is then done exactly as in edge coverage, with one extension: If two consecutive transitions have been found such that they correspond to source and target of the edge (line 24), it must additionally be checked whether their transitions result from invocation of the coverage item's bigstep rule; the latter is done in lines 25–29.

Edge Coverage Plus Plus Finally, let us consider the invocation graph of Fig. 8.11 again, which contains two similar edges (the ones between the nodes notifySpawns.1 and moveOffers.1). For edge coverage plus it would suffice to cover one of those edges. To make sure that the consecutive execution of the two rules works correctly when either rule accept.1 or accept.2 are executed beforehand, we need to cover both edges within that invocation graph (and therefore fulfill the edge coverage plus plus criterion).

The items to be covered in the case of ege coverage plus plus therefore also consist of a bigstep rule and an edge (as it was the case for edge coverage plus). However, the definition of two coverage items being equal to each other is different (see Fig. 8.9): It not suffices any more that the edge's source and target nodes have the same unique names. Instead, the edges themselves must be different (line 6). The computation of the coverage items is then done exactly as with edge coverage plus (but will usually end up with more coverage items).

Listing 8.9: Subroutines for computing edge coverage plus plus

```
CoverageItem                                                             1
  BigstepRule bigstepRule                                                2
  Edge edge                                                              3
                                                                         4
  boolean equals(CoverageItem other)                                     5
    return edge = other.edge and bigstepRule.uniqueName = other.         6
        bigstepRule.uniqueName
```

```
                                                                      7
Set computeCoverageItems(Ruleset ruleset, Map invocationGraphs)       8
  Set result := new Set()                                             9
  for each (Rule rule in ruleset.rules)                               10
    if (rule is BigstepRule)                                          11
      DirectedGraph invocationGraph := invocationGraphs.get(rule)     12
      for each (Edge edge in invocationGraph.edges)                   13
        CoverageItem item = new CoverageItem(rule, edge)              14
        if (not result.contains(item))                                15
          result.add(item)                                            16
  return result                                                       17
                                                                      18
boolean isCovered(GraphTransitionSystem gts, Map invocationGraphs, Map  19
    name2transitions, CoverageItem coverageItem)
  DirectedGraph invocationGraph := invocationGraphs.get(coverageItem.   20
      bigstepRule)
  Set sourceTransitions := name2transitions.get(coverageItem.edge.     21
      source.uniqueName)
  Set targetTransitions := name2transitions.get(coverageItem.edge.     22
      target.uniqueName)
  for each (GraphTransition sourceTransition in sourceTransitions)     23
    for each (GraphTransition targetTransition in targetTransitions)   24
      if (sourceTransition.target = targetTransition.source)           25
        return isCovered(gts, invocationGraph, coverageItem.bigstepRule  26
            , coverageItem.edge.source, sourceTransition, new Set())
  return false                                                         27
```

Checking whether a coverage item is indeed covered again starts with searching two consecutive transitions corresponding to the edge's source and target nodes. If such transitions are found, the same technique is used as in rule coverage plus plus to make sure that indeed this very edge is covered: The invocation graph as well as the transition system are explored backwards until a transition is found which corresponds to the coverage item's bigstep rule, and from where on a sequence of transitions starts which finally ends up in the two transitions corresponding to the edge. For this, the `isCovered()` function of rule coverage plus plus is reused (i.e., the call of `getSorted` in line 26 in fact refers to the lower `isSorted()` function of Listing 8.6).

After having introduced the six coverage criteria, the next section will discuss how the criteria relate to each other.

8.2.4 Hierarchy of Coverage Criteria

The definition of the six coverage criteria introduced in the last section has shown that the criteria are related to each other. In this section, we will discuss these relations, and we will give an intuition of their expressiveness.

Before we do that, let us define that a coverage criterion C_1 *implies* a coverage criterion C_2 if a C_1 coverage of 100% implies a C_2 coverage of 100% – in other words: If the test models of a semantics specification fulfill C_1, C_2 can be taken for granted.

It is easy to see that rule coverage plus implies rule coverage and that rule coverage plus plus implies rule coverage plus: If all nodes corresponding to a single DMM rule are covered (rule coverage plus plus), then this implies that at least one such node is covered in each invocation graph (rule coverage plus), and if at least one node corresponding to the same DMM rule is covered per invocation graph, this implies that at least one such node is covered at all (rule

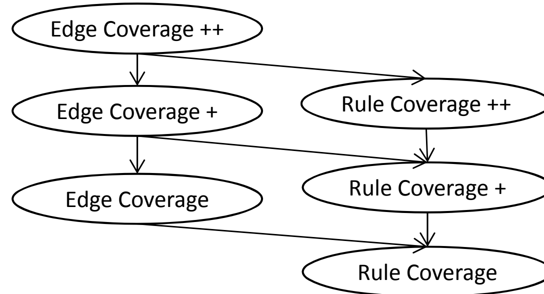

Figure 8.19: Hierarchy of coverage criteria.

coverage).

In the same way, we can see that edge coverage plus plus implies edge coverage plus, and that edge coverage plus implies edge coverage.

The relation between the rule and edge coverage criteria is only slightly more difficult to explain: Two edges are similar if their source and target nodes refer to the same rules. As such, edge coverage implies rule coverage: if all similar edges are covered at least once by our test models, than every rule must also be covered.

To make this even clearer, let us assume that we have an edge coverage of 100% and a rule coverage of less than 100%, i.e., there is an uncovered rule. Since rule coverage is defined over the rule nodes of the invocation graphs, and due to its definition, invocation graphs can not contain isolated nodes, it must be the case that there is an uncovered edge (since an edge is only covered if both its source and target rule nodes have corresponding transitions), contradicting our assumption of an edge coverage of 100%.

The same is true for edge coverage plus and rule coverage plus and for edge coverage plus plus and rule coverage plus plus. This results in the hierarchy of coverage criteria depicted as Fig. 8.19. The figure visualizes the relations between the criteria by containing an edge from criterion C_1 to C_2 if C_1 implies C_2. As such, one can easily see that e.g. edge coverage plus does not only imply edge coverage, but also rule coverage plus and rule coverage.

It remains to give the language engineer advice on how to choose appropriate coverage criteria: The more expressive a (fulfilled) criterion, the higher is the quality of the tests (and, hopefully,[9] the quality of the semantics specification). On the other hand, to fulfill a more expressive coverage criterion, more test models will in general be needed than for a less expressive coverage criterion. As such, it is important to choose an appropriate criterion.

It is immediately clear that a rule coverage of 100% should be the minimum requirement for every DMM specification – otherwise, there are rules which are not executed by any of our test models, and we do not have any confidence that these rules work as expected.

Despite that, our experience from testing several DMM semantics specifications [4] seem to imply that edge coverage plus is an appropriate coverage criterion in most cases. Our intuition behind this is that edge coverage is needed

[9] Of course, tests can only reveal existing problems, but they can not show the absence of problems.

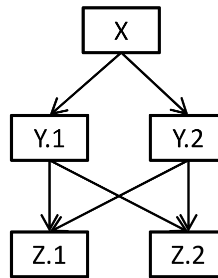

Figure 8.20: Invocation graph demonstrating the potential for dead edges.

in general because otherwise, cases such as recursive invocation of a rule are not covered at all (as we have seen with the invocation graph depicted as Fig. 8.12). However, each sequence of rule invocations starts with the invoking bigstep rule, and that rule needs to "set the stage" for the following invocations (e.g. by making sure that the invoked rules' application context exists – otherwise invocations will fail). Therefore, our experience is that it is a good idea to aim at edge coverage plus, which will make sure that we have test models for each edge and bigstep rule.

Finally, we want to point out that (similar to code coverage) a coverage of 100% might not always be possible. For instance, consider the invocation graph depicted as Fig. 8.20, which describes the situation where a DMM rule X invokes a rule Y for which two compatible rules exist, and where both Y.1 and Y.2 invoke a rule Z for which again two compatible rules exist.

Now, it might be the case that rules Y.1 and Z.1 are designed such that after Y.1 has been applied, only Z.1 can match, and that after Y.2 has been applied, only Z.2 can match. As a result, no test model will ever give rise to a transition system such that e.g. rule Z.2 is executed after rule Y.1. However, the coverage framework does not analyze the rules' structure – it only takes the rules' invocations into account. As such, no test model will ever be able cover e.g. the edge between nodes Y.1 and Z.2, and therefore, an edge coverage of 100% can not be achieved. It is the language designer's task to be aware of such design decisions, and to accordingly interprete the coverage rates computed by our framework.

8.3 Related Work

The existing work related to our approach of test-driven semantics specification can mainly be grouped into two categories: related test approaches and language engineering. For the former, the work most closely related to ours is the work by Sadilek et.al. [178]. His goal is to quickly prototype DSLs. The scenario is as follows: A language's semantics might first be specified using a formal language, e.g. *Abstract State Machines*, for the sake of proving properties of the DSL's semantics. Later on, a second, more efficient semantics specification might be created which shall be *semantically equivalent* to the first one. Since both semantics specifications allow for DSL instances to be executed, the language engineer can now create test models of the DSL, execute them and compare the resulting execution traces. The main difference to our approach is that Sadilek

uses tests to compare two semantics specifications, whereas we use them to convince ourselves that the semantics specification indeed produces the behavior the language engineer had in mind.

Another comparable approach is the so-called *scenario-based testing*. Xuandong et.al. [214] use UML Sequence diagrams to validate Java programs for safety consistency (sequences of method calls which must not occur during execution) and mandatory consistency (sequences of methods calls which have to occur). The main difference to our approach is that scenario-based testing focuses on testing a concrete object-oriented system, i.e., the communication between some objects, whereas we are testing semantics specifications describing the behavior of a complete language.

In the area of language engineering, several approaches for defining DSLs exist. For instance, MetaCase provides MetaEdit [188], Microsoft provides the DSL Tools as part of MS Visual Studio [28], and the Eclipse foundation provides the Graphical ModelingFramework [50]; all these approaches aim at an easy creation of visual languages. openArchitectureWare [92] provides a set of tools which allow for the easy creation of textual languages, including powerful editor support.

To our knowledge, all the above approaches focus on defining a DSL's behavioral semantics by providing support for code generation, but they do not provide a means to systematically create high-quality code generators; the generation is pretty much done ad-hoc.

The same holds for other semantics specification techniques which can be used in language engineering, e.g., the π calculus [143], Structural Operational Semantics [163], and others – we are not aware of a comparable test-driven process which helps to create high-quality semantics specifications.

Measuring test quality is a difficult task. One important approach is *mutation analysis*, where flaws are intentionally injected into a software system. The quality of the system's tests is then quantified as the relation between the number of flaws introduced and the number of these flaws actually discovered by the tests. For instance, Haschemi and Weißleder [94] present a generic approach to run mutation analysis, where the creation of mutants is separated from the mutation analysis execution environment. This approach could likely also be used in the context of DMM semantics specification. However, the metrics presented in this paper depends on the structural properties of DMM specifications (i.e. invocation structures); as a result, our framework is able to provide more concrete hints on how to improve test quality.

Our approach to measuring quality of tests is *test coverage*. Lots of research has been performed in this area – a comprehensive review would be out of scope if this thesis. See e.g. [147] for an introduction to software testing, including the definition of several coverage criteria for software systems.

Of particular interest in the context of this thesis is *model-based testing*, where different kinds of model coverage criteria have been defined (an introduction to model-based testing is provided e.g. in [201]). For instance, in [75], Friedman et al. describe different coverage criteria for state machines, which are then used to automatically generate and evaluate test cases. Another example is [72], where Ferreira et al. compute the coverage of UML activities by simulating them, and provide visual information on which parts of the activity have been covered.

On a more general level, Friske et al. [76] provide a framework for creating composed coverage criteria by combining different simple coverage criteria. They use OCL to formally describe the coverage criteria as well as the test goals, allowing to combine those formalizations into more complex ones. In [210], Weißleder and Schlingloff define new coverage criteria as a combination of advantages from condition- and boundary-based types, and use them to automatically generate test cases. The efficiency of the introduced coverage criteria is evaluated using mutation testing.

In contrast to our work, the above approaches define coverage criteria for executable models. They could thus be used to define and measure coverage of models the semantics of which is defined by means of a DMM specification, whereas we are interested in covering the DMM specification itself.

McQuillan and Power [136] examine white-box coverage criteria for model transformations specified with ATL [110]. In particular, they define the notions of *rule coverage* (which is very similar to our definition of rule coverage), *instruction coverage* and *decision coverage*, the latter roughly corresponding to the code coverage metrics *statement coverage* and *branch coverage*. This is the main difference to our work: Since ATL transformation rules are specified textually (in contrast to the visual DMM language), it is rather straight-forward for the authors to adjust existing coverage criteria for ATL transformations.

[209] measure the coverage of Tefkat [128] model transformations in terms of metamodel coverage: The transformation rules are analyzed for references to metaclasses, associations etc., marking the according metamodel elements as covered. The resulting coverage is not primarily meant as a quality measure of the model transformation, but as a means to give a quick overview of a model transformation's completeness with respect to the involved metamodels. In this sense, it could also be useful in the context of DMM, e.g. to measure the amount to which the semantics of a particular language has been specified.

Bauer et al. [17] are interested in testing chains of model transformations. Their approach is to compute a *footage* for each test case, which basically is a vector containing a coverage counter per coverage item. Defining a *distance* between two footages, they are then able to compare test cases with respect to similarity; e.g., two test cases with the exact same footage might test similar parts of the system, in which case one of the test cases could be removed (which still is to be decided by the modeler). On the other hand, if no test footage contains a coverage counter greater than zero for a particular coverage item, that item has not been excercised at all. The approach could be beneficial to DMM: Defining test case footages for the coverage criteria introduced in Sect. 8.2 might allow to define an appropriate similarity criterion for DMM test models, thus helping to keep DMM test suits reasonably small.

Summary of Part III

This part of the thesis was concerned with creating DMM semantics specifications of high quality. Our support of this goal is two-fold: First, we provide sophisticated tool support for the creation of DMM specifications. Second, we have suggested a test-driven development process for DMM specifications.

Chapter 7 introduced our tool support for creating DMM specifications. Part of this is the semi-automatic creation of executable mappings from syntax to runtime metamodels as needed for the DMM approach. As language for these mappings, we use DMM itself; thus, we first briefly discussed the usage of DMM in model transformation scenarios in Sect. 7.1; we then explained how to define a transformation from a syntax model into a runtime model in Sect. 7.2. We have suggested two approaches: In the decorator approach the runtime elements are kept in a different metamodel which references—and thus decorates—the syntax metamodel. In the from scratch approach, the runtime metamodel contains the syntax as well as runtime elements; runtime models are created from syntax models by a DMM based model transformation. We have described how to generate a base transformation from the syntax metamodel, which can then be customized for the sake of adding necessary runtime information.

Section 7.3 has given an introduction to the tool support for creating DMM specifications; we provide visual as well as tree-based editors for creating and editing DMM specifications, including validation of syntax and static semantics and annotation of problematic elements.

Chapter 8 has introduced our approach of test-driven semantics specification. Section 8.1 describes the approach itself; the idea is to first define example models of the target language, and to describe the expected semantics of those models by means of traces of execution events. Then, during the process of creating the DMM semantics specification, executable tests are automatically derived from the example models and their expected behavior; the tests check whether the semantics specification realizes exactly the desired behavior.

Section 8.2 defined a set of coverage criteria for tests of DMM specifications. The idea is to use the control structure implied by the rule invocation mechanism to derive a set of invocation graphs; the graphs describe potential orders of execution of smallstep rules. These graphs are then covered during the execution of the tests as described above, and the resulting coverage rate as well as te uncovered graph structures are reported. We have defined six coverage criteria which differ in expressiveness as well as computation complexity.

The results of this part are

- a test-driven process for the creation of high-quality DMM semantics specifications,
- coverage criteria for measuring the quality of a DMM specification's tests,
- rich tool support which enables the language engineer to create DMM specifications in a convenient manner, and to validate their syntactical correctness, and
- tool support for semi-automatically creating DMM-based model transformations from syntax into runtime models.

The results of this part thus enable us to turn to our final goal, analyzing and improving the quality of models, which is the topic of the next part.

Part IV

Quality of Models

9 Formulating and Verifying Requirements

In Part III, we have seen how to create a high-quality DMM semantics specification in a test-driven way. The current part will show how such semantics specifications can be used to verify the quality of *models*, which is our main goal when providing a formal semantics of a language. More precisely, given a set of *requirements* against a model, we will provide means to check whether they are indeed fulfilled by the given model.

Requirements are usually divided into two general kinds: *functional* and *non-functional* requirements. The former refers to the functions of the system: A functional requirement is fulfilled if the system—given an appropriate input—performs the correct behavior and produces the desired output. In contrast, non-functional requirements do not describe *what* the system shall do, but *how* it shall do this.

Let us illustrate the above with an example: Let us assume that an insurance company has described the workflow for dealing with a customer's insurance claim by means of a UML activity. In this context, a typical functional requirement would be that a second employee has to double-check the claim before money is sent to the customer; a typical non-functional requirement would be that on average, processing a customer's claim must not take longer than a certain period of time.

Given a model whose language is equipped with a DMM semantics specification, DMM allows to verify both functional and non-functional requirements against that model. Section 9.1 will show how functional requirements can be formulated visually and verified by using model checking techniques. Section 9.2 will then explain how to add non-functional properties to a DMM specification, and how these properties can then be used to analyze models with respect to e.g. average execution time.

9.1 Functional Requirements

One of the most important reasons for equipping a visual language with a formal semantics is that this allows to reason about the quality of models of that language in an automated way. In this section, we will show how this is done for *functional requirements*, which are the most important kind of requirements to be addressed by our models: If they do not produce the desired behavior, the models are pretty much useless.

As an example of functional requirements, we will use the workflow property

of *soundness* [203, 65], which basically states that a workflow should a) not contain any useless elements and b) should have a well-defined start and end. We will give a brief introduction to soundness in the next section.

Section 9.1.2 will then introduce the *Process Pattern Specification Language* (PPSL) [74, 73], a visual language dedicated to the formulation of behavioral properties, and EPPSL, an extension of that language [116].

As it turns out, the focus of (E)PPSL on business processes does not allow to directly apply the languages against other kinds of languages. Therefore, in Sect. 9.1.3 we show how we have generalized (E)PPSL to allow for the verification of temporal properties against arbitrary model states. The section is based on [190].

Finally, in Sect. 9.1.4 we use generalized EPPSL to formalize the soundness requirements as formulated in Sect. 9.1.1.

9.1.1 Example Requirement: Soundness

When modeling a business workflow, the business analyst will have certain functional requirements in mind. For instance, she wants to ensure that in case of a bicycle factory, each bicycle which is the result of the process will be tested before delivery. Such properties are specific for each individual workflow and are thus not reusable in most cases.

However, it would be useful to have general quality requirements for workflows, i.e., requirements every workflow should fulfill. Such requirements could then be formulated for once, and verified against each business process model to get at least a basic impression of its quality.

Soundness is such a requirement, which has been defined by van der Aalst [203, 204]. The basic idea is that each workflow should a) not contain any useless elements and b) should have a well-defined start and end.

For business process modeling, van der Aalst uses a syntactically restricted kind of Petri nets [162] called *workflow nets*. The restriction is that a workflow net must have exactly one *source* and *sink place*, the former having no incoming transitions, the latter having no outgoing transitions. A process is then started by putting a token onto the source place; the token represents some work to be processed and is routed through the workflow net until it ends up at the sink place.

In [65], we have transferred the soundness property into the world of UML activities. Thus, we do not present the formalization of workflow nets and their soundness here – the interested reader is pointed to [203]. Instead, we immediately turn our attention to soundness in the context of UML activities, which we have defined as follows:

1. The Activity must have exactly one `InitialNode` and `ActivityFinalNode`.

2. If a token arrives at the `ActivityFinalNode`, no more tokens are left in the Activity.

3. A token finally arrives at the `ActivityFinalNode`.

4. Any `Action` must be executed under at least one possible execution of the Activity.

The intuition of the above definition, which very closely follows the one for workflow nets, is as follows: Requirement 1 makes sure that the start and end of the activity are well-defined: The activity starts if a `Token` is put on its `InitialNode`, and it ends if a `Token` arrives at the `ActivityFinalNode`. Requirement 2 ensures that if the activity ends, no more work is left within the activity which still is to be processed. Requirement 3 makes sure that an activity always terminates, and finally, requirement 4 ensures that the activity does not contain any useless `Actions`.

In the next section, we will introduce the Pattern Process Specification Language (PPSL), a generalized version of which we will then use in Sect. 9.1.4 to formulate soundness with respect to UML activities.

9.1.2 Pattern Process Specification Language

In Chapter 8 we have seen how test-driven semantics specification (TDSS) makes use of so-called *execution events* to describe the expected semantics of test models, and how to translate traces of execution events into a certain kind of LTL formulas which are then verified against the models' transition systems. This approach contains all basic concepts needed to perform verification of functional requirements. However, it has a severe drawback: In the case of TDSS, the temporal logic formulas to be verified are generated by the test framework, which was possible because of the special sequential structure of the traces (and, thus, the resulting formulas). In contrast, in the case of arbitrary requirements the formulas would have to be created manually, a task which is expected to be beyond knowledge of the average business analyst.

The *Pattern Process Specification Language* (PPSL) [74, 73] has been developed to overcome a very similar issue. Its goal is to enable business analysts to formalize requirements against business processes in a visual, intuitive way. In the next section, we will give a brief introduction to PPSL.

However, the focus of the authors of the PPSL was to make their language as easy as possible to use. As a result, the language slightly lacks expressiveness, partly because some logical constructs are missing (e.g., it is not directly possible to express that something should *not* happen), partly because PPSL expressions are translated into the temporal logic dialect LTL [165] (which does not allow for the formulation of certain important requirements, as we have seen in Sect. 4.3.1).

Thus, an extension of the PPSL called EPPSL has been proposed in [116], which provides a more comprehensive set of constructs for formulation; additionally, EPPSL is translated to the temporal logic dialect CTL [60].

In the next section, we will introduce the PPSL, followed by EPPSL in Sect. 9.1.2.2.

9.1.2.1 PPSL

As we have mentioned above, the goal of the PPSL [73] is to enable business analysts (i.e., people not familiar with formalisms such as the temporal logic dialects CTL or LTL) to formulate requirements against their models in a formal way. This is achieved by providing a visual, easily understandable language for requirements formulation; sentences of that language are then automatically translated into according temporal logic structures.

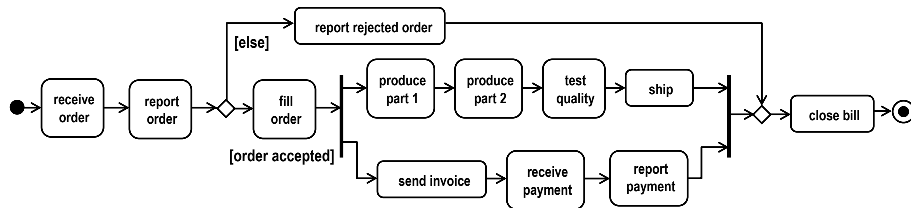

Figure 9.1: Example business process (from [74, p. 3]).

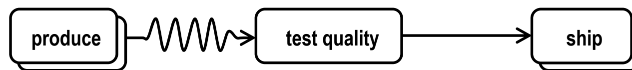

Figure 9.2: Example PPSL expression (from [74, p. 3]).

Let us investigate an example usage of the PPSL by first introducing a business process modeled as a UML activity, which is depicted as Fig. 9.1. The workflow models how a company deals with an incoming order by checking the order's validity – if the order is valid, the product is produced, tested, and delivered to the customer. In parallel, the payment belonging to the order is processed.

Now, one important requirement against the given business process can be formulated as follows: "After each production action a quality check has to be performed prior to delivery." However, it is not immediately obvious whether the process fulfills this requirement. Even if it was, more realistic business processes are expected to contain hundreds of steps, making it basically impossible to manually check requirements like the one above.

The PPSL can now be used to formulate the above requirement in a visual means; the resulting PPSL expression is depicted as Fig. 9.2 and shows the characteristics of the PPSL:

- To improve understandability for language users who are already familiar with UML activities, the PPSL reuses quite a number of elements from activities. In particular, a) the notion of rounded rectangles representing actions is reused, and b) these actions are connected by directed edges which are again reused from activities.

- Additionally, the PPSL introduces some custom language elements, the concrete syntax of which has been designed such that it is easily understandable. In particular, the PPSL uses a sidled arrow to express that two actions have to follow each other after an arbitrary (but finite) amount of time (see the arrow between "produce" and "test quality"), and it uses a "multi-action" representation to model that the requirements shall be true for all such steps of the workflow (see "produce" and "ship").

Therefore, the PPSL expression of Fig. 9.2 models the requirement presented above: The left part of Fig. 9.2 can be read as "all production action must be followed by a test quality action at some point in time", and the right part can be read as "all ship actions must be preceded by a test quality action".

We now show how PPSL expressions are translated into the temporal logic dialect LTL. For this, a number of translation rules are provided (see e.g. [73,

pp. 137f]) which map the visual elements of the PPSL into temporal logic counterparts (and thus define the semantics of PPSL). Let us again show this by means of our example: The left part of our requirement shall express that it must always be the case that if an item is produced, that item will be tested sometimes in the future. The resulting LTL formula is

$$\mathbf{G}(\text{produce} \Rightarrow \mathbf{F}(\text{test quality}))$$

The right part shall express that in all cases, the quality is tested immediately before shipping. The resulting LTL formula looks quite similar to the one above:

$$\mathbf{G}(\text{ship} \Rightarrow \mathbf{Y}(\text{test quality}))$$

The difference between the formulas is two-fold: First, the **F** expresses that the quality has to be tested at some point in the future (in contrast to **Y**, which would express that the test of quality had to be performed immediately before shipping, i.e., as the previous step of the process). Second, the **F** operator talks about the future, where the **Y** operator concerns the past.

Technically, PPSL is making use of the DMM specification for UML activities for the sake of describing the business processes' execution semantics. Thus, the last step in applying a PPSL expresssion consists of replacing the names of the actions occuring in our formulas with actual DMM rules against which we can perform model checking. Since rule action.start(name)# corresponds to the execution of an action, it suffices to replace each action name in the above formulas with the action.start(name)# rule we have seen earlier in this thesis, finally resulting in the LTL formulas

$$\mathbf{G}(\text{action.start(“produce”)\#} \Rightarrow \mathbf{F}(\text{action.start(“test quality”)\#}))$$

and

$$\mathbf{G}(\text{action.start(“ship”)\#} \Rightarrow \mathbf{Y}(\text{action.start(“test quality”)\#}))$$

The conjunction of the two formulas above formalizes our original requirement. It can now be model-checked using the GROOVE model checker – if the verification reveals that the formula indeed holds for a model's transition system, we can be sure that the model fulfills our requirement.

It remains to discuss the complete workflow of modeling and verifying a business process with the PPSL, which is depicted as Fig. 9.3. The upper part of that figure shows the "standard" DMM workflow: A DMM semantics specification as well as a model (in this case, a UML activity) are fed into the DMM generator component, which transforms the artifacts into a GROOVE grammar and computes the model's transition system.

The PPSL-specific steps can be seen at the bottom of Fig. 9.3: The requirements are formulated as PPSL expression and fed into the PPSL translator, which generates temporal logic formulas using the translation rules as described above (and replacing the actions' names with according instances of the rule action.start(name)#). Then, the GROOVE model checker receives the transition system as well as the temporal logic formulas and verifies whether these hold. If one of our requirements is violated, the model checker will provide a counter example which will (hopefully) prove helpful when fixing our model.

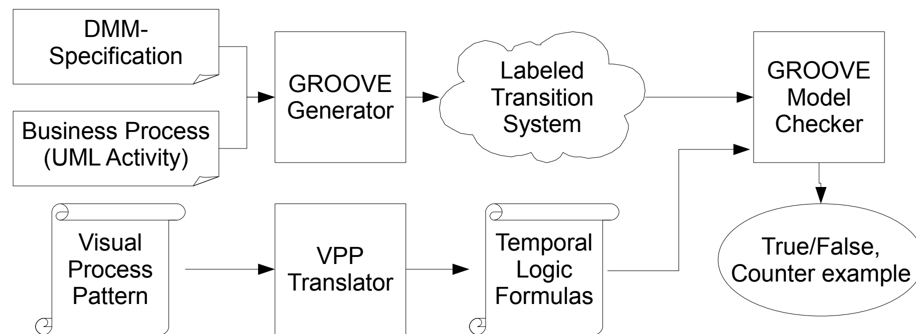

Figure 9.3: Overview of the PPSL approach (from [190, p. 3]).

Temporal Relationship	Notation	Temporal Relationship	Notation
Next		PossiblyNext	
After		PossiblyAfter	
Until		PossiblyUntil	
All		PossiblyAll	

Figure 9.4: Temporal operators provided by the EPPSL approach (from [116, p. 6]).

9.1.2.2 EPPSL

EPSSL is an extension of PPSL which adds support for more logical operators and translates to CTL. It has been proposed in [116].

The most important addition of EPPSL to PPSL is that it allows to formulate conditions which must only be true by some (but not all) possible executions of the business process under consideration. The temporal operators provided by EPPSL are depicted as Fig. 9.4; the figure's left column refers to temporal operators which must be true for all possible process executions, and the right column refers to temporal operators which must be true for at least one of the possible executions. For instance, the "Next" operator translates to CTL's **AX**, where the "Possibly next" operator translates to CTL's **EX**; "After" translates to **AF**, and "Possibly after" to **EF**; etc.

Additionally, EPPSL provides more possibilities to refer to the business process than PPSL does, and provides a means to hierarchically compose EPPSL expressions. The necessary building blocks are depicted as Fig. 9.5. For instance, the "InitialNode" building block refers to the start of a UML activity, and the "ConstraintContainer" allows to encapsulate EPPSL expressions for the sake of combining them to more complex expressions.

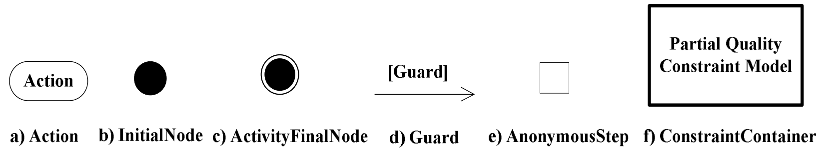

Figure 9.5: Basic building blocks provided by the EPPSL approach (from [116, p. 5]).

Logical Relationship	Notation	Logical Relationship	Notation
Join/ ForkNode	▌	Not	╳
Decision/ MergeNode	◇	Connector	——•

Figure 9.6: Logical operators provided by the EPPSL approach (from [116, p. 7]).

For the latter task, a number of logical operators exist, which are depicted as Fig. 9.6. These allow to combine "ConstraintContainers" with the "Join/-ForkNode" operator (logical `and`), the "Decision/MergeNode" operator (logical `or`), the "Not" operator, and the "Connector" operator (implication).

Let us now consider an example EPPSL expression, which is depicted as Fig. 9.7 and combines the different language elements introduced above. The requirement realized by the EPPSL expression is described as follows: "If there exists a possibility to make an interview, then there exists no possibility to accept the application online and there exists a possibility to make an interview per phone or there exists a possibility to make an interview per Internet" [116, p. 12].

The translation of EPPSL expressions into CTL is rather complex: Patterns of language elements are translated into CTL formulas – for some examples of patterns, see Fig. 9.8. For translating combinations of the patterns, different strategies are followed depending on the complexity of a given expression. We

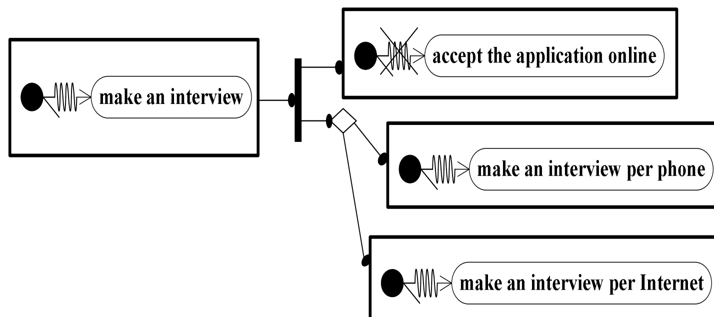

Figure 9.7: Complex example EPPSL expression (from [116, p. 13]).

EPPSL Pattern	CTL-formula
B A1 . . An	AG (B → (AF(A1) V .. V AF(An)))
C	(C*)
	→
C1 . . Cn	(C1*) Λ .. Λ (Cn*)
C1 . . Cn	(C1*) V .. V (Cn*)

EPPSL Pattern	CTL-formula
S	EF (S*)
Action S	Action → S*
S	AX (S*)
Action S	AG (Action → S*)
M	¬M*

Figure 9.8: Translation of EPPSL patterns into CTL formulas (from [116, p. 10]).

do not explain that translation here; the interested reader is pointed to [116].

For the sake of still giving an impression of the translation, we do show the translation result of the EPPSL expression depicted as Fig. 9.7.

$$\begin{gathered}
(\text{Start} \Rightarrow \mathbf{EF}(\text{make an interview})) \Rightarrow \\
((\text{Start} \Rightarrow \neg\mathbf{EF}(\text{accept the application online}))\wedge \\
((\text{Start} \Rightarrow \mathbf{EF}(\text{make an interview per phone}))\vee \\
(\text{Start} \Rightarrow \mathbf{EF}(\text{make an interview per Internet}))))
\end{gathered}$$

The translation result is straight-forward: The first line results from the left constraint container of Fig. 9.7. The "ForkNode" to the container's right models the logical `and`; as such, the top right container can be found in line 2, and the middle and lower container at lines 3 and 4.

The final step in applying EPPSL is the same as in PPSL: Within the formula, DMM rules have to be inserted which represent the according events. For the "Start" event, rule initialNode.flow()# is used; the references to actions such as "make an interview" are replaced by occurences of the rule action.start("make an interview")#. The final formula is then ready to be model-checked by GROOVE.

To recapulate, PPSL as well as EPPSL provide a convenient means to formulate temporal requirements against business processes modeled by UML activities, which can then be automatically translated into LTL/CTL formulas and model-checked by the GROOVE tool. However, both PPSL and EPPSL are dedicated to business process verification; they do not allow for the verification of arbitrary model properties. In the next section, we will show how we have generalized them to overcome this issue.

9.1.3 Generalizing (E)PPSL

In the last section, we have seen how functional requirements of business processes can be formulated by means of PPSL and EPPSL. However, we have also seen that both languages only allow to formulate expressions about executions of certain elements of UML activities – the languages do not allow for the formulation of requirements over arbitrary states of execution of our models. In this section, we show how we have generalized (E)PPSL to improve on this situation.

Before we dive into this, let us recall the general idea of model checking as seen in Chapter 4: A GROOVE transition system consists of states and transitions, where each state is a typed graph, and two states s, s' are connected by a transition t if there exists a rule r such that r matches s, and the application of r to s results in s' – in this case, t is labeled with r's name. GROOVE does not directly allow to model check against properties of a transition system's states; instead, GROOVE allows to model check against the transitions' labels. However, this allows to indirectly reason about the states: If r matches s, then we have knowledge about s (in particular, we know that s contains an object structure such that r matches).

Of course, also (E)PPSL makes use of this fact. However, since the goal was to be able to specify and verify requirements for business processes only, PPSL makes use of the action.start(name)# rule only (matching of this rule corresponds to the execution of the according action). EPPSL improves on that by allowing to formulate expressions over more UML activity constructs such as the `InitialNode` and `FinalNode`. This has the advantage that language users do not even have to know that there are rules which correspond to the according events – they only need to provide the business process and the requirements (which are formulated in a language very close to the one the process is formulated in).

However, our goal is to specify requirements against all kinds of languages (not only UML activities), and we want to be able to specify those requirements over all language elements of the target language. For instance, neither PPSL nor EPPSL allows for the formulation of requirements against, say, states of a UML state machine.

Now, recall from Sect. 8.1 that during creation of a DMM ruleset, execution events are identified. These are selected such that they suffice to describe the test models' expected behaviors, and are thus dedicated for formulating temporal properties about models.

Therefore, the first step of generalizing (E)PPSL is to bind the "nodes" occuring in the (E)PPSL expression not to a fixed rule (for instance, rule action.start(name)# or rule initialNode.flow()#), but to an arbitrary execution event of the DMM semantics specification of the language at hand.

Let us illustrate this using the DMM semantics specification for UML state machines as developed by Nesterow as part of his diploma thesis [149]. The specification consists of 12 bigstep and 77 smallstep rules organized in 9 packages. During creation of the ruleset, Nesterow identified and formalized the language's execution events; in particular, he identified an execution event corresponding to executing a state's associated behavior, which he called "StateExecutes(name)".

In generalized (E)PPSL, we can now make use of this execution event: We bind it to a node by displaying the execution event within the node (instead of

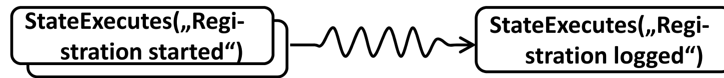

Figure 9.9: Example EPPSL for state machines, referring to execution event "StatExecutes(name)".

only the name of the according action). The resulting nodes can then be used in (E)PPSL expressions as usual (E)PPSL nodes, since the backing mechanism does not change: In (E)PPSL, nodes are bound to fixed rules of the activity semantics specification (and the rules' names are used within the generated temporal logic formulas). These rules are just replaced by the specific rules bound to the execution events at ruleset definition time.

An example generalized EPPSL expression is depicted as Fig. 9.9. It models that as soon as a given state machine enters the state where a registration process is started, the registration will always be logged at some point in time (no matter whether the registration succeeded or was canceled by the user). The expression translates to

$$\mathbf{AG}(\text{StateExecutes(“Registration started”)}$$
$$\Rightarrow \mathbf{AF}(\text{StateExecutes(“Registration logged”)}))$$

However, this is still not good enough. Suppose that we want to model that as soon as a state machine is started, it shall never be in states "A" and "B" at the same time. For referring to the start of a state machine, we can use execution event "StatemachineStarts()", which is part of the semantics specification. However, there is no execution event which corresponds to being in two states at the same time (nor can there be such an execution event, since execution events are generic and can not reflect properties of a certain model).

The solution are DMM property rules. Recall from Sect. 6.2.2.8 that a property rule has similar left-hand and right-hand graphs and thus does not change the states a property rule matches and is applied to – instead, a property rule results in a self-transition of the according states. Thus, we can add property rules to an existing DMM specification without changing the modeled behavior. The property rule state.A_and_B()! realizing our current requirement is depicted as Fig. 9.10. It matches if and only if both `States` "A" and "B" carry a `Marker` and are thus active, as desired.

We can now formulate our above requirement by referring to the added property rule. The resulting EPPSL expression is depicted as Fig. 9.11. It models the desired requirement: As soon as the state machine has started, the property rule must never match in the future. The expression translates to

$$\mathbf{AG}(\text{StatemachineStarted()}$$
$$\Rightarrow \neg\mathbf{EF}(\text{state.A_and_B()!}))$$

To summarize, PPSL as well as EPPSL allow for the convenient, visual formulation of temporal requirements against business processes modeled as UML activities. We have generalized them such that we allow to bind arbitraty execution events or property rules to nodes of (E)PPSL expressions; by making use of property rules (see Sect. 6.2.2.8 on page 68), we are able to formulate

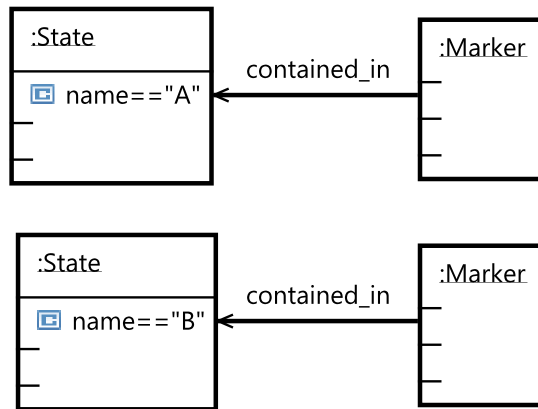

Figure 9.10: Example property rule state.A_and_B()! for state machines.

Figure 9.11: Example EPPSL for state machines, referring to the property rule of Fig. 9.10.

temporal requirements against arbitrary model states. In the next section we will use generalized EPPSL to formulate the soundness property.

9.1.4 Formalizing and Verifying Soundness

It remains to show how soundness as defined in Sect. 9.1.1 on page 184 can be formalized and verified. First of all, requirement 1 is a syntactical one and is therefore easy to check by analyzing the syntactical representation of a given model: It suffices to traverse the abstract syntax of the model, counting the objects of type `InitialNode` and `ActivityFinalNode` of an `Activity` instance. If the analysis reveals that exactly one of both node types is contained in the model, the requirement is fulfilled.

The other requirements are more complicated to check since the actual behavior of the model has to be taken into account. Let us start with requirement 4, which states that the activity must not contain any useless `Actions`. The requirement can be verified as follows:

1. Traverse the model's abstract syntax and collect the names of all `Actions` occuring in the activity.
2. For each action name:
   (a) Create a generalized EPPSL expression such as the one depicted as Fig. 9.12, where "<action name>" is replaced by the name of the according action.
   (b) Verify the created EPPSL expression, making use of the GROOVE model checker.

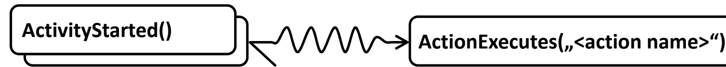

Figure 9.12: Generalized EPPSL expression realizing requirement 4 on page 184.

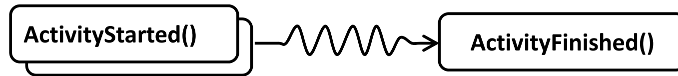

Figure 9.13: Generalized EPPSL expression realizing requirement 3 on page 184.

(c) If the verification fails, report the action as unused (and therefore superfluous).

The generalized EPPSL expression of Fig. 9.12 translates to

$$\mathbf{AG}(\text{ActivityStarted}()$$

$$\Rightarrow \mathbf{EF}(\text{ActionExecutes}(\text{“<action name>”})))$$

Let us now turn our attention to requirement 3, which states that an activity shall always terminate. Using a generalized EPPSL expression, this can now easily be formulated. The EPPSL expression is depicted as Fig. 9.13. It states that for all possible executions of the model, the execution event “ActivityFinished()” has to occur at some point in time, corresponding to the fact that the activity will end its execution.

Finally, let us investigate requirement 2, which states that the moment a `Token` arrives at the activity's `ActivityFinalNode`, no other `Tokens` are allowed to flow through the activity. Since we do not have any DMM rule in the activity semantics which corresponds to this very situation, we have to make use of an according property rule.

The property rule finalNode.ERROR()! containing according object structure is depicted as Fig. 9.14. The rule matches any situation such that an incoming `ActivityEdge` of an `ActivityFinalNode` carries an `Offer` (and—transitively—a `Token`), and if there exists at least a second `Token` somewhere in the activity.[1] The latter holds because the according `Token` object, which can be seen in the right upper corner of Fig. 9.14, is not associated to any other element – in other words: The rule does not make any statement about that `Token` despite that it must exist at all.

Referring to the newly introduced property rule finalNode.ERROR()! now allows to formulate requirement 2 on page 184 – the resulting EPPSL expression is depicted as Fig. 9.15.

The expression is true for a transition system which does not contain any reachable transition labeled with the property rule's name. An activity giving rise to such a transition system will never produce a situation such that a `Token` arrives at an `ActivityFinalNode` and one or more other `Tokens` are still

[1] Recall from Sect. 6.3.2 that DMM rules match injectively, which means that for the rule to match, at least two tokens must exist.

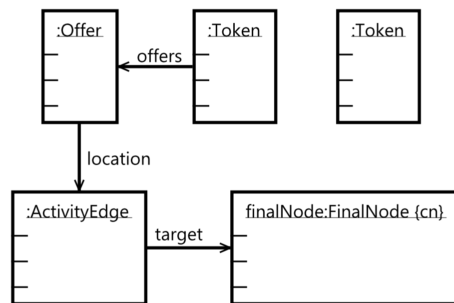

Figure 9.14: DMM property rule finalNode.ERROR()! used to formulate requirement 2 on page 184.

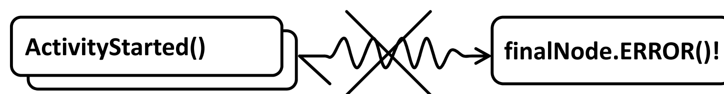

Figure 9.15: Generalized EPPSL expression realizing requirement 2 on page 184.

flowing through the activity. In other words: The according activity fulfills requirement 2 on page 184.

Note that the property rules used for defining soundness do not have to be added to the DMM ruleset specifying the semantics of UML activities, which would "pollute" that semantics specification. Instead, the property rules can be contained in a different ruleset dedicated to soundness verification, and the ruleset containing the semantics rules can be imported into that ruleset (see Sect. 6.2.1.2).

9.2 Non-Functional Requirements

In the last section, we have seen how functional requirements against our models' behavior can be formulated and verified. In this section, we will show how basic nonfunctional requirements can be formulated, and how the models can be analyzed with respect to these requirements.

As an example, we will use a business process modeled by means of a UML activity. We will introduce that example model in the next section, and we will discuss a scenario where—given a fixed budget—the average throughput of the business process can be improved as much as possible.

The actual analysis of the nonfunctional requirements makes use of the Performance Evaluation Process Algebra (PEPA) [100, 101]. It is performed in three steps:

1. First, performance information is added to the given combination of model and semantics specification, e.g., information about the expected runtime of actions of our bussiness process needs to be modeled.

2. Using that performance data, a PEPA model is generated.

3. Finally, the PEPA model can be analyzed by means of existing PEPA tooling.

Section 9.2.2 will give a brief introduction to the PEPA concepts and tools. Based on that, we will show in Sect. 9.2.3 how performance information can be added to a given DMM ruleset. We will also see that in most cases, only considering the ruleset itself will not suffice to add appropriate performance information; in these cases, we will also have to consider the concrete model the nonfunctional properties of which we are interested in.

Finally, in Sect. 9.2.4 we will add performance information to our example model of Sect. 9.2.1, and will analyze the model's nonfunctional properties and identify the best way to improve the model's average throughtput.

The concepts presented in this section have been implemented within the plug-in `de.upb.dmm.performance`.

9.2.1 Example: Process Improvement with Fixed Budget

In this section, we will introduce our example scenario for analyzing non-functional properties. First, an example model is shown as Fig. 9.16 – the given UML activity models a business process in the insurance domain. Let us investigate the activity more closely. The activity's name is "Process Claim". It contains several `Actions` such as "Get Insurance Info", "Check Validity" etc. which are connected with `ActivityEdges`. The activity starts as soon as a `Claim` arrives at the `ParameterNode` on the activity's left side. The black bar following the `ParameterNode` is a `JoinNode`: It makes sure that the processing of the claim as well the updating of the statistics are performed in parallel.

In the activity's main part, the claim is investigated: Is it a valid claim (e.g., does the customer have an insurance policy covering the claim)? How big is the probability that the incident has happened as described in the claim? Depending on the answers of those questions, the claim will be routed through the activity, finally resulting in the letter of rejection or the sending of the money as mentioned above. If, however, the claim turns out to be rather complicated, or if the information necessary to process the claim is not currently available, the handling of the claim will be scheduled for later processing.

We can now present our example scenario as follows: The above business process was in place for quite some time, which allowed to monitor the process's execution. In particular, one result of that monitoring is that the average execution times of the process's tasks (i.e., the UML actions) have been determined; these average execution times are depicted as Table 9.1.

Our goal is to improve the average execution time of the business process. For this task, a fixed budget is available, which only allows to perform certain changes on the process. In particular, we can use the given money to realize one of the following alternatives:

1. Reduce the average execution time of task "Check probability" from 15 to 10 minutes.

2. Reduce the average execution time of task "Calculate amount" from 10 to 6 minutes.

3. Reduce the average execution time of task "Verify Calculation" from 10 to 6 minutes.

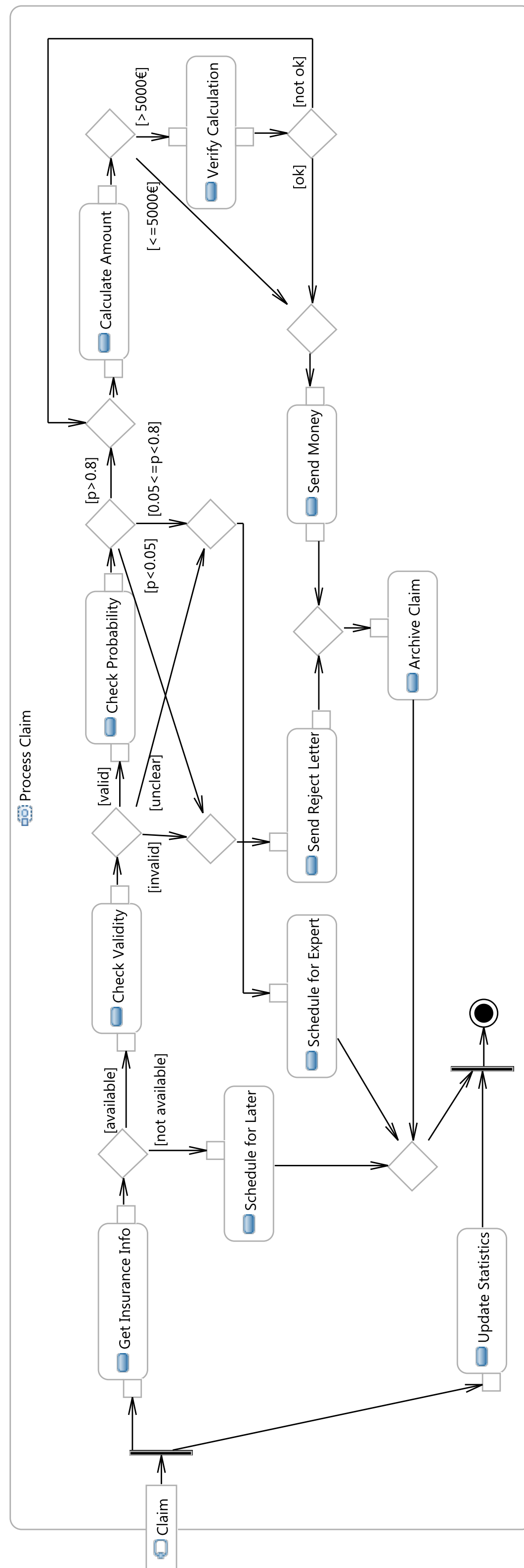

Figure 9.16: Example business process modeled as a UML activity.

Task	Avg execution time
Get Insurance Info	2 mins
Check Validity	5 mins
Check Probability	15 mins
Calculate Amount	10 mins
Verify Calculation	10 mins
Send Money	3 mins
Send Reject Letter	4 mins
Archive Claim	2 mins
Schedule for Expert	1 min
Schedule for later	1 min
Update Statistics	5 mins

Table 9.1: Average execution times of the tasks of the business process depicted as Fig. 9.16

As a consequence, we need to find out which of the above changes will result in the greatest improvement of our process's average execution time. We will tackle this problem by modeling the performance information for the three alternatives, and to then compute the resulting process's average execution time. Before we do that, let us first give a brief introduction to PEPA in the next section.

9.2.2 Performance Evaluation Process Algebra

Process algebras such as *Communicating Sequential Processes* (CSP, [102, 103]), the *Calculus of Communicating Systems* (CCS, [141]), or its continuation, the π-calculus [142] provide means to model and analyze concurrent systems. The general idea is that the interaction of independent processes is modeled, i.e., the communication between those processes as well as their synchronization. The resulting models can be evolved by means of algebraic operations.

An extension of process algebras are *stochastic process algebras*, which do not only contain information about what the modeled system will do, but additionally provide timing and probability information. In addition to the capabilities of simple process algebras, this allows to reason about several performance properties of the modeled system such as *steady-state solution* (the probability that the modeled system will be in a certain state) or *transition throughput* (long-term frequencies at which events occur in the modeled system).

The *Performance Evaluation Process Algebra* (PEPA) is a stochastic process algebra. It has been developed by Jane Hillston [100, 101] with the goal of providing a formalization of concurrent processes. For this, each action of the process is associated with a *rate*, i.e., a numerical value which describes how often an action is expected to be executed within a fixed period of time. Formally, each action's rate is interpreted as parameter of a negatively exponentially distribution (see e.g. [101]), allowing to translate a PEPA process into an equivalent *Continuous Time Markov Chains* (CTMCs, see e.g. [187]), for which powerful analysis techniques based on simple linear algebra exist.

The PEPA language consists of a small set of combinators. Sentences of the PEPA language describe the behavior of the modeled system by means of the performed tasks and their interactions. The grammar which formally defines

the syntax of the PEPA language is depicted as Fig. 9.17. The semantics of the combinators is defined as follows:

$$
\begin{array}{lcll}
S & ::= & & \text{(sequential components)} \\
 & & (\alpha, r).S & \text{(prefix)} \\
 & | & S + S & \text{(choice)} \\
 & | & C_S & \text{(constant)} \\
 & & & \\
P & ::= & & \text{(model components)} \\
 & & P \underset{L}{\bowtie} P & \text{(cooperation)} \\
 & | & P/L & \text{(hiding)} \\
 & | & C & \text{(constant)}
\end{array}
$$

Figure 9.17: The syntax of PEPA.

Prefix: The most important construct for describing the behavior of a system is to sequentially compose the behavior by means of actions and their given rate. For instance, the process $(\alpha, r).S$ describes that action α is executed, and that action has a rate of r; afterwards, the modeled system will behave as S. Sequences of actions can be combined to build up a life cycle for a component.

Choice: The choice combinator describes alternative behaviors the system may perform. For instance, the PEPA expression $P + Q$ represents a system which may behave either as P or as Q.

Constant: Constants can be used to define a behavior at some place and reuse that behavior somewhere else.

Hiding: The hiding operator allows to declare some behavior as internal to a process. For instance, the PEPA expression P/L describes that the set of actions L are to be considered internal or private. Note that this does not affect the actions' rates.

Cooperation: Most systems consist of several components which interact with each other. In PEPA, direct interaction, or *cooperation*, between components is modeled by the butterfly combinator ($\bowtie$). The subscript L of the cooperation symbol denotes the actions on which the cooperands are *forced* to synchronize. This means that for actions not in L, the components proceed with their work in an independent concurrent manner; if a component's action α is to be executed which is contained in L, execution stops until other components which have α as part of their actions are also ready to execute α. Thus, if L is empty, the two components work completely independently.

To make the PEPA concepts even more clear, let us examine an example PEPA process, which is depecited as Fig. 9.18. The process's initial state is `S1`. In that state, action `a1` is applied with a performance rate of `r1`, bringing the

```
S1 = (a1, r1).S2;
S2 = (a2, r2).S3 + (a3, r3).S4;
S1;
```

Figure 9.18: Example PEPA process.

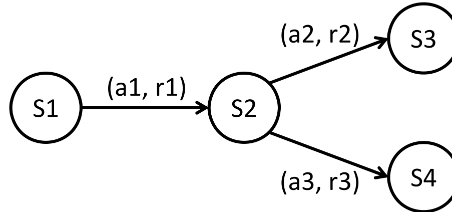

Figure 9.19: Example PEPA process (graph representation).

system to state S2. Here, two alternative executions are possible: The system will either execute action a2 with rate r2, or action a3 with rate r3.

9.2.2.1 From GROOVE to PEPA

Now that we have an understanding of PEPA and it's syntax and semantics, let us establish the connection between PEPA models and GROOVE labeled transition systems as described in [6]. The idea is to not specify a PEPA model directly, but to generate it from the labeled transition system GROOVE produces from a DMM specification and an according model.

To grasp this idea, let us consider a visual representation of the PEPA process depicted as Fig. 9.18, which is depicted as Fig. 9.19. It is immediately clear that the data structure of the PEPA model is very similar to an LTS as we have seen in Chapter 4. More precisely:

- Both the nodes in PEPA processes and the nodes of labeled transition systems correspond to states of the modeled system. The difference is that in PEPA, nothing is said about the state – two states differ if they are the result of different sequences of actions. In contrast, as we have seen in Chapter 4, GROOVE state nodes are typed graphs describing the complete current state of the system under consideration.

- In PEPA, two states are connected by an application of one of the system's actions - the resulting transition is labeled with the action's name and the rate of that action. In GROOVE, the situation is very similar: Two nodes of the LTS transition are connected by a transition if there exists a rule which matches the first state, and which—when applied—results in the second state; the resulting transition is labeled with the rule's name.

As such, the generation of a PEPA model from a transition system as produced by GROOVE is rather straight-forward – the algorithm is depicted as Listing 9.1. It receives an LTS as input and outputs a text file representing the PEPA model.

The algorithm implements a breadth-first traversion of the LTS, starting with the LTS's start state – the states to be processed are stored in a Stack

(line 2). To be able to deal with loops, the algorithm stores the states which have already been processed in a `Set` (line 3). Then, the LTS's start state is pushed onto the stack; the stack is processed in lines 5 and 6. If the stack (i.e., all states) has been processed, the initial start state is defined as the starting point of the PEPA process.

Method `processState` takes care of generating the necessary PEPA constructs for each state. First, the current state is added to the already processed states (line 10). Then, the state's number is used to generate a state label in line 11. Next, the algorithm iterates over the outgoing transitions of the current state – if there are none, then a final state has been found which has to be connected with the start state.[2] Otherwise, for each transition, an according PEPA sequence is generated (line, 17). Finally, if the target state of the processed transition is neither contained in the states to be processed nor in the already processed states, it is pushed onto the stack for later exploration.

A transition is mapped to a PEPA action in lines 24–26: Taking the transition and the transition's label into account, the performance rate is computed (line 25). Then, in line 26 the actual PEPA contructs are written, where the transition label is used as the action, and the resulting action/rate specification is sequentially composed with the target state of the transition in line 26.

Listing 9.1: Algorithm for generating PEPA model from transition system

```
void generatePepaModel(GraphTransitionSystem gts)                          1
  Stack statesToBeProcessed := new Stack()                                 2
  Set processedStates := new Set()                                         3
  statesToBeProcessed.push(gts.startState)                                 4
  while (not statesToBeProcessed.isEmpty())                                5
    processState(statesToBeProcessed.pop())                                6
  output("\nS" + gts.startState.number)                                    7
                                                                           8
void processState(GraphTransitionSystem gts, GraphState state, Stack       9
    statesToBeProcessed, Set processedStates)
  processedStates.add(state)                                               10
  output("S" + state.number + " =\n")                                      11
  Set transitions = gts.getOutgoingTransitions(state)                      12
  if (transitions.isEmpty())                                               13
    output("    ('RESTART', infty).S" + gts.startState.number)             14
  else                                                                     15
    for each (GraphTransition transition in transitions)                   16
      writeTransition(transition);                                         17
      if (not processedStates.contains(transition.target) and not          18
          statesToBeProcessed.contains(transition.target))
        statesToBeProcessed.push(transition.target)                        19
      if (more transitions to be processed)                                20
        output(" +\n")                                                     21
  output(";\n")                                                            22
                                                                           23
void writeTransition(GraphTransition transition)                           24
  double rate := computePerformanceRate(transition)                        25
  output("    ('" + transition.label + "', " + rate + ").S" +              26
      transition.target.number)
```

In [6], the authors have associated each GROOVE rule with a fixed performance rate; as such, the computation of the rate as seen in line 26 comes down to a lookup in a map containing the rules' labels and associated performance

[2] This is because PEPA assumes a strongly connected graph as input; see e.g. [100]

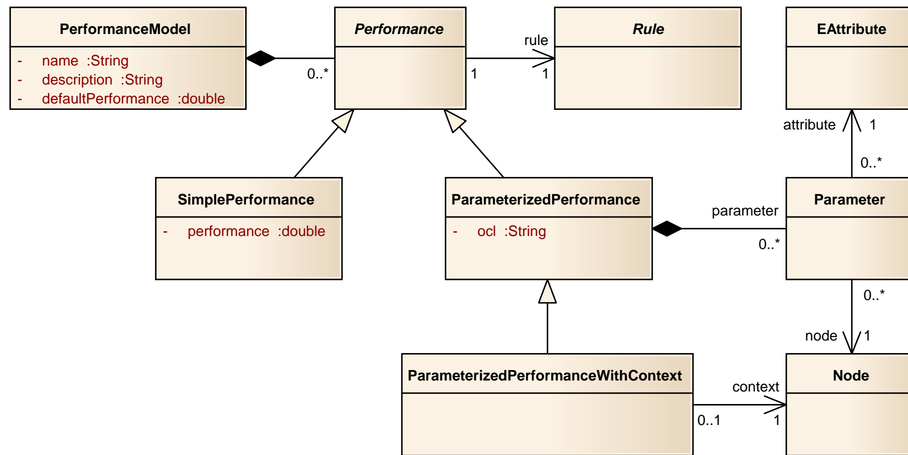

Figure 9.20: Performance metamodel of DMM.

rates. In DMM, more sophisticated approaches for specifying performance rates are provided, as we will see in the next section.

9.2.3 Modeling Performance Information in DMM

In this section, we will explain how performance rates are defined for DMM specifications in a model-driven way. We start by introducing the metamodel for defining performance rates in the next section. Sections 9.2.3.2 to 9.2.3.4 will then present the three different kinds of performance definition within DMM.

9.2.3.1 DMM Performance Models

As we have seen in Sect. 6, the DMM rules are the actual places where behavior is going on. Consequently, our goal is to associate each rule with a performance rate. This is done using a *performance model*, the metamodel of which is depicted as Fig. 9.20.

The general structure of the performance metamodel is straight-forward: a `PerformanceModel` contains a number of `Performances` (the different kinds are discussed below). Each `Performance` is associated to a DMM rule for which it defines the performance rate. The `PerformanceModel` class also allows to provide a default performance rate which will be used for DMM rules for which no other performance rate has been defined.

We now turn to the different kinds of performance definitions, starting with rule-based performance definition.

9.2.3.2 Rule-Based Rate Definition

The most simple case of performance definition is that a rule will always take about the same time to be executed, independent from the context the rule is applied in. To model such cases, the performance metamodel contains the concept of a `SimplePerformance` whose only feature is an attribute with name `performance` and type `double`. The effect is the same as in [6]: A `SimplePerformance` associates a rule with a fixed performance rate.

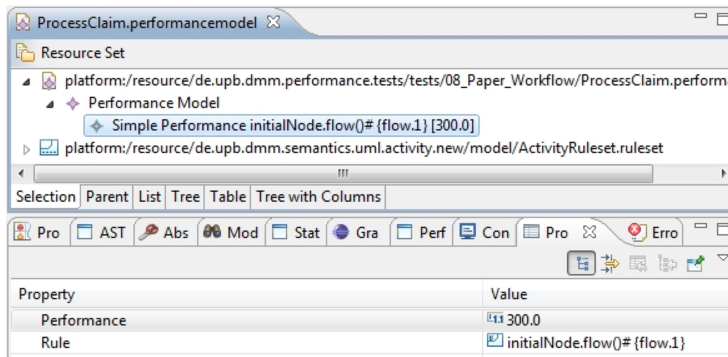

Figure 9.21: Performance model containing a single `SimplePerformance` for rule initialNode.flow()#.

Figure 9.21 shows a performance model containing a single `SimplePerformance`. The target rule of this performance definition is rule initialNode.flow(), and that rule is given a performance rate of 300.0.

9.2.3.3 Parameterized Rate Definition

As mentioned in the last section, `SimplePerformance` rate definitions are well suited for situations where the DMM rule (i.e., the event of the reality to which the DMM rule corresponds) always has the same average duration, as it might be the case for the initialization of our "Process Claim" activity. However, often the context in which a rule is applied has to be taken into account when computing the performance rate of a rule.

As an example, let us consider rule action.start(name)#, which in reality corresponds to the execution of one of the tasks contained in our process. Obviously, these tasks will not all have the same average duration: Some of them (like "Schedule for Later") will be performed quite quickly while others (like "Check Validity") will take much more time.

Therefore, the performance metamodel contains the concept of a `ParameterizedPerformance`, the idea being that parameters are passed to the rate definition which can be used to decide how long the rule will take. A `ParameterizedPerformance` contains a number of `Parameters`, each of which refers to a `Node` of the associated rule and an `EAttribute`. Finally, the rate definition contains an `ocl` attribute which is used to store the OCL expression computing the actual rate. In that expression, the parameters can be used to take the context of the rule application into account.

In our example, the (single) parameter is the name of the action in the context of which the rule is applied. Figure 9.22 shows an example for such a performance definition. It refers to rule action.start(name)# and contains a single parameter. The parameter refers to the `name` attribute of the rule's node action. That node has type `Action` (we have seen in Chapter 3 that UML actions indeed have such an attribute).

The most interesting part of the performance definition is the OCL expression. It consists of a number of nested `if-then-else` statements, each statement referring to a particular action's name. Within the expression, the

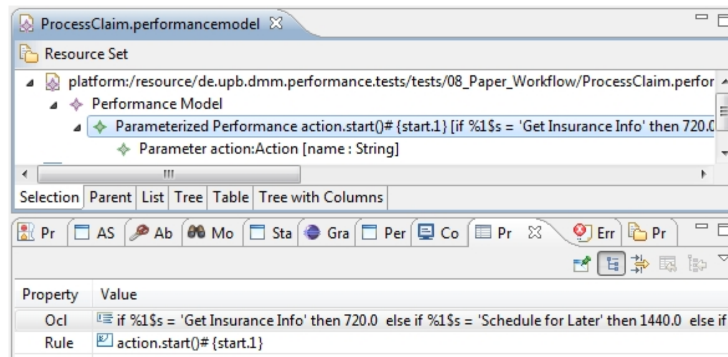

Figure 9.22: Performance model containing a single `ParameterizedPerformance` for rule action.start(name)#.

parameters can be referenced; the syntax is that of the Java `String.format(String, Object...)` method, e.g., the first parameter is referenced by `%1$s`. We will see below how the parameter references will be bound to attribute values during generation of the PEPA process.

`ParameterizedPerformance` rate definitions are a powerful tool. However, we will see in the next section that in some cases we need even more information to be able to compute a meaningful performance rate.

9.2.3.4 Parameterized Rate Definition with Context

DMM semantics specifications such as the one for UML activities are designed to be generic, i.e., they describe the semantics of *all* UML activities. As such, the DMM rules of the activities semantics specification can only refer to concepts of the runtime metamodel (which contains the classes of the static activity structure as seen in Fig. 3.3 on page 20 plus some generic runtime concepts such as `Token`, `Offer` etc.).

As the modeler, we are interested in analyzing a specific model, and we have more knowledge about that model than can be contained in the semantics specification. For instance, we know that an object of type `Claim` is passed to our "Process Claim" activity. However, since none of the DMM rules "knows" about the `Claim` concept, and since our `ParameterizedPerformances` can only refer to attributes of rule nodes, there is no way to refer to attributes of the `Claim` object within a `ParameterizedPerformance` rate definition.

We solved this problem by introducing the concept of a `ParameterizedPerformanceWithContext`. The according metaclass inherits from `ParameterizedPerformance` and therefore has all its references and attributes. Additionally, the `ParameterizedPerformanceWithContext` class refers to a `context` object of type `Node`. In the OCL expression, this object will be bound to the `self` object. It can then be used to (indirectly) access attributes of objects not having a corresponding node in the rule.

Figure 9.23 shows an example of such a rate definition. As the modeler, we know that a `Claim` object is flowing through our activity, and that it has an attribute `size` describing the size of the actual claim (e.g. in terms of pages, the number of words, etc.). We can now make use of this knowledge by declaring

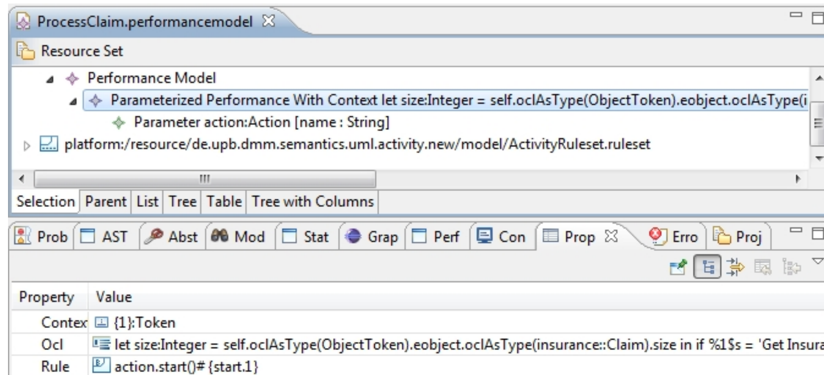

Figure 9.23: Performance model containing a single `ParameterizedPerformanceWithContext` for rule action.start(name)#.

the rule's `Token` object as the context object, and by then using that object within the OCL expression to access the attribute we are interested in: The OCL expression

```
let size:Integer = self.oclAsType(ObjectToken).eobject
      .oclAsType(insurance::Claim).size
```

casts the `self` object to an `ObjectToken`, accesses the carried object through the `eobject` reference, casts this object to a `Claim`, and finally accesses the `size` attribute of that claim. It stores the attribute's value into a local variable `size` which can then be used within the computation of the actual performance rates.[3]

9.2.3.5 Evaluating Performance Definitions

In Sect. 9.2.2.1, we have seen the general algorithm for generating a PEPA model from a GROOVE transition system. That algorithm contained a method `computePerformanceRate(GraphTransition)` which we will define in this section.

Before we turn to the pseudo code formalizing the evaluation, let us first get an idea of what the algorithm does. First, if the given `Performance` is of type `SimplePerformance`, the evaluation result is the rate associated with that `SimplePerformance`. Otherwise, the evaluation is (much) more complex: The graph transition is used to compute the actual objects to which the DMM rule nodes have been mapped when the rule had been matched to the state graph – from these objects, the referenced attribute values are received and bound to the parameters of the `ParameterizedPerformance`'s OCL expression. Finally, the OCL expression is evaluated, and the resulting value is returned.

Let us now investigate the algorithm implementing the described behavior, which is depicted as Listing 9.2. In line 2, the `Performance` for the given rule label is received; we do not show the details of this here. Then, in lines 3 to 6, the simple cases are handled: If there is no performance definition for the current

[3]Recall from Sect. 9.2.2 that a rate describes how often an event occurs in a fixed period of time; we therefore divide through the `size` value, resulting in a longer execution time for a bigger `size`.

transition, we return the PerformanceModels defaultPerformance, and if the performance definition is of type SimplePerformance, we return the associated rate.

Listing 9.2: Algorithm for evaluating DMM performance definitions

```
double computePerformanceRate(GraphTransition transition)                  1
  Performance performance := getPerformance(transition.label)              2
  if (performance = null)                                                  3
    return getDefaultPerformance()                                         4
  if (performance is SimplePerformance)                                    5
    return performance.performance                                         6
  Map map := getDmmNodesToObjectsMap(performance.rule, transition)         7
  String ocl := bindParameterValuesToOclExpressionVariables(performance    8
      , transition, map)
  EObject self := null;                                                    9
  if (performance is ParameterizedPerformanceWithContext)                  10
    self := map.get(performance.context)                                   11
  return OCL.evaluate(self, ocl)                                           12
                                                                           13
String bindParameterValuesToOclExpressionVariables(                        14
    ParameterizedPerformance performance, GraphTransition transition,
    Map map)
  List arguments := new List()                                             15
  for each (Parameter parameter in performance.parameters)                 16
    EObject realObject := map.get(parameter.node)                          17
    arguments.add(realObject.eGet(parameter.attribute))                    18
  return java.util.String.format(performance.ocl, arguments);              19
                                                                           20
Map getDmmNodesToObjectsMap( Rule rule, GraphTransition transition)        21
  Map result := new Map()                                                  22
  for each (Node dmmRuleNode in rule.nodes)                                23
    for each (GrooveNode grooveRuleNode in transition.ruleGraph.nodes)     24
      if (grooveRuleNode represents dmmRuleNode)                           25
        GrooveNode grooveStateNode := transition.getMatchedStateNode(      26
            grooveRuleNode);
        EObject realObject := Emf2Groove.getEObject(grooveStateNode)       27
        map.put(dmmRuleNode, realObject)                                   28
  return result                                                            29
```

Otherwise, the above behavior is realized by first computing a map from DMM rule nodes to the objects they have been mapped to in line 7. The implementing method (lines 21–29) works as follows: It traverses over the DMM rule's nodes in line 23. It then searches for the corresponding GROOVE node of the transition's rule graph (i.e., the node to which the DMM node had been mapped during transformation into the GROOVE grammar as described in Sect. 6.3). Then, it uses the transition's matching to identify the GROOVE state node to which the GROOVE rule node had been mapped (line 26), and finally, it receives the corresponding EObject from the EMF2Groove tool (line 27).

With this map, we can now bind the parameters of the Parameterized-Performance's OCL expression to the actual values of the state's objects in line 8. The implementation of this (lines 14-19) works as follows: for each Parameter of the ParameterizedPerformance, the EObject is received from the map computed above (line 17). Then, Ecore's reflection mechanisms are used to receive the object's attribute value (line 18). Finally, the OCL string is formatted, i.e., the occurences of parameter references within the string are

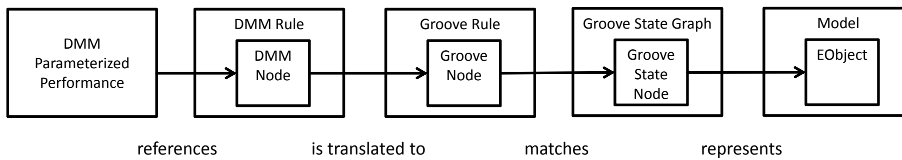

Figure 9.24: Connection between DMM performance model and state model.

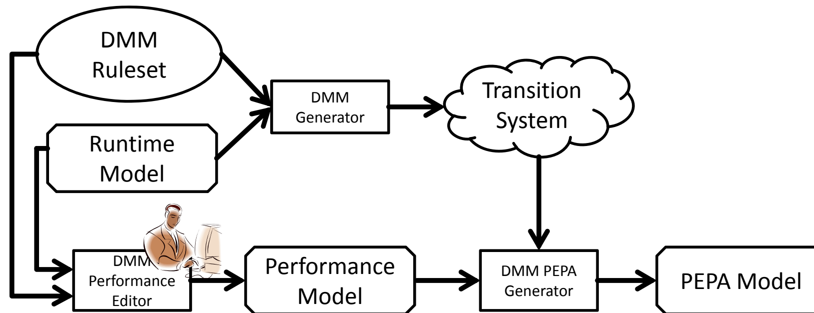

Figure 9.25: Workflow generating a PEPA model from a DMM specification, a model, and a performance model.

replaced with the computed attribute values in line 19.

Figure 9.24 visualizes the connection between the DMM rule nodes referenced by a DMM performance definition and a model's contained EObjects.

In the case of a `ParameterizedPerformance`, we are now ready to evaluate the OCL expression and return the result. In the case of a `ParameterizedPerformanceWithContext`, one more step is necessary: In lines 10 and 11, the `EObject` which corresponds to the `ParameterizedPerformanceWithContext`'s context node is received from the map and then used as the OCL `self` object.

9.2.4 DMM goes PEPA

Now that we have seen the different kinds of performance definitions DMM provides, we present the workflow of generating PEPA models, which is shown in Fig. 9.25. At the top, the standard DMM workflow is shown: A DMM specification and a model are fed into the DMM generator which computes the labeled transition system.

At the bottom, the PEPA-specific part of the workflow can be seen: Using the DMM specification and the model, the modeler creates a `PerformanceModel`, using `SimplePerformance`, `ParameterizedPerformance`, and `ParameterizedPerformanceWithContext` instances as desired. Then the PEPA model is generated from the labeled transition system and the `PerformanceModel` as described in Sect. 9.2.2.1, computing the performance rates from the `PerformanceModel` as described in the last section. The PEPA model can then be analyzed using the PEPA tooling.

Task	Avg execution time	Performance rate
Get Insurance Info	2 mins	720
Check Validity	5 mins	288
Check Probability	15 mins	96
Calculate Amount	10 mins	144
Verify Calculation	10 mins	144
Send Money	3 mins	480
Send Reject Letter	4 mins	360
Archive Claim	2 mins	720
Schedule for Expert	1 min	1440
Schedule for later	1 min	1440
Update Statistics	5 mins	288

Table 9.2: Average execution times and resulting performance rates of the tasks of the business process depicted as Fig. 9.16

9.2.5 Improving the Example Process

It remains to show how the concepts explained in the last sections can be used to improve our example model of Sect. 9.2.1. This is now rather straight-forward:

1. The modeler creates a DMM performance model reflecting the execution times of Table 9.1 on page 198. This can be done using a `ParameterizedPerformance` which distinguishes different names of the actions as we have seen in Sect. 9.2.3.3, and returns the according performance rate. If desired, attributes of the `Claim` object flowing through the activitiy can additionally be taken into account as seen in Sect. 9.2.3.4.

2. The modeler creates variants of the performance model of step 1, reflecting the possible changes to our business process as suggested on page 198.

3. Using the given activity "Process Claim" and the DMM semantics specification for UML activities, the modeler generates the labeled transition system describing the activity's behavior.

4. For each performance model of step 2, the modeler generates an according PEPA model, making use of the transition system computed in step 3.

5. Finally, the modeler uses the PEPA tooling to analyze each of the PEPA models and identifies the change which results in the best average execution time.

In the case of our example business process, the result of step 1 will be a performance model with a single `ParameterizedPerformance` – we have seen such a performance model as Fig. 9.22. The rates assigned to the different actions are computed as follows: As 24 hours consist of 1440 minutes, we divide 1440 through the average execution time of our tasks. For instance, task "Get Insurance Info" takes 2 minutes on average; thus, we define the performance rate of that task as $1440/2 = 720$, the meaning being that within 24 hours, that task will on average be executed 720 times. Table 9.2 shows the performance rates for all tasks of our activity.

Variant	Throughput	Avg execution time
Process as is	101.3	14.2 minutes
“Check Probability”	111.6	12.9 minutes
“Calculate Amount”	103.7	13.9 minutes
“Verify Calculation”	102.1	14.1 minutes

Table 9.3: Average execution times of the business process depicted as Fig. 9.16 after improvement

Then, variants of the performance model are created in step 2, each modeling the desired change as suggested on page 198. In particular, the first variant will have a performance rate of 144 (10 minutes) for task “Check Probability” (and the other rates stay unchanged), the second variant will have a rate of 240 (6 minutes) for task “Calculate Amount”, and the final variant will have the same rate of 240, but this time for task “Verify Calculation”.

In steps 3 and 4, the modeler uses the DMM tooling to generate the activity's labeled transition system and, using that transition system, one PEPA model for each performance model variant.

Finally, the modeler uses the PEPA tooling to analyze the resulting PEPA models. Here, the property of interest is the transition throughput of the activityFinalNode.consume()# rule: The higher the throughput of that transition, the more often our activity has been completed within the fixed time period of 24 hours we were assuming. Figure 9.26 shows a screenshot of the according view of the PEPA tooling: For each transition, the average throughput has been computed.

The resulting throughputs of the activityFinalNode.consume()# transition are depicted as Table 9.3. It turns out that the most improvement is achieved by speeding up task “Check Probability”. Note that this is consistent with our intuition: Due to the decisions contained in the process, task “Check Probability” is part of more possible executions of the process than the other tasks; as such, improving that task's execution time has a higher effect on the process's average execution time than the other tasks we considered to improve.

9.3 Related Work

In this section we review related work to (generalized) (E)PPSL and model checking in the next section.

9.3.1 Functional Requirements

The most important related work to ours can be grouped into three categories: Visual languages for expressing temporal logic properties, model checking of graph transformatiom systems, and model checking of UML models.

9.3.1.1 Visual temporal logic

Del Bimbo et al. [40] provide a visual language for branching time temporal logic which is more expressive than the PPSL since it allows to express all possible CTL formulas visually. However, the approach lacks the understandability of

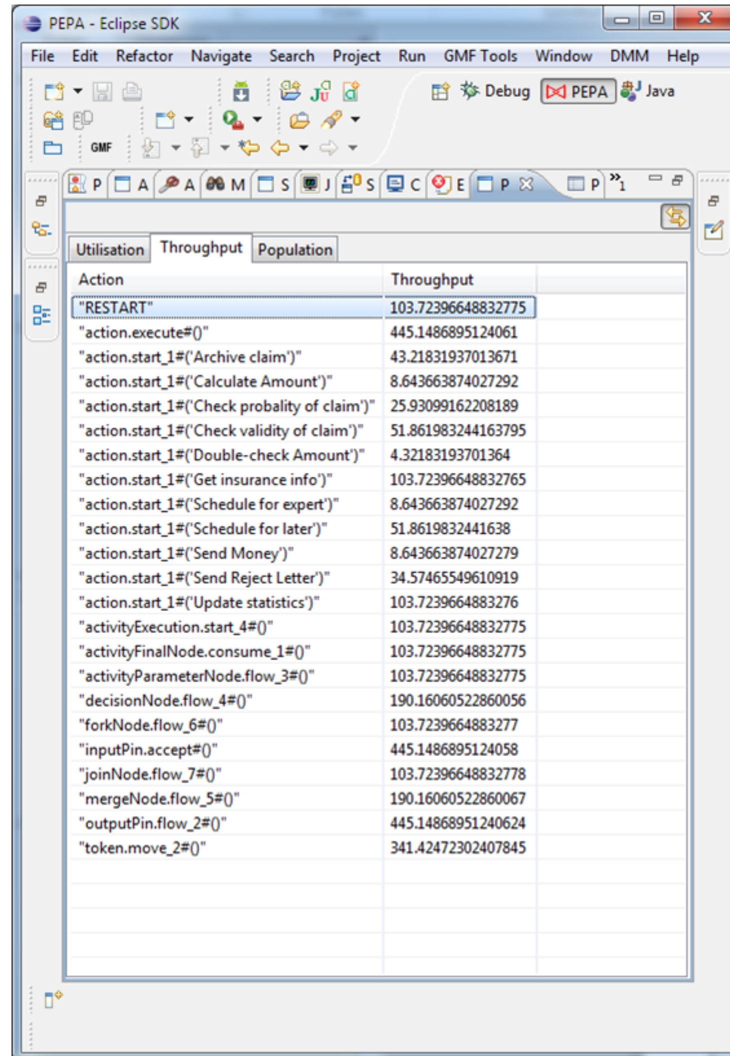

Figure 9.26: Screenshot of the PEPA tooling, transition throughput view.

(E)PPSL. It could still be a candidate for a more expressive visual language for functional requirements specification in case it turns out in practice that (E)PPSL is not expressive enough for some tasks.

Another visual language for specifying functional requirements is the *Graphical Interval Logic* (GIL) by Kutty et al. [124]. It supports a subset of LTL and is thus comparable to PPSL.

Janssen et.al. [109] suggest an approach for the verification of business processes using model checking techniques which is based on a proprietary process modeling language. The authors formalize different basic constraints; the approach does not allow for the verification of custom, user-defined properties.

9.3.1.2 Model Checking Graph Transformation Systems

One of the most important distinctions of the several approaches is whether they apply model checking directly on the graphs and rule applications, or whether the graph transformation system is translated into another language for which model checking tools exist. In [171], Rensink and Varro compare two different such approaches for model checking systems whose behavior is specified by graph transformation systems. One approach is to build up the state space by directly simulating the graph transformation rules, where the other approach encodes graphs into fixed vectors and rules into commands modifying these state vectors. The authors conclude that the latter outperforms the former if the dynamic nature of the described systems is limited, while the first approach shows its strength for highly dynamic systems.

GROOVE is one of the tools which follow the former approach. In his PhD thesis [113], Kastenberg has developed model checking techniques for the GROOVE tool set, including a new technique of partial order reduction dedicated to reducing the state space without loss of possible behavior of the system. One of the results of his thesis is the possibility to verify GROOVE transition systems using LTL. Additionally, in [114] Kastenberg and Rensink show how to model check GROOVE transition systems by means of CTL formulas. In our work, we have used their techniques extensively (see Sect. 4).

Constituents of the second approach of translating graph grammars into intermediate formats ready for model checking can be found more often. For instance, Schmidt et al. [182] translate instances of arbitrary modeling languages into an intermediate representation of transition systems defined by a corresponding metamodel, from which they automatically generate input models for the SPIN model checker. Baresi [15] translates AGG specifications [200] into BOGOR models [175], enabling the verification of LTL properties. De Lara et al. [126] use the AToM3 tool to analyze and simulate systems which have been described using different formalisms. For this, the different formalisms are translated into a common one by means of graph transformations. The latter formalization can then be model checked.

Eckardt et al. [43] present a rather interesting approach for dealing with the state explosion problem: They formally define *coordination patterns* by means of graph transformations, which allows to first prove that an architecture includes only component connections which correspond to the defined coordination patterns, and then to model check safety and liveness properties only for each individual coordination pattern rather than for the system as a whole. For model checking, they use the model checker UPPAAL [21].

9.3.1.3 Model Checking UML Models

There are many approaches for model checking UML models. Many of these deal with UML state machines, the reason probably being that the semantics of those is relatively clear (at least compared with, say, UML activities), and that state machines are quite often used for modeling real-world systems. For instance, Jussila et al. [111] translate hierarchical UML state machines into input for the model checker SPIN [104] with the goal of checking properties of protocol specifications. Dong et al. [41] translate UML state machines into Büchi automata and verify behavioral properties using LTL. Schäfer et al. [181] translate state machines into input for SPIN and UML collaborations into Büchi automata; then, they use the model checker SPIN to verify whether the state machines can handle the interactions described by the collaborations.

In the case of UML activities, less research has been performed. In [67], Eshuis translates Activity Diagrams from UML 1.5 into the input language of the model checker NuSVM [27], giving activities a statechart-like semantics as stated by the UML 1.5 specification, and uses that semantics for verification of certain properties. This work is extended in [68], where Eshuis and Wieringa provide guidelines on how to support the analysis of control-flow properties in the case of UML activities.

Another class of behavioral UML diagrams are UML interactions. Knapp et al. [119] translate UML interactions into *interaction automata* and use the tool HUGO/RT [131] for model checking.

There are also approaches which are not restricted to a single UML sub language, but treat models consisting of several languages. Gagnon et al. [77] translate class, state and communication diagrams into MAUDE specifications [133] and use SPIN to model check UML models consisting of a combination of the mentioned diagrams. Their focus lies on concurrent systems. In the same line, Niewiadomski et al. [152] formalize all executions of a UML system. In contrast to Gagnon et al., their target formalism are boolean propositional formulas, the satisfiability of which is then checked using a SAT-solver.

xUML [137] is an executable subset of the UML. In [213], Xie et al. translate xUML models into S/R, the input language for the COSPAN model checker [93], making use of xUML's execution semantics.

All of the above approaches differ to our work in that they are targeted at the UML; this is not the case for DMM, which can be used to provide semantics for all kinds of metamodel-based behavioral languages, and to analyze the behavior of according models. The model checking concepts presented in Sect. 9.1 can be applied to all those languages.

However, in terms of scalability, the above approaches tend to be more efficient than the currently available DMM tooling. This is mainly due to the complexity of rule matching, which is NP-complete in the size of the rule graph (which is both bad and good news; rule graphs tend to have a rather small size). Additionally, the state graphs have to be compared to each other, which is again NP-complete; however, GROOVE successfully applies heuristics (graph certificates) to significantly reduce the number of actual graph comparisons. See [167] for more information.

9.3.2 Non-Functional Requirements

Analyzing models for performance properties first of all requires the possibility to add according performance information to the models, e.g. by means of durations of executions, probabilities etc. To our knowledge, there is no standard way to add such information to Ecore/MOF models; however, in the case of the UML, two dedicated profiles exist: the *UML Profile for Schedulability, Performance, and Time Specification* (SPTS) [153] and the *UML Profile for Modeling and Analysis of Real-Time Embedded Systems* (MARTE) [159]. SPTS provides concepts for analyzing schedulability and performance as well as time and time-related mechanisms. MARTE basically is a follow-up to SPTS, providing more expressiveness and flexibility and being targeted at the UML 2.0 (which was a major re-design to previous UML versions). Woodside [212] discusses usage scenarios of the profiles. In our work, we use the custom metamodel depicted as Fig. 9.20 on page 202 for performance specification, which is tightly coupled to DMM since it references DMM's metamodel, but still provides means to model performance information strictly separated from the semantics information itself.

As the de facto modeling standard, many approaches of analysing the performance of UML models have been suggested. Cortellessa et al. [32] extract performance relevant information from several UML diagrams (i.e., use case, sequence, and deployment diagrams). That information is then used to generate a queuing network which can be analyzed with standard tools. Bouarioua et al. [24] use graph transformations to translate UML sequence diagrams into stochastic petri nets, both being defined by custom metamodel. The resulting petrinet can then be analyzed again by means of standard tools. King and Pooley [117] follow a similar approach for collaboration and statechart diagrams. Lindemann et al. [130] translate UML statecharts and activities into generalized semi-Markov processes; their tool DSPNexpress is then capable of doing performance analysis. D'Ambrogio and Bocciarelli [37] describe web service orchestrations by means of annotated UML activities; the results of the performance analysis are provided through the service registry by means of Performance-enabled WDSL [36].

Other approaches provide dedicated languages for performance modeling. For instance, the *Palladio Component Model* (PCM) [20] allows to model software components and their performance characteristics in a—compared to the UML—much more detailed way. PCM models can then be analyzed using the Eclipse-based Palladio workbench. Another example in the context of component-driven software engineering is KLAPER [87], which defines a simple but expessive kernel language for performance modeling, reducing a language engineer's task to defining a model transformation into that language (instead of directly mapping her language into the target formalism suited for analysis).

All of the above approaches translate models into formalisms suitable for analysis. However, every such translation assumes an individual behavioral semantics of the according languages. In contrast, DMM is targeted at defining the behavioral semantics of the UML and other languages in the first place, which can—as we have seen during this thesis—be used for all kinds of purposes, including simulation, functional analysis, or even language reference for advanced language users. As such, in the ideal situation, DMM performance information is added to a language the semantics of which is already accepted

and used in different contexts, and the performance analysis is based on exactly that semantics.

10

Debugging Models

In the last chapter, we have seen how we—given a model and an according DMM semantics specification—can formulate and verify behavioral properties of that model. For this, we have used GROOVE to generate a labeled transition system which we have then model checked using the formulas generated from the generalized (E)PPSL expressions. Of course, the GROOVE model checker delivers counter examples in terms of states and transitions of the transition system.

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

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

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

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

In the next section, we will show what information the language engineer has to provide, and how this information is specified by means of certain *configuration models*. Section 10.2 then introduces to the debugging of models in a visual way; we show how to define breakpoints and integrate the animation capabilities developed in Sect. 10.1 into the model checking process. The implementation of the above concepts is briefly discussed in Sect. 10.3, and finally,

related work is reviewed in Sect. 10.4. This chapter is based on [12], which is a result of the diploma thesis of Nils Bandener [11].

10.1 Visual Model Execution

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

Thus, we have developed a tool for visually executing and debugging models with DMM-specified semantics, the *DMM Player* [12, 11]. From a technical perspective, the tool is a set of Eclipse plug-ins, which is able to process models and DMM semantics specification. The visualization is realised using the *Graphical Modeling Framework* (GMF) [50], a framework for developing visual editors for EMF-based models. As there are already numerous existing GMF editors for behavioral models which—however—do not support displaying runtime-information such as tokens or active states, the DMM Player also provides means to augment existing editors by such elements.

Let us now focus on the underlying concepts which make the visualization of a model execution possible. Using UML activities as a running example, we start with the fundamental question on how to visualize execution states and how the augmentation of existing visualizations can be specified in a model-driven way. Beneath the graphical dimensions, we will also have to consider the time axis when visualizing the execution. This will be covered in Sect. 10.1.2. Section 10.1.3 covers means of controlling the execution path when external choices are necessary. Finally, in Sect. 10.1.4 we will demonstrate our approach on another language, i.e., UML Statemachines.

10.1.1 Visualizing Runtime Information

In order to visualize the behavior of a model, i.e., the development of its runtime state over time, it is essential to be able to visualize the model's runtime state at all. However, this cannot be taken for granted. While certain visual languages have an inherent visual syntax for runtime information—such as Petri nets [162] visualizing the state using tokens on places—many visual languages only support the definition of the static structure. For instance, as we have seen earlier, the behavioral semantics of UML activities is based on token and offer flow, but the abstract syntax definition, i.e., the UML metamodel, does not contain those concepts. Therefore, existing visual editors for UML activities only support visualization of the static structure of activities.[1]

As we have seen, DMM copes with this by encoding such information into the runtime metamodel, which formally defines an abstract syntax for runtime states of models in the particular language. With the DMM Player, we have

[1] We do not explain the definition of such diagram visualizations within this thesis; however, in Fig. 9.16 on page 197 we have seen an example for the visualization of a UML activity's static structure by means of an editor provided by the UML2Tools project [52] and realized on top of the *Graphical Modeling Framework* [50].

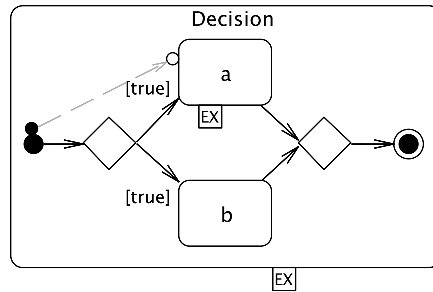

Figure 10.1: An UML activity diagram with additional runtime elements.

developed a set of concepts and techniques to define a concrete syntax for those runtime states. Similar to the enhancement by runtime information in the abstract syntax, the concept allows for building on the concrete syntax of the static part of models in order to create the concrete syntax for runtime states. The enhanced concrete syntax is defined in a completely declarative, model-based way.

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

Our implementation of the DMM Player allows to extend GMF-based editors, as GMF offers all required extension mechanisms. The particular implementation is described in Sect. 10.3. In the following, we will focus on the concepts, which are—while being partially inspired by—independent from GMF. Essentially, this means that definitions for an enhanced concrete syntax may be also used in conjunction with other frameworks. This of course requires an implementation interpreting the DMM artifacts for these particular frameworks.

The first concern is how the runtime information should be visualized in concrete syntax. Beneath the obvious question on the shape or appearance of runtime information, it may also be necessary to ask what runtime objects should be included in the visualization at all. Certain runtime information may be only useful in certain contexts. In the example of UML activity diagrams, the visualization of tokens is certainly essential; we visualize tokens—aligned with the visualization in Petri nets—as filled black circles attached to activity nodes. An example for such a diagram can be seen in Fig. 10.1. For debugging of models and semantics, visualized offers may also be useful; offers are visualized as hollow circles. As multiple tokens and offers may be in action at once, an arrow visualizes which offers are owned by which tokens.

The diagram in Fig. 10.1 also shows boxes labeled with the letters EX. These boxes indicate that the particular node is currently executing.

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

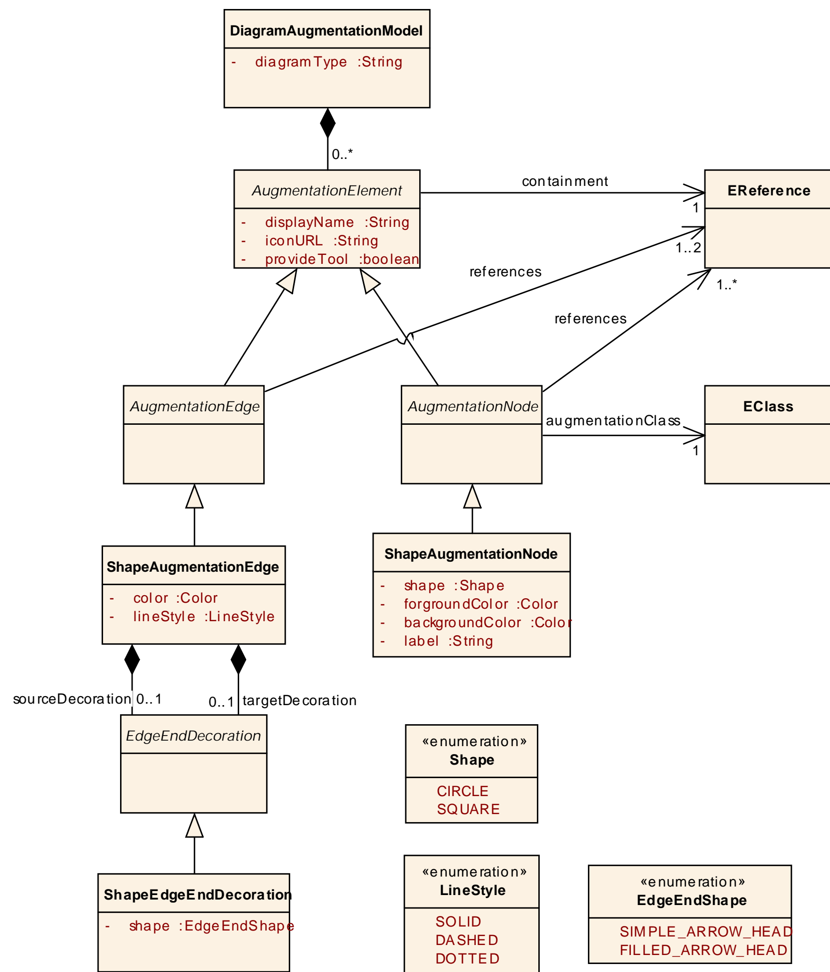

Figure 10.2: Metamodel for diagram augmentation models.

Diagram augmentation models are defined by means of the metamodel which is pictured in Fig. 10.2. The class `DiagramAugmentationModel` is the root element, i.e., each augmentation model contains exactly one instance of this class. The attribute `diagramType` is used to associate a particular diagram editor with the augmentation model – this diagram editor will then be used for displaying runtime states.

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

The way the elements are integrated into the existing diagram is determined by the references between `AugmentationNode` and `AugmentationEdge` on the one side and `EReference` and `EClass` on the other side. They represent elements from the DMM runtime metamodel the visualization is supposed to be based on.

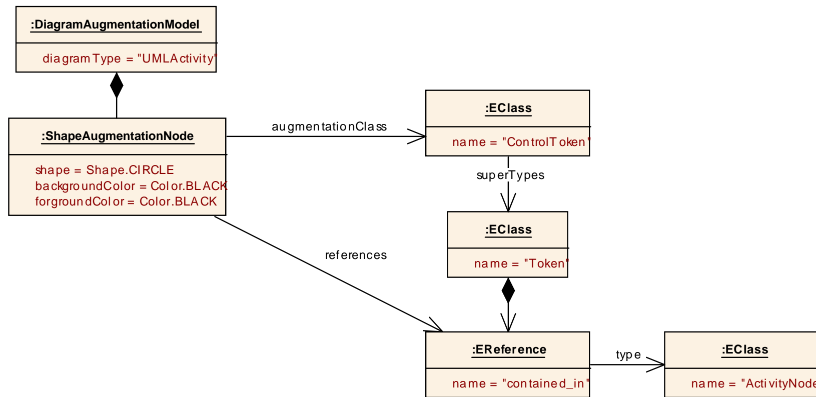

Figure 10.3: Excerpt of the augmentation model for UML activities.

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

Figure 10.3 shows the part of the augmentation model for UML activities which specifies the appearance of control tokens, which are a subclass of tokens. The references link specifies that the reference named contained_in determines to which ActivityNode objects the new nodes should be attached to. The references link is part of the class Token. However, as the link augmentationClass specifies the class ControlToken as the class to be visualized, this particular ShapeAugmentationNode will not come into effect when other types of tokens occur in a runtime model. Thus, other augmentation nodes may be specified for other tokens.

10.1.2 Defining the Steps of Executions

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

However, the sequence of states produced by DMM is not necessarily well suited for a visualization. In many cases, subsequent states only differ in parts that are not visualized. These parts are primarily responsible for internal infor-

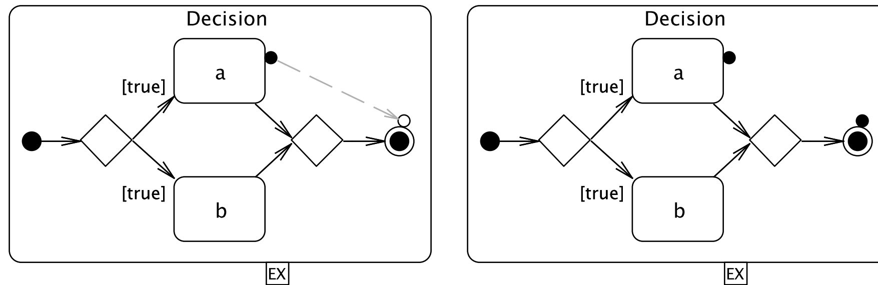

Figure 10.4: A sequence of states exhibiting a temporary inconsistency.

mation which is specific to the particular implementation of the semantics, but is not relevant for the behavior of the final model. Including these steps in the animation would cause strange, irregular pauses between visual steps.

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

Figure 10.4 shows an example for a temporary inconsistency. In the first state, an offer has reached the final node. The activity diagram semantics now demands that the corresponding token should be moved to the node reached by the offer. The DMM implementation of the semantics however creates a new token on the target node before removing the original token from its location. This creates a temporary inconsistency as introduced above with two tokens being visible at once.

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

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

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

If temporary inconsistencies are to be avoided in the visualization, other measures need to be applied. A simple and straight-forward approach would be to visualize only the state when the application of a bigstep rule has been

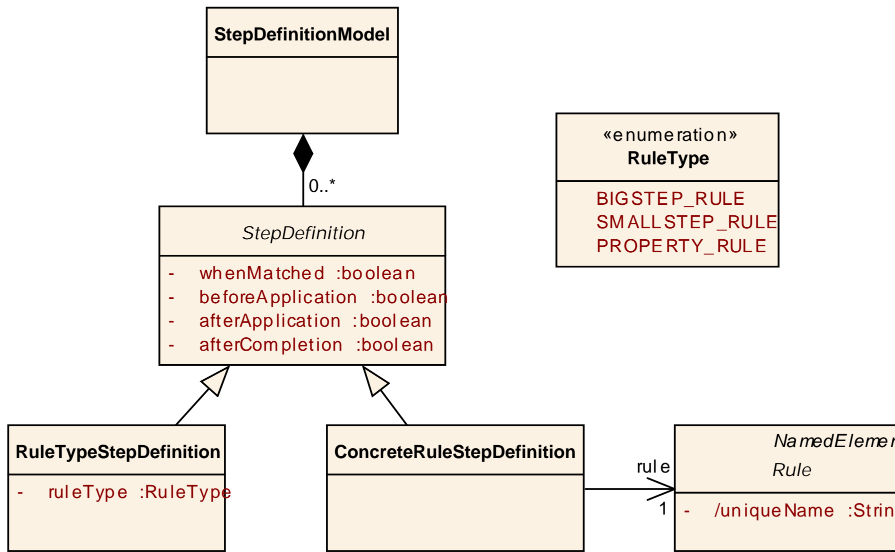

Figure 10.5: Step definition metamodel.

finished; application in this context means that the changes by the bigstep rule and by its invocations have been performed. As temporary inconsistencies are typically raised by an invocation and again fixed by a consecutive invocation, temporary inconsistencies will be fixed when all invocations have been finished and thus the application of a bigstep rule has been finished.

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

An obvious solution would be the explicit specification of all rules that should trigger a visual step. This is, however, also the most laborious solution, as each rule of the semantics specification needs to be treated. In the case of the DMM-based UML activity semantics specification by Hornkamp [105], this means the inspection of 217 rules.

Our solution is a combination of the approaches – the metamodel defining step definition models is depicted as Fig. 10.5. Thus, in addition to the specification of individual rules triggering visual steps, the DMM Player also allows for the specification of e.g. all bigstep rules. Furthermore, it is possible to specify whether the visualization should be triggered before or after the application of the particular rules, already when they just match, or after completion of the rule (which refers to the point in time where all the rule's invocations have been processed).

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

10.1.3 Controlling Execution Paths

A limitation of the behavior visualization using an animated sequence of states is its linearity. In some cases, the behavior of a model may not be unambiguously defined. For instance, this is the case in the activity diagram we have seen before in Fig. 10.1; the left decision node has two outgoing transitions. Both are always usable as indicated by the guard `[true]`. In a transition system, such a behavior is reflected by a fork of transitions leading from one state to several distinct states. In an animation, it is necessary to choose one path of the fork. At first sight, it is evident that such a choice should be offered to the user.

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

However, there are cases in which it is not feasible for the user to choose the path to be used for every fork in the transition system. This is primarily the case for forks caused by concurrency in the executed model. Even though a linear execution does not directly suffer from state space explosion, concurrency might require a decision to be made before most steps of a model execution. This is due to the interleaving semantics GROOVE realizes. For instance, consider the transition system depicted as Fig. 4.3 on page 28: The interleaving structure caused by the underlying model's concurrency results in basically every state having more than one outgoing transition.

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

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

As an example, consider the UML activity depicted as Fig. 10.6. Its transition system contains a lot of forks, only a few of which are actually caused by `DecisionNodes`; the other forks are caused by the interleaving semantics resulting from the model's concurrency. Since concurrency means that the order of the concurrently executed `Actions` does not matter, the user should not be bothered with those forks. In the following, we will discuss how the desired behavior can be achieved.

First of all, a basic measure for identifying the model construct that caused a fork or a part of it is considering the transformation rules that are used for the transitions forming the fork. In the case of the DMM semantics for UML activities, the transitions which cause the forks at decision nodes are produced by the bigstep rule decisionNode.flow()# (see Fig. 10.7). Forks which consist of transitions caused by other rules can be regarded as forks caused by concurrency.

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

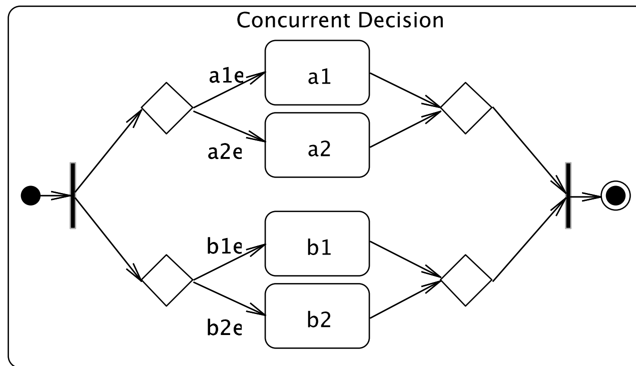

Figure 10.6: Example UML activity that contains concurrency and decisions (from [11, p. 59]).

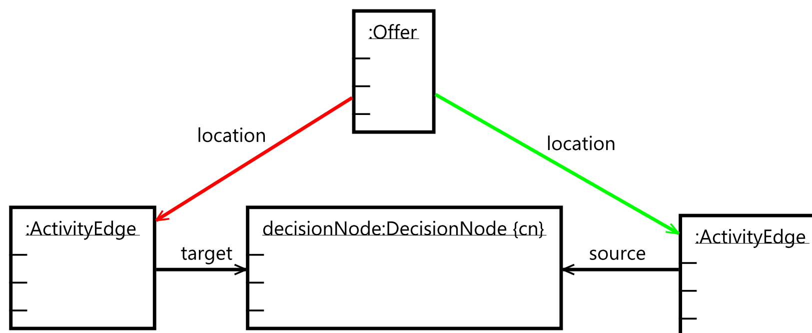

Figure 10.7: DMM Rule decisionNode.flow()#.

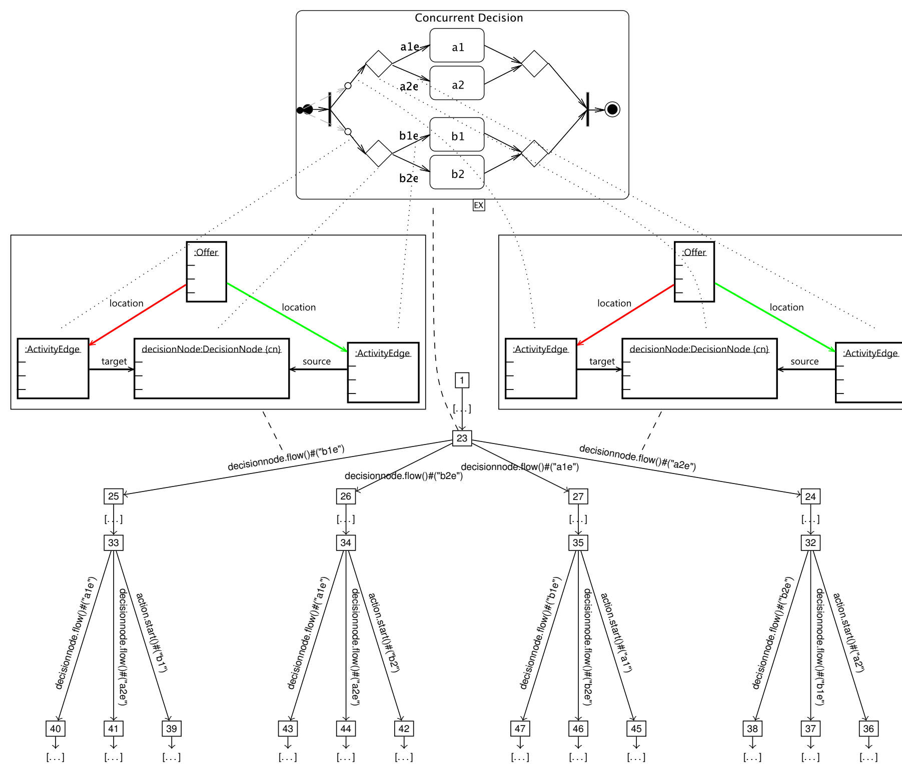

Figure 10.8: A part of the transition system induced by the model from Fig. 10.6 (after [11, p. 59]).

the user but just choices purely caused by concurrency.

Thus, it is necessary to group the transitions at a fork by the model element they are related to. This model element can be identified by using the rule match associated to the particular transition (see Sect. 6.3 for the definition of of rule matching and application). The match is a morphism between the nodes of the particular DMM rule and elements from the model. Generally, one node from a rule that is supposed to trigger a user choice can be used to identify the related model element. In the case of the DMM activity semantics, this is the `DecisionNode` element.

Figure 10.8 visualizes the above concepts. At the top, a runtime state of our activity from Fig. 10.6 can be seen: The single token is sitting on the `InitialNode`, and its `Offer` has just passed the `ForkNode` and has been copied. Thus, in the next step, either the upper or the lower `Offer` is going to pass the according `DecisionNode`. For each of the `DecisionNodes`, two outgoing `ControlFlows` exist. Thus, there are four possible continuations.

At the bottom, a part of the activity's transition system can be seen. The runtime state of our model is state 23, which accordingly has four outgoing edges, each labeled with rule decisionNode.flow()#. Note that the rule has slightly been modified for this example: An emphasized node attribute has been added which displays the name of the `DecisionNode's` outgoing `ControlFlow` within the transition's label.

In the middle, the rule has been depicted two times, and for each rule, an underlying rule matching (i.e., the morphism from rule node to state node, the latter in concrete syntax) has been visualized. The right matching refers to the rule application between states 23 and 25: The `DecisionNode` node of the rule has been mapped to the model's lower `DecisionNode`, and the `DecisionNode's` outgoing `ActivityEdge` within the rule has been mapped to the `ControlFlow` with name “b1e”. The other match refers to the rule application between states 23 and 24; the rule's elements have been mapped to the upper `DecisionNode` and `ControlFlow` “a2e”.

Now, to distinguish the matchings by the model's `DecisionNodes`, it suffices to group them by the match between the rule's `DecisionNode` and the model's `DecisionNode`: Two rule matchings are in the same group if their group node is matched to the same runtime model node. For state 23, this results in two groups of rule applications.

With these components, we can build an algorithm for identifying the instances of transition system forks in which the user should be asked for a choice; the algorithm is depicted as Fig. 10.9.

It first receives all outgoing transitions of the current state. If no or only one transition is found, the algorithm is done and returns the found transition (or null, respectively). Otherwise, the transitions are grouped by the rule and by the elements bound to the grouping node defined for the particular rule. If there is no grouping node or the rule is not supposed to cause user choices, the particular transition forms a group of size one.

Next, an arbitrary group is selected. If that group contains only one transition, the algorithm is again done and returns that transition. Otherwise, the selection of a transition depends on the kind of transitions contained in the group: If switch instances are defined for them, than the selection is delegated to the user. Otherwise, we have concurrent behavior, in which case the algorithm arbitrarily chooses a transition. In both cases, the selected transition will

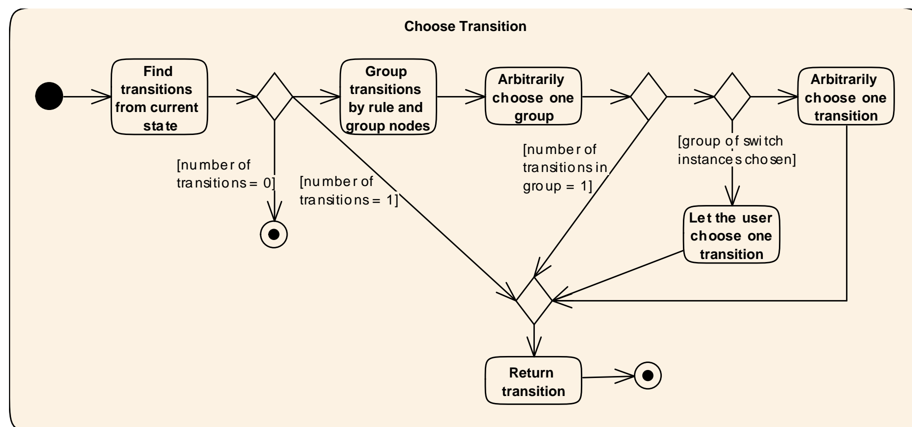

Figure 10.9: Algorithm for selecting a transition to be followed.

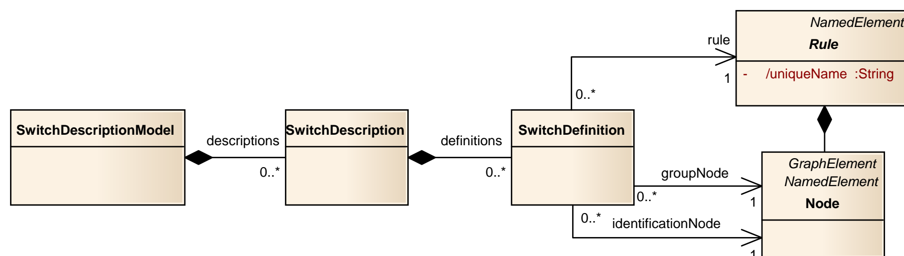

Figure 10.10: Metamodel for describing rules which indicate decisions.

be returned.

This algorithm enables us to ensure that the user is only required to make choices regarding single instances of certain model constructs, such as decision nodes.

A remaining problem is how to give the user an overview over the possible choices. We can again utilize nodes from the rules that are supposed to trigger user choices. In most cases, those rules contain a node which represents the different targets which can be reached while the aforementioned grouping node stays constantly bound to the same model element. If the model element represented by the target node is part of the diagram, this diagram element can be used for identifying the different choices.

In the case of UML activities, this is the target `ActivityEdge` node in the rule decisionNode.flow()#. This can be seen in Fig. 10.8: The left, visualized match differs by the match corresponding to the transition between states 25 and 26 only by the match target of the `DecisionNode's` outgoing `ActivityEdge`.

The information on rules indicating user choices, grouping nodes and identification nodes is captured as instances of the metamodel depicted as Fig. 10.10. A `SwitchDescriptionModel` consists of `SwitchDefinitions` which are grouped to `SwitchDescriptions`. Each `SwitchDefinition` references its `rule` as well as the `groupNode` and the `identificationNode`.

The DMM Player can now use switch description models to visualize the

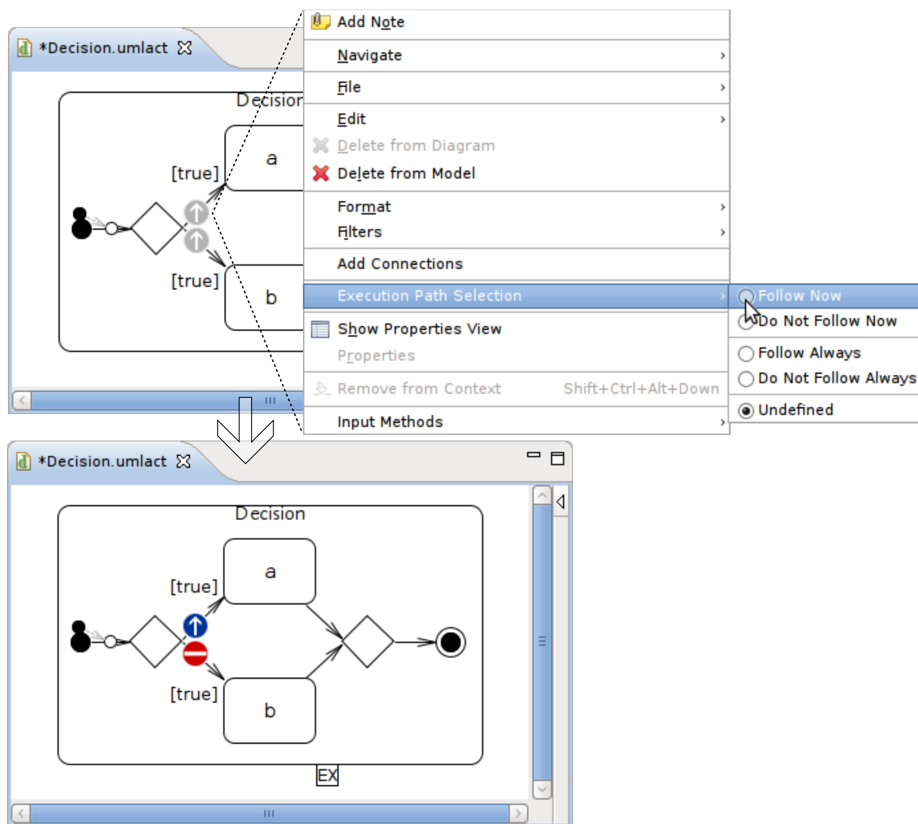

Figure 10.11: UI for choosing execution paths (from [11, p. 50]).

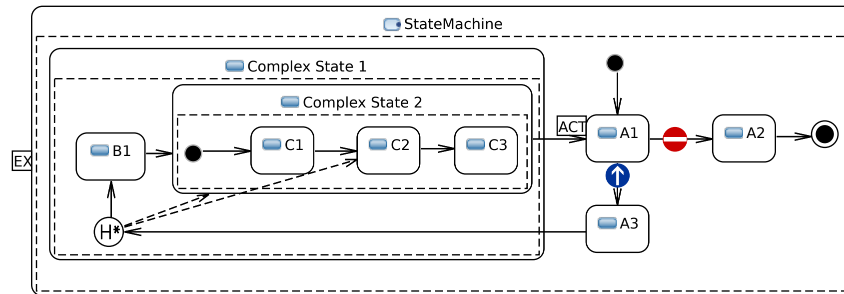

Figure 10.12: Example state of Statemachine execution.

possible choices using generic marker signs as is depicted in Fig. 10.11. The user may easily select one of the choices using the context menu of one of these model elements.

10.1.4 Example: UML Statemachines

To further demonstrate the usefulness of our approach, within this section we provide a second language for which we have applied our visualization/animation approach: UML statemachines.

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

A sample statemachine is depicted as Fig. 10.12 (note that this figure already contains runtime information). The first active state will be state *A1*. From this state, either state *A2* or *A3* will be activated. In case of state *A3*, the *Complex State 1* will be entered. The state marked *H** is a so-called *deep history state*; it makes sure that in case state *Complex State 1* is activated again, all states which were active when that state was left are reactivated.

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

Next, we want to discuss how the execution of a Statemachine is visualized. In Fig. 10.12, we have already seen a Statemachine augmented with runtime

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

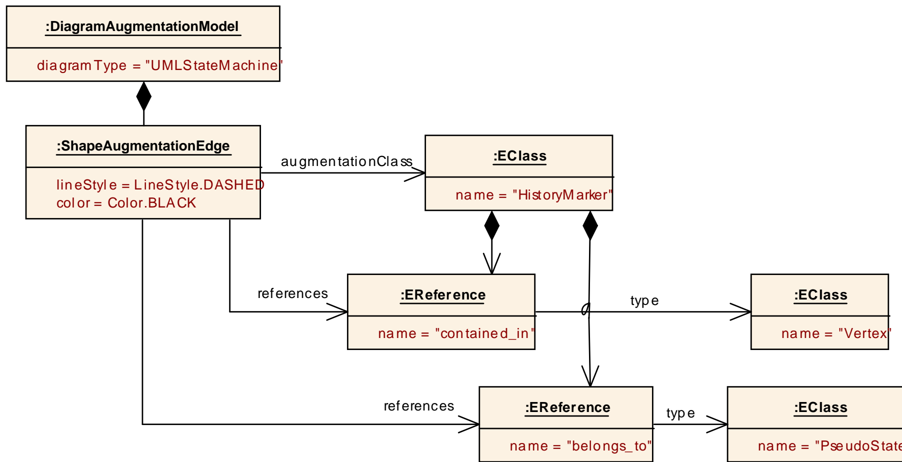

Figure 10.13: Excerpt of the augmentation model for UML Statemachines.

information. Active states can be recognized by an attached box with the label *ACT*. These boxes represent the `Marker` instances from the runtime metamodel.

Further runtime information can be seen around the deep history state *H**. The dashed arrows pointing away from that state signify the states that will be activated as soon as the complex state containing the history state is entered again. Thus, these arrows represent `HistoryMarker` instances. The part of the augmentation model that realizes the arrows representing history markers can be seen in Fig. 10.13. The `ShapeAugmentationEdge` instance specifies the class to be additionally visualized, i.e., the `HistoryMarker` and its references which determine the end points of the visualized edge.

We are now ready to explain the runtime state of the Statemachine which can be seen in Fig. 10.12. The currently active state is *A1*. Since from that state, either state *A2* or *A3* can be reached, the DMM player has already asked for a user decision – as the icons show, the user has decided to follow the transition leading to state *A3*.

To realize the user choice, the implementations of the rule transition.fireTransition()# had to be taken into account which handle the different circumstances under which a transition might fire. Four of these rules are related to actual choices; the other rules are related to explicit or implicit concurrency. Explicit concurrency can be modeled within a UML Statemachine by means of using the `PseudoState` with kind `Fork`, and implicit concurrency can be modeled through states containing parallel `Regions`. Therefore, it sufficed to mark the four rules related to choice as `ExecutionPathSwitches`.

The visualization also reveals that *Complex State 1* had already been active in the past. This is because there do exist `HistoryMarker` edges. The edges point to the states which had been active within state *Complex State 1* before it was left through the transition between *Complex State 2* and *A1* (i.e., *Complex State 2* and, within that state, *C2*). Therefore, after two further execution steps, these states will be set active again.

The DMM specification of Statemachines contains several

10.2 Model Examination

In the last section, we have seen how DMM specifications can be enriched with visualization and execution information and then animated. This is already useful when trying to understand the semantics of a certain model. In this section, we will introduce additional support for understanding a model's semantics and fixing problems revealed with the analysis techniques introduced in Ch. 9.

In some sense, we are aiming at providing concepts and tools similar to *debuggers* in classical programming environments. The main concept of debuggers is the *breakpoint*; in the next section, we will discuss the notion of breakpoints DMM supports. Section 10.2.2 then explains the process of executing a model while respecting breakpoints and watchpoints.

We have seen that analyzing functional requirements will—in the case a requirement does not hold—result in a counter example, i.e., a path through the transition system which violates the according requirement. Using the visualization techniques from Sect. 10.1 and the debugging techniques from Sect. 10.2.2, it is straight-forward to provide visual feedback on counter examples to the user, which is discussed in Sect. 10.2.3.

10.2.1 Controlling Model Execution

As we have seen up to now, the main objective of the DMM Player is the visualization of a model's behavior by means of animated concrete syntax. This is very useful when trying to understand the details of a model's behavior, for instance if the model contains flaws. One feature which would obviously be of great use in such scenarios is the possibility to stop the execution of a model as soon as certain states of execution are reached. In other words: The transformation of concepts known from debuggers for classical programming languages to the DMM world is needed.

The purpose of a classical debugger is to help the programmer in searching for and correcting errors of the program. It does this by allowing the programmer to interrupt a program's execution and investigate the internal execution state of the program, e.g. by means of the current program counter, call stack or values of locally and globally visible variables.

The most important concept of classical debuggers is the *breakpoint*, i.e., a line marked such that the program execution stops when that line is reached. In the next section, we will discuss the realization of breakpoints in DMM. A related concept are *watchpoints*, where the developer defines expressions over variables of the program; as soon as a watchpoint's expression evaluates to true, the program stops execution. DMM also supports watchpoints, which are discussed in Sect. 10.2.1.2.

10.2.1.1 Breakpoints

The main concept of classical debuggers is the breakpoint, which refers to a marker in the source code; as soon as program execution reaches that marker, the execution is suspended. In DMM, the situation is slightly different: Executing a model means to apply the rules of the model's semantics specification one after the other. In other words: The execution of a model relies on the model itself and the rules of the semantics specification.

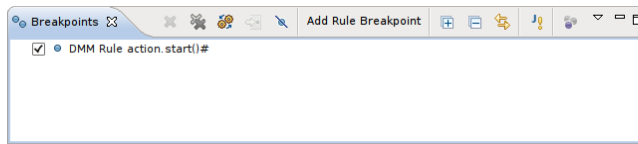

Figure 10.14: Eclipse breakpoint view with single breakpoint for rule action.start()#.

One approach to defining breakpoints in the context of DMM would be to associate them with syntax elements of the executed model; for instance, a certain `Action` could be marked as breakpoint, and the execution could stop as soon as that `Action` is to be executed. However, a number of consecutive rule applications might be needed to realize the semantics of a single syntax element; thus, relying only on a model's syntax elements for defining breakpoints appears to be too coarse-grained.

In DMM, the smallest unit of execution are rules. Thus, a DMM breakpoint is associated to a rule (in contrast to a marker in the source code as in traditional debugging); as soon as this rule takes places when debugging a model, the execution is stopped. DMM breakpoints can be configured to stop execution either if a rule matches the current state, before being applied,[3] or after being applied.

Some program debuggers allow to equip breakpoints with additional conditions. A *conditional breakpoint* will only cause the program's execution to be suspended if the breakpoint is reached *and* the condition evaluates to true at that time. This is also supported by DMM: A breakpoint can be associated with a condition about the rule's nodes' attribute values. This can also be used to realize the scenario described above: A breakpoint can be defined which carries an `Action`'s name as its condition; associated with the main rule for executing `Actions` (i.e., rule action.start()# as seen earlier in this thesis), the breakpoint causes the interruption of the model execution as soon as the according `Action` is to be executed.

However, when using conditions it is not sufficient to use an existing DMM rule for reference of the breakpoint: Adding a condition to the rule would change the rule's—and thus the DMM specification's—semantics. For this reason, DMM breakpoints with conditions are treated slightly differently: If a model is debugged, copies of the rules occuring in conditional breakpoints are created, and the referenced node of each of these rules is equipped with the according condition (the target node of a condition is selected by prefixing the condition's left side's attribute name with the node's name).

The resulting transition system will then only differ from the orginal one by additional transitions: If the original rule is applied *and* the condition is true, an edge "parallel" to the edge resulting from the original rule's application will exist (since the rule copy will result in the same target state as the original rule). These additional edges can then be utilized by the debugger for according model suspensions, but do not change the model's semantics.

Figure 10.14 shows the Eclipse Breakpoint view containing a single DMM breakpoint, in this case for rule action.start()#; to the top of the figure, the "Add

[3] Many rules might match a state, but only one will be chosen for application.

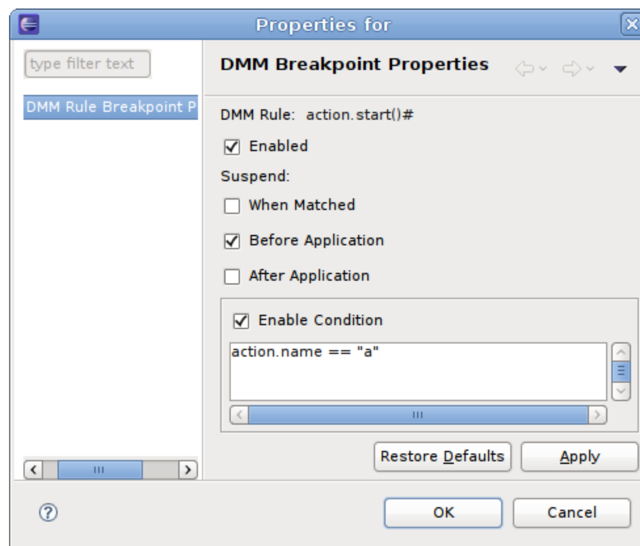

Figure 10.15: DMM breakpoint configuration dialog.

Rule Breakpoint" button can be seen, with which new DMM breakpoints can be defined.

Figure 10.15 shows the dialog allowing to configure the breakpoint shown as Fig. 10.14. At the top, the target rule can be seen, and the breakpoint can in general be enabled or disabled. Below, the actual moment of interrupting model execution can be configured by selecting "When Matched", "Before Application", and "After Application". Note that more than one option may be selected, possibly resulting in more than one suspension per rule application. At the bottom, a condition can be specified, resulting in the behavior as described above. In Fig. 10.15, the condition makes sure that model execution is suspended as soon as the action with name "A" is to be applied.

10.2.1.2 Watchpoints

In contrast to breakpoints, watchpoints are not associated to a certain line of code. Instead, a watchpoint defines a condition over variables of the program; the program's execution is suspended as soon as the condition evaluates to true.

In DMM, property rules can be used for the same purpose. Recall from Sect. 6.2.2.8 on page 68 that property rules do not change the state they are applied to, they only result in a self-transition of that state – that is why we have used them to formulate and verify functional requirements in Sect. 9.1.

As such, the modeler can add property rules to the semantics specification which describe the (global) state she is interested in, and then define rule breakpoints using these property rules, including conditions.

10.2.2 Model Execution Process

Debugging a model now means to take control of the graph transformation process; the idea is to "walk" a certain path through the transition system, visualizing execution steps as explained in Sect. 10.1.2 and letting the user

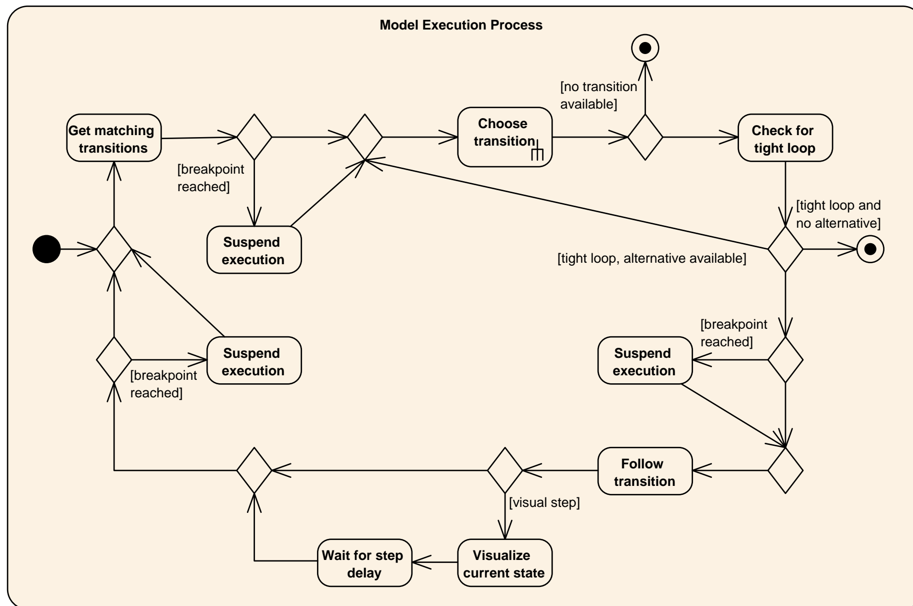

Figure 10.16: Model Execution Process.

control the path at decision points as described in Sect. 10.1.3. The underlying execution process is depicted as Fig. 10.16.

The process starts with getting the transitions originating from the initial runtime state of our model. Since a breakpoint can already apply when its rule only matches the current state, this is the first time to check whether a breakpoint is indeed hit. If this is the case, the execution is suspended.

Otherwise, the DMM Player chooses a transition. This is the most complex of the process's actions: The algorithm to choose a transition as described in Sect. 10.1.3 is executed here, and if it results in a user choice, the user is asked to make her selection. If no transition has been found, the execution is ended.

Otherwise, the algorithm checks for *tight loops*. To understand why this is necessary, consider a DMM specification and model which give rise to a transition system containing loops. This is in general perfectly fine and will quite often be the case. However, when studying the execution of a model by means of animated concrete syntax, the user can only recognize such a loop if it results in at least two visually distinct steps of execution; otherwise, the model will not change, and the user will have to study the executed rules to figure out the reason for this, which can be quite cumbersome. This situation is called a tight loop. Tight loops can be detected by keeping track of the states and visual steps; if the same state is seen with less than two visual steps in between, a tight loop is found.

If a tight loop has been detected, the algorithm tries to choose a different transition; if this is not possible, execution is ended (and the user is notified of the cause). Otherwise, or if no tight loop had been detected, we have finally found the transition to follow. This is again time to check for hitting breakpoints (this time we consider the ones being configured to match "Before application").

Then, the transition is followed. Now, if the transition corresponds to a

visual step, the visualization is updated to reflect the new model state; after waiting for the configured delay, the execution is continued.

Finally, it is once more checked whether any breakpoints are hit, this time for the ones configured as "After application". Afterwards, the process is started over by computing the next set of transitions.

10.2.3 Debugging Models

Finally, let us show how the above concepts can be applied to improve a model's quality. This is now rather straight-forward:

1. First, the functional requirements which the model shall fulfill are formulated using generalized (E)PPSL as described in Sect. 9.1.
2. Then, the GROOVE model checker is used to verify the requirements. In case all requirements hold, we are done. Otherwise, GROOVE will deliver a counter example, i.e., a path through the model's transition which violates the requirement.
3. The counter example is visualized by means of the DMM Player. To understand the issue at hand, the debugging facilities as provided above can be applied.
4. The model is modified to correct the problems identified in step 3.
5. The requirements are verified again (i.e., we continue with step 2). Note that since we have changed the model, we also have to verify the requirements again that were fulfilled.

Figure 10.17 shows a screenshot of the complete DMM debugging facilities. In the center is the augmented UML activity editor showing a simple activity, `Action` "b" of which is currently executed. To the top of the editor's tool panel, the augmented tools for creating `Tokens` and `Offers` are shown.

Right to the activity editor, the property view can be seen, which allows to inspect the properties of the model's elements; it shows the property of the `Offer` selected in the activity editor. Above the activity editor, the Eclipse debug view is located, which provides controls to pause and re-start model execution as well as to step through a model.

For the current debug session, one breakpoint has been defined for rule action.start()#. That breakpoint can be seen within the Eclipse breakpoint view to the right of the Eclipse debug view. Finally, to the very bottom, the console shows the log output produced by the DMM Player; here, all executed rules can be seen.

10.3 Implementation

This section will give insights into our implementation of the concepts described in the last section. However, our explanations are rather high-level – the reader interested in more technical details is pointed to [11]. A high-level view of the DMM Player's architecture is provided as Fig. 10.18.

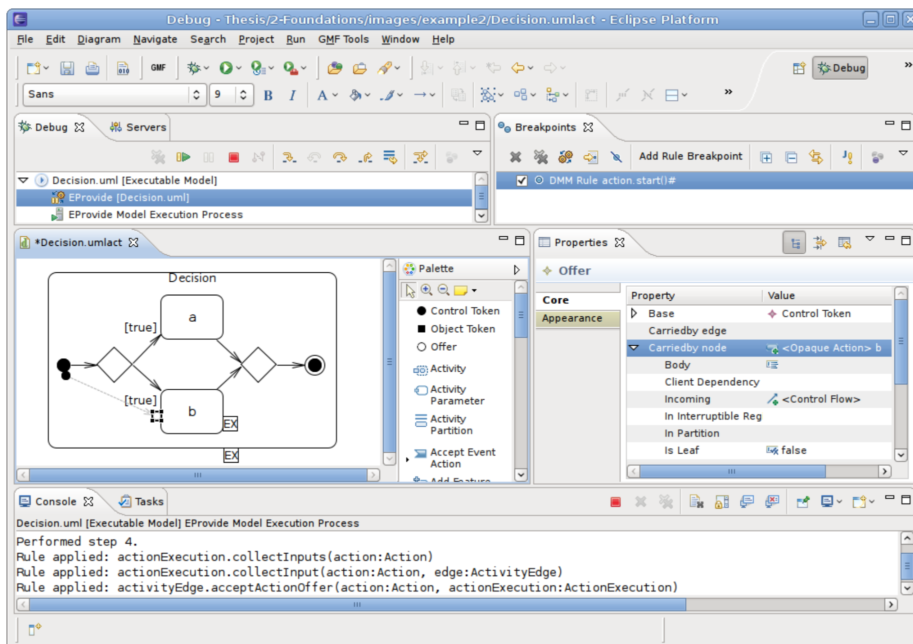

Figure 10.17: Screen shot of the DMM Player in debug mode with the debug controls, the breakpoints view, the properties editor for the selected offer and the execution log in the console.

As mentioned above, the DMM Player builds upon Eclipse technologies; still, the concepts of DMM are completely technology-independent. The implementation can be divided into two mostly independent parts: The diagram augmentation and the model execution. Both are connected by the EMF [44] model just using its standard interfaces; the model execution process changes the model. The diagram augmentation part listens for such changes and updates the visualization accordingly.

The model execution part utilizes EProvide [179], a generic framework for executing behavioral models inside of Eclipse. EProvide decouples the actual execution semantics and the method to define them using two layers:

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

EProvide essentially acts as an adaptor of the semantics description languages to the Eclipse UI on one side and EMF-based models which shall be executed on the other side. The DMM Player code receives commands from EProvide along with the model to be executed and the semantics specification to be used. The most important command is the step, i.e., the command to execute the next atomic step in the given model. The DMM implementation realizes that step by letting the backing graph transformation tool GROOVE [166] perform the application of the according rule, and by translating the ma-

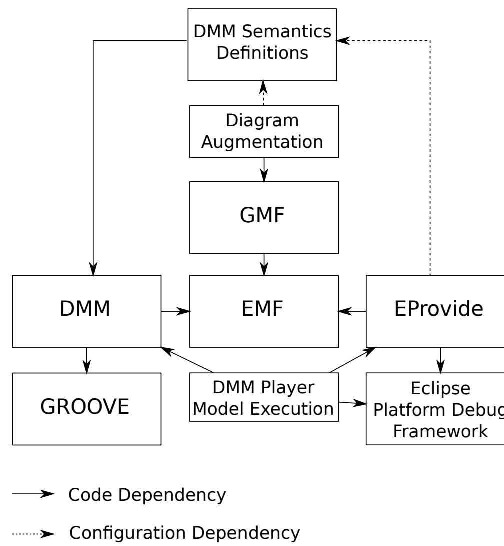

Figure 10.18: Architecture of DMM tooling.

nipulations of the GROOVE rule back to the EMF model which is visualized.

The advanced features, such as the definition of visual steps—which actually combines multiple steps into single ones—, the user control of execution paths, and the debugging functionality, are realized directly by the DMM Player. The EProvide module MODEF [22] also offers debugging functionality which, however, could not be directly utilized, as it makes a quite strong assumption. It is assumed that the model's state can be deduced from one single model element. Since this is not the case with DMM, where a state is a complete model, we needed to bypass this module.

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

The diagram augmentation part of the DMM Player uses extension points of the GMF [50] for extending existing diagram editors. GMF offers quite extensive and flexible means for customizing editors using extension points and factory and decorator patterns.

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

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

Thus, the DMM Player provides a completely declarative, model driven way of augmenting diagram editors; there is no need to writing new or altering existing source code. Figure 10.11 shows screenshots of the activity diagram editor that comes with the Eclipse UML2 Tools [52] which has been augmented by runtime elements using the DMM Player.

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

10.4 Related Work

The first work obviously related to ours is the EProvide framework [179], which we have used for the realization of the DMM Player. EProvide allows to associate some kind of execution semantics with a GMF-based editor (in our case, that semantics is of course the DMM semantics specification); as soon as the backing model changes, the model's concrete syntax is updated accordingly. However, EProvide makes a quite strong assumption within its debugging component MODEV: It is assumed that the model's state can be deduced from one single model element. Since this is not the case with DMM, where a state is a complete model, we had to adjust EPRovide quite a bit.

The scientific work related to ours can mainly be grouped into two categories: Visualization of program execution and animation of visual languages. For the former, an interesting tool is eDOBS [79], which is part of the Fujaba tool suite. eDOBS can be used to visualize a Java program's heap as an object diagram, allowing for an easy understanding of program states without having to learn Java syntax; as a result, one of the main usages of eDOBS is in education. In contrast to that, the concrete syntax of such eDOBS visualizations is fixed (i.e., UML object diagrams), whereas in our approach, the modeler has to come up with his own implementation of the concrete syntax, but is much more flexible in formulating it.

In the area of graph grammars and their applications, there are a number of approaches related to ours: For instance, in [144, 13, 66], the authors use GTRs to specify the abstract syntax of the language under consideration and operations allowed on language instances. The main difference to our approach is that in [144, 13], the actual semantics of the language for which an editor/simulator is to be modeled is not as clearly separated from the specification of the animation as in DMM, where the concrete syntax just reflects what are in fact model changes caused solely by (semantical) DMM rules. In the Tiger approach [66], a GEF [49] editor is generated from the GTRs such that it only allows for edit operations equivalent to the ones defined as GTRs; however, Tiger does not allow for animated concrete syntax.

Another related work is [33]; the DEViL toolset allows to use textual DSLs to specify abstract and concrete syntax of a visual editor as well as the language's semantics. From that, a visual editor can be generated which allows to create, edit, and simulate a model. The simulation uses smooth animations based on *linear graphical interpolation* as default; only the animation of elements which shall behave differently needs to be specified by the language engineer.

There is one major difference from all approaches mentioned to ours: As we

have seen in Sect. 10.1.1, DMM allows for the easy reuse of existing (GMF based) editors. As a result, the language engineer only has to create the concrete syntax for the runtime elements not contained in the language's syntax metamodel, in contrast to the above approaches, where an editor always has to be created from scratch; reusing and extending an existing editor at runtime is not possible.

Summary of Part IV

In this part, we have explained how to formulate and analyze functional as well as non-functional requirements against behavioral models whose semantics is defined by means of a DMM specification, and we have shown how to fix flaws of such models with debugging techniques.

Chapter 9 has covered the topic of requirements formulation and analysis. We have first treated functional requirements: In Sect. 9.1 we have given an introduction to the visual languages PPSL and EPPSL which are dedicated at formulating functional requirements against business processes by means of intuitive visual expressions, which are then translated into according temporal logic formulas. We have then generalized the two languages for the sake of being able to formulate requirements against all kinds of languages (not only business processes), and we have shown how to integrate the concept of property rules into the above to be able to refer to basically arbitrary object structures within the generalized (E)PPSL expressions.

Section 9.2 was concerned with non-functional properties. Our approach is to add performance information to the DMM semantics specification and model at hand, and then to automatically derive a PEPA model, i.e., an instance of the Performance Evaluation Process Algebra. This allows to analyze non-functional properties with the standard PEPA tooling (which is—as the DMM tooling—based on Eclipse and thus integrates nicely).

Finally, Chapter 10 has shown how models can be debugged, e.g. to understand and correct errors revealed with the techniques of the last chapter. In Sect. 10.1, we have explained our strictly model-driven approach to visualize runtime information: An augmentation model defines the concrete syntax of the runtime elements and their visual relations to elements of the static structure (e.g., whether a runtime elements' visual representation is contained within the visual representation of a syntax element or attached to its border), a rule step model defines the visual steps to be shown during animation of a model's execution, and a switch model marks elements such as `DecisionNodes` to cause the user to selected the execution alternative. At runtime, the model's execution is visualized within existing GMF editors which are augmented with the runtime elements.

Section 10.2 has then explained our concepts for debugging models: Breakpoints and watchpoints can be defined by binding them to rules; if one of these rules is to be applied, execution is suspended, and the model's runtime state can be examined. Every model can be debugged this way, but of particular interest are models which have failed functional requirements: The resulting counter example can be visualized to understand the cause of the problem.

The results of this part are

- a method to augment existing visual GMF editors with runtime information which can then be used to visualize a model's execution by means of animation,
- the possibility to define breakpoints and watchpoints which—when hit—suspend a model's execution and allow to investigate the model's runtime state,
- and an integration of the above with the GROOVE model checker, counter

examples of which can be visualized for the sake of understanding and correcting the underlying problems.

As such, this part of the thesis has covered the complete model quality lifecycle, from formulating requirements to their verification and correction.

Summary and Outlook

Summary

Dynamic Meta Modeling is a semantics specification language targeted at behavioral languages whose syntax is defined by means of a metamodel. It has first been suggested by Engels et al. [63]; the actual conceptual base has then been created by Hausmann [96], who has defined the DMM language by means of mathematical terms, and who has provided examples of the visual concrete syntax of DMM as well as ideas on how to translate DMM specifications into GROOVE grammars. As such, DMM allows to formalize a language's behavior in terms of an easily understandable visual semantics specification, with all the benefits such as preciseness and analyzability.

Obviously, the next step had to be to make DMM alive by creating appropriate tooling. Based on that, concepts and tools had to be developed to make use of DMM's benefits: In particular, it had to be investigated how—given an according DMM specification—to perform quality assurance on a given model. This is the main topic of this thesis.

To realize this goal, we had to overcome three main problems: First, we needed to build *tool support* from scratch; the second problem was that quality analysis of a model based on an erroneous semantics specification would render that analysis nearly useless, meaning that we had to develop ways to guarantee the *quality of the DMM semantics specifications.* Finally, the last problem—and the one we are eventually interested in—was how to formulate and then verify requirements against models equipped with a DMM specifications, and how to fix possibly exisiting flaws of the models.

We were able to provide rich visual editors for DMM specifications mainly because of the advances of model-driven development; in particular, we have defined the DMM language by means of an Ecore metamodel, and based on that, we have used EMF-based frameworks such as the Eclipse Graphical Modeling Framework to model and then generate and customize our editors. The resulting tooling supports different views on the edited DMM specification and allows for the validation of the language's static semantics by means of OCL constraints, violations of which are displayed as annotations in the editors.

In parallel, we have implemented Java-based model transformations from DMM specifications (i.e., instances of the DMM metamodel) into GROOVE rules and from EMF models (i.e., instances of arbitrary Ecore metamodels) into GROOVE graphs, allowing us to use GROOVE's capabilities of exploring a model's state space and performing model checking on that state space. Note

that by doing that, we have re-defined the core of DMM: The semantics of a DMM rule now is the semantics of the resulting GROOVE rule.

We then had the basic building blocks in place to tackle the actual problems of the thesis: Ensuring the quality of DMM semantics specifications as well as models. For the former, we have developed a process called *test-driven semantics specification*, where example models and their expected behavior are encoded into test cases; during creation of the DMM specification, the tests are executed, which basically comes down to checking for each example model whether the semantics specification produces exactly the desired behavior.

Knowing that a DMM specification produces the correct behavior for its test cases indeed gives some confidence about the specification's quality. However, it is desirable to be able to measure the quality of a specification's tests; otherwise, parts of our semantics specification might remain untested even with a high number of tests. Therefore, we have developed a set of six *coverage criteria* for tests of DMM specifications. The criteria are defined on so-called *invocation graphs*, where nodes are rule applications, and edges are consecutive applications of rules.

While developing actual DMM specifications, it turned out that reusability of those specifications is hampered by the fact that DMM only allowed to *add* rules to an existing DMM specification, but not to *refine* rules. Thus, we have developed and integrated the concept of *rule overriding*, where a DMM rule can override another DMM rule such that only the overriding rule will be applied (even if the overridden rule also matches).

Finally, we turned our attention to the quality assurance of models. Since DMM is supposed to be easily understandable, our first step was to overcome the need to study a model's state space within GROOVE to understand that model's behavior. Thus, we have developed the *DMM Player* which allows to execute a model visually by means of animated concrete syntax. The challenge was to make it easy for the the language engineer to add the necessary visualization animation; we have solved this by reusing existing visual editors which are augmented with visual constructs representing the runtime elements of the executed model. The information needed for this augmentation (and other information such as step definitions) are provided by the language engineer as models; there is no need to write additional (or even change existing) code.

It remained to provide means to formulate requirements, analyze them, and—if necessary—improve the according models. Once more to support understandibility, we have used visual languages, in this case for the specification of functional requirements: The *pattern process specification language* as well as its extension, the *enhanced pattern process specification language* are dedicated to formulate functional requirements against business processes. We have generalized these languages to be able to formulate requirements against arbitrary languages. Expressions of the language are translated into temporal logic formulas which can then be model checked with GROOVE. We have demonstrated this approach using the example requirement of *soundness*.

In terms of non-functional requirements, our approach was to add performance information to existing DMM specifications and models by means of decoration, leaving the specification and model unchanged; that information is then used to generate an instance of the *performance evaluation process algebra* (PEPA), which can be analyzed for non-functional requirements such as average throughput with the existing PEPA tooling.

In a nutshell, we have seen the following contributions in this thesis:

- The definition of DMM++ as an extension to DMM, including concepts to refine DMM rules and a semantics defined by transforming a DMM specification into a GROOVE grammar (Part II).
- A test-driven semantics specification process including test coverage computation and tooling for creating high-quality semantics specifications (Part III).
- A visual language for the formulation of functional requirements, performance analysis of models through the PEPA tooling, and a model-driven means to equip languages with animated concrete syntax (Part IV).

The result of this thesis are concepts and tools which—with the exceptions of requirements specification and syntax definition—support the complete lifecycle of behavioral languages, from specifying a language's semantics to analyzing and improving the quality of models.

As such, we hope that we have contributed to the field of model-driven engineering in a useful way.

Outlook

As we have seen (not only) in the summary, this PhD project has touched a lot of different areas of semantics specification; therefore, many approaches presented within this thesis (and especially the implementations of those approaches) have a proof-of-concept character. In this section, we will point out open issues of the DMM approach in general and the different concepts it consists of.

First of all, the DMM approach claims that its specifications are easily understandable due to their visual, the language metamodel using appearance. However, it has never been thoroughly investigated whether this is indeed the case. For instance, textual specification languages tend to be more compact, thus maybe allowing for an easier overview of a complete semantics specification. Now that a rich DMM tooling is available, it might be the right time to study this question in an empirical way.

Our own experience with respect to this question is two-fold: The students who were working with DMM and creating rather complex semantics specifications (see [105, 180, 149]) were on the one hand able to pretty quickly understand the general DMM concepts; on the other hand, they had problems managing the complexity of their specifications. One important point was *changeability*: For instance, changing the name of a smallstep rule implies that all according invocations have to be changed (and, in addition, other smallstep rules carrying the same name); it is quite easy to oversee some invocations, ending up with an erroneous semantics specification. As such, one area where serious work is necessary is *refactoring of DMM specifications*. Note that the results of an empiric study on DMM usability could (and should) be respected by such research (and vice versa).

Another important area is *scalability*. We have seen that the underlying formalism of DMM, graph transformations, has a rather high computational complexity, which makes the general model checking problem of *state explosion* even worse. As such, all measures should be taken to improve DMM's scalability. There are several existing approaches which could be investigated for applicability in the DMM scenario; for instance, the work on *graph abstraction* (see e.g. [215]) looks promising; another area is *(de)composition*, e.g. Heckel [97] has described the compositional verification of reactive systems the semantics of which is described by graph transformations.

The approach of test-driven semantics specification has proven to be very helpful when creating DMM specifications. However, there is still room for improvement in the area of coverage computation. First, our current approach only considers the invocations of smallstep rules inside a single bigstep rule; however, the relation of the bigstep rules to each other is not taken into consideration. Further work needs to be performed in this area; one possible approach could be the computation of *critical pairs* (see e.g. [138]) of bigstep rules (or some generated rules which take a bigstep rule's invocations' changes into account).

We have seen that invocation graphs contain pathes of execution which are in fact not possible; this is because the computation of invocation graphs does—despite the invocations—not take the rules' content into account. For instance, consider a bigstep rule invoking a smallstep rule `s`, for which two implementations exist. Thus, the invocation graph will contain both possible execution paths. However, it might be (and will quite often be) the case that the big-

step rule's structure only allows for the matching of one of the two smallstep rules. Critical pair analysis might be helpful with this respect, too, by reducing the number of execution paths by removing impossible sequences of rule applications.

Finally, research should be performed on how to improve the feedback of the coverage tooling. In its current state, the feedback is very technical; basically, a set of rule or invocation sequences is provided which is not covered, but no feedback is given on how to improve on the situation. It might be possible to analyze the rules contained in such uncovered sequences, and to suggest models or model parts as result of that analysis which will cause the according sequences to be covered.

In the area of formulating and verifying functional requirements, our first paragraph of this section basically holds once more: To our knowledge, it is yet to be shown how helpful the usage of the *(enhanced) process pattern specification language* is in practice (although the performed case studies suggest that it indeed is), and the same of course holds for our generalization of that language. Such research could (and once more: should) be conducted together with the empirical studies on the general usability of DMM.

Additionally, neiter (E)PPSL nor our generalization support the full expressiveness of LTL/CTL (in contrast to e.g. attempts such as [40]). Since the application range of generalized (E)PPSL (all kinds of languages) is much larger than that of (E)PPSL (business processes), it should be investigated whether its expressiveness is still sufficient.

Our work on doing performance analysis on models with DMM specifications nicely separates the model and its execution semantics from the performance information and allows for powerful performance descriptions which take the model's runtime states into account. However, we do not yet support the addition of probability information. This problem could probably be tackled by defining a notion of DMM probability models, to compute the probability of each transition of the model's transition system, and to then compute the final performance rates while taking the transitions' probabilities into account.

The DMM Player and the surrounding framework fulfill the basic task of allowing to describe the necessary information for visual executing "token-based" languages such as UML activities, Petri nets, or UML statemachines (for the latter, the markers of the active states can be seen as tokens). In general, the DMM Player's configuration models have been designed to fulfill more use cases; however, this is yet to be shown.

Additionally, it might be a good idea to consider that different people working with DMM might want to see different amounts of detail while simulating a model. For instance, the language engineer probably wants to see more execution details while developing the semantics of a language than end users. Moreover, there might even be people which are only interested in an even higher view of a model's behavior; for instance, they might not care about the location of the offers in the case of UML activities. To suite the needs of these different kinds of users, an area of research could be to extend our approach such that the augmentation and rulestep models can be *refined*. This would allow to start with a specification of the visualization which reveals all execution details, and then to refine that specification step by step, each refinement fulfilling the information needs of a different kind of language users.

List of Figures

2.1 A metamodel and one of its instances 10
2.2 The four-layer metamodel hierarchy 11
2.3 Simplified Ecore metamodel 13

3.1 Example UML activity modeling a workflow 18
3.2 UML activity packages and their dependencies 18
3.3 Simplified metamodel of UML activities 20

4.1 GROOVE host graph representing a Petri net 26
4.2 GROOVE rule describing the semantics of a Petri net transition 27
4.3 Labeled transition system of a Petri net 28
4.4 GROOVE rule describing the semantics of a customized transition 29
4.5 Labeled transition system of a customized Petri net 30
4.6 Counter example for property **AG**(!fire("A")) 33
4.7 Screenshot of the GROOVE simulator 34

5.1 Overview of the DMM approach 39
5.2 Excerpt of the UML activity runtime metamodel 40
5.3 Excerpt of the UML activity meta relations 41
5.4 Metamodel of meta relations 41
5.5 Example DMM rule initialNode.createToken() 42
5.6 Example DMM rule fork.getOffer()* 47
5.7 Graph concepts in DMM and GROOVE 48
5.8 DMM invocation stack 50
5.9 Application control 51

6.1 DMM Metamodel: Ruleset view 60
6.2 Package Notation 63
6.3 DMM Metamodel: Rule hierarchy view 64
6.4 Bigstep rule 66
6.5 Smallstep rule 67
6.6 Premise rule 68
6.7 Property rule 69
6.8 Soft rule overriding 70
6.9 Complete rule overriding 71
6.10 DMM Metamodel: Internal rule structure view 71
6.11 Internal structure of rule activityExecution.start()# 72
6.12 Internal structure of rule action.start()# 72
6.13 DMM Metamodel: Expression language 80
6.14 Transformation Overview 88

6.15 Empty invocation stack . . . 93
6.16 Bigstep rule with invocation . . . 94
6.17 Invocation stack of bigstep rule . . . 94
6.18 Smallstep rule with invocation . . . 95
6.19 Invocation stack of smallstep rule . . . 96
6.20 DMM rule with UQS . . . 97
6.21 GROOVE rule with UQS . . . 98
6.22 Smallstep rule with invocation . . . 99
6.23 Invocation stack of smallstep rule . . . 99
6.24 Helper rule inserting an invocation into the stack . . . 100
6.25 Helper rule removing the UQSInvocation from the stack . . . 100
6.26 GROOVE state graph with enumeration . . . 102
6.27 Syntax metamodel of UML Activities . . . 105
6.28 Runtime metamodel of UML Activities . . . 105
6.29 DMM rule action.execute(ActivityExecution)# . . . 106
6.30 Metamodel extending the syntax and runtime metamodels . . . 106
6.31 DMM rule extendedInitialNode.flow()# . . . 107
6.32 DMM rule extendedAction.execute(ActivityExecution) . . . 107
6.33 GROOVE rule resulting from an overridden rule . . . 112
6.34 GROOVE state after application of rule action.flow()# . . . 113
6.35 GROOVE rule with soft rule overriding structure . . . 114
6.36 GROOVE rule moving the `activated` edge to the next level . 114
6.37 Metamodel with complex inheritance hierarchy . . . 115
6.38 Sequence of states of the rule hierarchy graph . . . 116
6.39 Labeled transition system showing the matching of the rules . . 117

7.1 “From Scratch” approach: Runtime metamodel . . . 131
7.2 “From Scratch” approach: Tool support . . . 132
7.3 Transformation generation ruleset . . . 133
7.4 Main generation rule . . . 133
7.5 `EClass` correspondence generation rule . . . 134
7.6 `EClass` rule generation rule . . . 135
7.7 Generated transformation ruleset . . . 136
7.8 Generated rule for `Activity` . . . 137
7.9 Generated rule for reference `Activity::nodes` . . . 137
7.10 Modified runtime metamodel . . . 138
7.11 Refined rule for `Activity` . . . 138
7.12 “Decorator” approach: Runtime metamodel . . . 139
7.13 “Decorator” approach: DMM-based transformation . . . 140
7.14 DMM Workbench – Ruleset view . . . 142
7.15 DMM Workbench – Rule view . . . 142
7.16 DMM Workbench – “Open invoked rule” dialog . . . 143

8.1 Comparison of software testing and semantics testing . . . 148
8.2 Process of creating example models . . . 149
8.3 Example Activity containing only one `Action` . . . 150
8.4 Example Activity with a simple decision/merge structure . . . 150
8.5 Example Activity containing a loop . . . 151
8.6 Metamodel of Traces language . . . 153
8.7 Example Traces model . . . 154

8.8 Evaluation of Traces model 154
8.9 Specify semantics, create test cases from example models 155
8.10 Invocation graph for rule action.start()# 160
8.11 Invocation graph for rule forkNode.flow()# 161
8.12 Invocation graph for rule inputPin.supplyStreamingToken()# . . . 162
8.13 Excerpt of transition system 165
8.14 Excerpt of transition system 165
8.15 Invocation graph for rule action.start()# 166
8.16 Excerpt of transition system 167
8.17 Excerpt of transition system 168
8.18 Excerpt of transition system 170
8.19 Coverage Hierarchy 175
8.20 Invocation graph with potentially dead edges 176

9.1 Example business process 186
9.2 Example visual process pattern 186
9.3 Overview of the PPSL approach 188
9.4 Temporal operators provided by EPPSL 188
9.5 Building blocks provided by EPPSL 189
9.6 Logical operators provided by EPPSL 189
9.7 Example EPPSL expression 189
9.8 Translation of EPPSL patterns 190
9.9 Example EPPSL for state machines 192
9.10 Example property rule state.A_and_B()! 193
9.11 Example EPPSL for state machines 193
9.12 Generalized EPPSL expression for requirement 4 on page 184 . 194
9.13 Generalized EPPSL expression for requirement 3 on page 184 . 194
9.14 Property rule finalNode.ERROR()! 195
9.15 Generalized EPPSL expression for requirement 2 on page 184 . 195
9.16 Example business process 197
9.17 The syntax of PEPA 199
9.18 Example PEPA process 200
9.19 Example PEPA process (graph representation) 200
9.20 DMM performance metamodel 202
9.21 Performance model with a `SimplePerformance` 203
9.22 Performance model with a `ParameterizedPerformance` . . 204
9.23 PM with a `ParameterizedPerformanceWithContext` . . 205
9.24 Connection between DMM performance model and state model 207
9.25 PEPA generation workflow 207
9.26 PEPA analysis tooling – Throughput view 210

10.1 An UML activity diagram with additional runtime elements . . 217
10.2 Metamodel for diagram augmentation models 218
10.3 Excerpt of the augmentation model for UML activities 219
10.4 A sequence of states exhibiting a temporary inconsistency . . . 220
10.5 Step definition metamodel 221
10.6 Example UML activity with concurrency and decisions 223
10.7 DMM Rule decisionNode.flow()# 223
10.8 Transition system, rules, and morphisms 224
10.9 Algorithm for selecting a transition to be followed 226

10.10 Metamodel for describing rules which indicate decisions 226
10.11 UI for choosing execution paths 227
10.12 Example state of Statemachine execution 228
10.13 Excerpt of the augmentation model for UML Statemachines . . 229
10.14 Eclipse breakpoint view with a single DMM breakpoint 231
10.15 DMM breakpoint configuration dialog 232
10.16 Model Execution Process 233
10.17 Screen shot of the DMM Player in debug mode 235
10.18 Architecture of DMM tooling 236

List of Listings

8.1 Algorithm for computing the invocation graph of a bigstep rule . 162
8.2 General algorithm for coverage computation 164
8.3 Subroutines for computing rule coverage 166
8.4 Subroutines for computing rule coverage plus 167
8.5 Routines to be reused by the coverage algorithms 168
8.6 Subroutines for computing rule coverage plus plus 169
8.7 Subroutines for computing edge coverage 171
8.8 Subroutines for computing edge coverage plus 172
8.9 Subroutines for computing edge coverage plus plus 173
9.1 Algorithm for generating PEPA model from transition system . . 201
9.2 Algorithm for evaluating DMM performance definitions 206

List of Tables

6.1 DMM expression language: Lexical Tokens 80
6.2 DMM expression language: Grammar 81
6.3 DMM expression language: Operators 86
6.4 Java and GROOVE datatypes 102
6.5 DMM operators and GROOVE productions 103

9.1 Average execution times of example business process 198
9.2 Average execution times and rates of example business process . 208
9.3 Effects of example process improvements 209

Bibliography

[1] Abstract Solutions. iUML Modeler and Simulator. http://www.kc.com/PRODUCTS/iuml/. Online, accessed 5–14–2013.

[2] C. Amelunxen, A. Königs, T. Rötschke, and A. Schürr. MOFLON: A Standard-Compliant Metamodeling Framework with Graph Transformations. In A. Rensink and J. Warmer, editors, *Proceedings of ECMDA '06*, volume 4066 of *LNCS*, pages 361–375, Berlin/Heidelberg, 2006. Springer.

[3] T. Arendt, E. Biermann, S. Jurack, C. Krause, and G. Taentzer. Henshin: Advanced Concepts and Tools for In-Place EMF Model Transformations. In D. C. Petriu, N. Rouquette, and Ø. Haugen, editors, *MoDELS (1)*, volume 6394 of *LNCS*, pages 121–135. Springer, 2010.

[4] S. Arifulina. Coverage Criteria for Testing DMM Specifications. Master's thesis, University of Paderborn, 2011.

[5] S. Arifulina, C. Soltenborn, and G. Engels. Coverage Criteria for Testing DMM Specifications. In *Proceedings of the 11th International Workshop on Graph Transformation and Visual Modeling Techniques (GT-VMT 2012), Tallinn (Estonia)*, volume 47 of *Electronic Communications of the EASST*. European Association of Software Science and Technology, 2012.

[6] N. Arijo, R. Heckel, M. Tribastone, and S. Gilmore. Modular Performance Modelling for Mobile Applications. In S. Kounev, V. Cortellessa, R. Mirandola, and D. J. Lilja, editors, *ICPE*, pages 329–334. ACM, 2011.

[7] U. Aßmann. Graph Rewrite Systems for Program Optimization. *ACM Trans. Program. Lang. Syst.*, 22(4):583–637, 2000.

[8] C. Atkinson and T. Kühne. Model-driven Development: A Metamodeling Foundation. *Software, IEEE*, 20(5):36 – 41, 2003.

[9] P. Baldan, A. Corradini, and U. Montanari. Concatenable Graph Processes: Relating Processes and Derivation Traces. In K. G. Larsen, S. Skyum, and G. Winskel, editors, *ICALP*, volume 1443 of *LNCS*, pages 283–295, Berlin/Heidelberg, 1998. Springer.

[10] Z. Balogh and D. Varró. Model Transformation by Example using Inductive Logic Programming. *Software and System Modeling*, 8(3):347–364, 2009.

[11] N. Bandener. Visual Interpreter and Debugger for Dynamic Models Based on the Eclipse Platform. Diploma thesis, University of Paderborn, 2009.

[12] N. Bandener, C. Soltenborn, and G. Engels. Extending DMM Behavior Specifications for Visual Execution and Debugging. In M. v. d. B. B. Malloy, S. Staab, editor, *Proceedings of the 3rd International Conference on Software Language Engineering (SLE 2010)*, volume 6563 of *LNCS*, pages 357–376, Berlin/Heidelberg, 2011. Springer.

[13] R. Bardohl, C. Ermel, and I. Weinhold. GenGED – A Visual Definition Tool for Visual Modeling Environments. In J. L. Pfaltz, M. Nagl, and B. Böhlen, editors, *Proceedings of AGTIVE '03*, volume 3062 of *LNCS*, pages 413–419, Berlin/Heidelberg, 2003. Springer.

[14] L. Baresi, R. Heckel, S. Thöne, and D. Varró. Style-based Refinement of Dynamic Software Architectures. In *Software Architecture, 2004. WICSA 2004. Proceedings. Fourth Working IEEE/IFIP Conference on*, pages 155–164, 2004.

[15] L. Baresi, V. Rafe, A. T. Rahmani, and P. Spoletini. An Efficient Solution for Model Checking Graph Transformation Systems. *Electronic Notes in Theoretical Computer Science*, 213(1):3 – 21, 2008. Proceedings of the Third Workshop on Graph Transformation for Concurrency and Verification (GT-VC 2007).

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

[17] E. Bauer, J. M. Küster, and G. Engels. Test Suite Quality for Model Transformation Chains. In *TOOLS (49)*, pages 3–19, 2011.

[18] J. Bauer, I. Boneva, M. E. Kurbán, and A. Rensink. A Modal-Logic Based Graph Abstraction. In Ehrig et al. [58], pages 321–335.

[19] K. Beck. *Test-Driven Development by Example.* Addison-Wesley Longman, Amsterdam, The Netherlands, 2002.

[20] S. Becker, H. Koziolek, and R. Reussner. The Palladio Component Model for Model-driven Performance Prediction. *Journal of Systems and Software*, 82:3–22, Jan. 2009.

[21] J. Bengtsson, K. Larsen, F. Larsson, P. Pettersson, and W. Yi. UPPAAL – A Tool Suite for Automatic Verification of Real-time Systems. In R. Alur, T. Henzinger, and E. Sontag, editors, *Hybrid Systems III*, volume 1066 of *LNCS*, pages 232–243. Springer, Berlin/Heidelberg, 1996.

[22] A. Blunk, J. Fischer, and D. A. Sadilek. Modelling a Debugger for an Imperative Voice Control Language. In R. Reed, A. Bilgic, and R. Gotzhein, editors, *Proceedings of SDL 2009*, volume 5719 of *LNCS*, pages 149–164, Berlin/Heidelberg, 2009. Springer.

[23] P. Bottoni, M. Koch, F. Parisi-Presicce, and G. Taentzer. Termination of High-Level Replacement Units with Application to Model Transformation. *Electr. Notes Theor. Comput. Sci.*, 127(4):71–86, 2005.

[24] M. Bouarioua, A. Chaoui, and R. Elmansouri. From UML Sequence Diagrams to Labeled Generalized Stochastic Petri Net Models Using Graph Transformation. In J. Yonazi, E. Sedoyeka, E. Ariwa, and E. El-Qawasmeh, editors, *e-Technologies and Networks for Development*, volume 171 of *Communications in Computer and Information Science*, pages 318–328. Springer, Berlin/Heidelberg, 2011.

[25] S. Burmester, H. Giese, M. Hirsch, D. Schilling, and M. Tichy. The FUJABA Real-time Tool Suite: Model-driven Development of Safety-critical, Real-time Systems. In G.-C. Roman, W. G. Griswold, and B. Nuseibeh, editors, *ICSE*, pages 670–671. ACM, 2005.

[26] J. M. Chiaradía and C. Pons. Improving the OCL Semantics Definition by Applying Dynamic Meta Modeling and Design Patterns. In *OCL for (Meta-)Models in Multiple Application Domains 2006*, volume 5 of *Electronic Communications of the EASST*. European Association of Software Science and Technology, 2006.

[27] A. Cimatti, E. M. Clarke, E. Giunchiglia, F. Giunchiglia, M. Pistore, M. Roveri, R. Sebastiani, and A. Tacchella. NuSMV 2: An OpenSource Tool for Symbolic Model Checking. In E. Brinksma and K. G. Larsen, editors, *CAV*, volume 2404 of *LNCS*, pages 359–364, Berlin/Heidelberg, 2002. Springer.

[28] S. Cook, G. Jones, S. Kent, and A. Wills. *Domain-Specific Development with Visual Studio DSL Tools*. Addison-Wesley Professional, 2007.

[29] A. Corradini, H. Ehrig, M. Löwe, U. Montanari, and J. Padberg. The Category of Typed Graph Grammars and its Adjunctions with Categories. In J. E. Cuny, H. Ehrig, G. Engels, and G. Rozenberg, editors, *TAGT*, volume 1073 of *LNCS*, pages 56–74, Berlin/Heidelberg, 1994. Springer.

[30] A. Corradini, H. Ehrig, U. Montanari, L. Ribeiro, and G. Rozenberg, editors. *Graph Transformations, Third International Conference, ICGT 2006, Natal, Rio Grande do Norte, Brazil, September 17–23, 2006, Proceedings*, volume 4178 of *LNCS*, Berlin/Heidelberg, 2006. Springer.

[31] A. Corradini, U. Montanari, F. Rossi, H. Ehrig, R. Heckel, and M. Löwe. Algebraic Approaches to Graph Transformation - Part I: Basic Concepts and Double Pushout Approach. In Rozenberg [177], pages 163–246.

[32] V. Cortellessa and R. Mirandola. Deriving a Queueing Network Based Performance Model from UML Diagrams. In *Proceedings of the 2nd international workshop on Software and performance*, WOSP '00, pages 58–70, New York, NY, USA, 2000. ACM.

[33] B. Cramer and U. Kastens. Animation Automatically Generated from Simulation Specifications. In *Proceedings of VL/HCC '09*. IEEE Computer Society, 2009.

[34] K. Czarnecki and S. Helsen. Feature-based survey of model transformation approaches. *IBM Syst. J.*, 45(3):621–645, July 2006.

[35] K. Czarnecki and S. Helsen. Feature-based Survey of Model Transformation Approaches. *IBM Systems Journal*, 45(3):621–646, 2006.

[36] A. D'Ambrogio. A WSDL Extension for Performance-Enabled Description of Web Services. In P. Yolum, T. Güngör, F. S. Gürgen, and C. C. Özturan, editors, *ISCIS*, volume 3733 of *LNCS*, pages 371–381, Berlin/Heidelberg, 2005. Springer.

[37] A. D'Ambrogio and P. Bocciarelli. A Model-driven Approach to Describe and Predict the Performance of Composite Services. In *Proceedings of the 6th international workshop on Software and performance*, WOSP '07, pages 78–89, New York, NY, USA, 2007. ACM.

[38] J. de Lara, R. Bardohl, H. Ehrig, K. Ehrig, U. Prange, and G. Taentzer. Attributed Graph Transformation with Node Type Inheritance. *Theoretical Computer Science*, 376(3):139–163, 2007.

[39] M. de Mol and A. Rensink. On A Graph Formalism for Ordered Edges. In *Proceedings of GT/VMT 2010*, volume 29 of *Electronic Communications of the EASST*. European Association of Software Science and Technology, 2010.

[40] A. Del Bimbo, L. Rella, and E. Vicario. Visual Specification of Branching Time Temporal Logic. In *Visual Languages, Proceedings., 11th IEEE International Symposium on*, pages 61–68, 1995.

[41] W. Dong, J. Wang, X. Qi, and Z.-C. Qi. Model checking uml statecharts. In *Software Engineering Conference, 2001. APSEC 2001. Eighth Asia-Pacific*, pages 363–370, 2001.

[42] M. B. Dwyer, G. S. Avrunin, and J. C. Corbett. Property Specification Patterns for Finite-State Verification. In *Proceedings of the second workshop on Formal methods in software practice (FMSP '98)*, pages 7–15, New York, NY, USA, 1998. ACM.

[43] T. Eckardt, C. Heinzemann, S. Henkler, M. Hirsch, C. Priesterjahn, and W. Schäfer. Modeling and Verifying Dynamic Communication Structures Based on Graph Transformations. *Computer Science - Research and Development*, 28(1):3–22, Feb. 2013.

[44] Eclipse Foundation. Eclipse Modeling Framework. http://www.eclipse.org/modeling/emf/. Online, accessed 9–1–2010.

[45] Eclipse Foundation. Eclipse Remote Application Platform. http://eclipse.org/rap/. Online, accessed 3–18–2013.

[46] Eclipse Foundation. EMF Compare. http://www.eclipse.org/emf/compare/. Online, accessed 3–18–2013.

[47] Eclipse Foundation. EMF OCL. http://www.eclipse.org/modeling/mdt/?project=ocl. Online, accessed 3–18–2013.

[48] Eclipse Foundation. EMF Validation Framework. http://www.eclipse.org/modeling/emf/?project=validation. Online, accessed 3–18–2013.

[49] Eclipse Foundation. Graphical Editing Framework. http://www.eclipse.org/gef/. Online, accessed 9–15–2010.

[50] Eclipse Foundation. Graphical Modeling Framework. http://www.eclipse.org/modeling/gmf/. Online, accessed 5–5–2009.

[51] Eclipse Foundation. UML2 Project. http://www.eclipse.org/modeling/mdt/?project=uml2. Online, accessed 3–18–2013.

[52] Eclipse Foundation. UML2 Tools. http://www.eclipse.org/modeling/mdt/?project=uml2tools. Online, accessed 9–15–2010.

[53] Eclipse Foundation. Xcore. http://wiki.eclipse.org/Xcore/. Online, accessed 10–16–2012.

[54] Eclipse Foundation. XText. http://www.eclipse.org/Xtext/. Online, accessed 3–18–2013.

[55] H. Ehrig and K. Ehrig. Overview of Formal Concepts for Model Transformations Based on Typed Attributed Graph Transformation. *Electr. Notes Theor. Comput. Sci.*, 152:3–22, 2006.

[56] H. Ehrig, K. Ehrig, J. de Lara, G. Taentzer, D. Varró, and S. Varró-Gyapay. Termination Criteria for Model Transformation. In M. Cerioli, editor, *FASE*, volume 3442 of *LNCS*, pages 49–63. Springer, 2005.

[57] H. Ehrig, G. Engels, H.-J. Kreowski, and G. Rozenberg, editors. *Handbook of Graph Grammars and Computing by Graph Transformation, Vol. 2: Applications, Languages, and Tools*. World Scientific Publishing Co., Inc., River Edge, NJ, USA, 1999.

[58] H. Ehrig, R. Heckel, G. Rozenberg, and G. Taentzer, editors. *Graph Transformations, 4th International Conference, ICGT 2008, Leicester, United Kingdom, September 7-13, 2008. Proceedings*, volume 5214 of *LNCS*. Springer, 2008.

[59] K. Ehrig, E. Guerra, J. D. Lara, L. Lengyel, U. Prange, G. Taentzer, D. Varro, and et al. Model Transformation by Graph Transformation: A Comparative Study. In *Proceedings of MTIP 2005*, pages 71–80, 2006.

[60] E. Emerson and J. Y. Halpern. Decision Procedures and Expressiveness in the Temporal Logic of Branching Time. *Journal of Computer and System Sciences*, 30(1):1–24, 1985.

[61] E. A. Emerson and J. Y. Halpern. “Sometimes” and “not never” revisited: on branching versus linear time temporal logic. *J. ACM*, 33(1):151–178, Jan. 1986.

[62] G. Engels, D. Fisseler, and C. Soltenborn. Improving Reusability of Dynamic Meta Modeling Specifications with Rule Overriding. In M. E. R. DeLine, M. Minas, editor, *Proceedings of the 2009 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC 2009), Corvallis, Oregon (USA)*, pages 39–46, Piscataway, NJ (USA), 2009. IEEE Computer Society.

[63] G. Engels, J. H. Hausmann, R. Heckel, and S. Sauer. Dynamic Meta-Modeling: A Graphical Approach to the Operational Semantics of Behavioral Diagrams in UML. In A. Evans, S. Kent, and B. Selic, editors, *Proceedings of UML 2000*, volume 1939 of *LNCS*, pages 323–337, Berlin/Heidelberg, 2000. Springer.

[64] G. Engels and C. Soltenborn. Test-driven Language Derivation with Graph Transformation-based Dynamic Meta Modeling. In C. Ermel, H. Ehrig, F. Orejas, and G. Taentzer, editors, *Proceedings of the International Colloquium on Graph and Model Transformation (GraMoT 2010), Berlin (Germany)*, volume 30 of *Electronic Communications of the EASST*, pages 240–257. European Association of Software Science and Technology, 2010.

[65] G. Engels, C. Soltenborn, and H. Wehrheim. Analysis of UML Activities using Dynamic Meta Modeling. In M. M. Bosangue and E. B. Johnsen, editors, *Proceedings of FMOODS 2007*, volume 4468 of *LNCS*, pages 76–90, Berlin/Heidelberg, 2007. Springer.

[66] C. Ermel, K. Ehrig, G. Taentzer, and E. Weiss. Object Oriented and Rule-based Design of Visual Languages using Tiger. In *Proceedings of GraBaTs '06*, volume 1 of *Electronic Communications of the EASST*. European Association of Software Science and Technology, 2006.

[67] R. Eshuis. Symbolic Model Checking of UML Activity Diagrams. *ACM Trans. Softw. Eng. Methodol.*, 15(1):1–38, 2006.

[68] R. Eshuis and R. Wieringa. Verification Support for Workflow Design with UML Activity Graphs. In *Proceedings of the 24th International Conference on Software Engineering*, ICSE '02, pages 166–176, New York, NY, USA, 2002. ACM.

[69] M. Eysholdt and H. Behrens. XText: Implement Your Language Faster than the Quick and Dirty Way. In *Proceedings of the ACM international conference companion on Object oriented programming systems languages and applications companion*, SPLASH '10, pages 307–309, New York, NY, USA, 2010. ACM.

[70] J.-R. Falleri, M. Huchard, M. Lafourcade, and C. Nebut. Metamodel Matching for Automatic Model Transformation Generation. In K. Czarnecki, I. Ober, J.-M. Bruel, A. Uhl, and M. Völter, editors, *MoDELS*, volume 5301 of *LNCS*, pages 326–340, Berlin/Heidelberg, 2008. Springer.

[71] A. P. L. Ferreira and L. Ribeiro. A Graph-based Semantics for Object-oriented Programming Constructs. *Electron. Notes Theor. Comput. Sci.*, 122:89–104, 2005.

[72] R. D. F. Ferreira, J. P. Faria, and A. C. R. Paiva. Test Coverage Analysis of UML Activity Diagrams for Interactive Systems. In *Proceedings of QUATIC 2010*, pages 268–273, Washington, DC (USA), 2010. IEEE Computer Society.

[73] A. Förster. *Pattern-Based Business Process Design and Verification.* PhD thesis, University of Paderborn, 2008.

[74] A. Förster, G. Engels, T. Schattkowsky, and R. V. D. Straeten. Verification of Business Process Quality Constraints Based on Visual Process Patterns. In *TASE*, pages 197–208. IEEE Computer Society, 2007.

[75] G. Friedman, A. Hartman, K. Nagin, and T. Shiran. Projected State Machine Coverage for Software Testing. *SIGSOFT Softw. Eng. Notes*, 27:134–143, 2002.

[76] M. Friske, B.-H. Schlingloff, and S. Weißleder. Composition of Model-based Test Coverage Criteria. In *MBEES*, volume 2008-2 of *Informatik-Bericht*, pages 87–94. TU Braunschweig, Institut für Software Systems Engineering, 2008.

[77] P. Gagnon, F. Mokhati, and M. Badri. Applying Model Checking to Concurrent UML Models. *Journal of Object Technology*, 7(1):59–84, 2008.

[78] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns – Elements of Reusable Object-oriented Software*. Addison-Wesley, 1995.

[79] L. Geiger and A. Zündorf. eDOBS – Graphical Debugging for Eclipse. In *Proceedings of GraBaTs '06*, volume 1 of *Electronic Communications of the EASST*. European Association of Software Science and Technology, 2006.

[80] M. Gemis, J. Paredaens, I. Thyssens, and J. van den Bussche. GOOD: A Graph-Oriented Object Database System. In P. Buneman and S. Jajodia, editors, *Proceedings of the 1993 ACM SIGMOD International Conference on the Management of Data*, volume 22, pages 505–510, New York, NY, USA, 6 1993. ACM.

[81] A. Gerber and K. Raymond. MOF to EMF: There and Back Again. In *Proceedings of the 2003 OOPSLA workshop on Eclipse technology eXchange*, eclipse '03, pages 60–64, New York, NY, USA, 2003. ACM.

[82] A. H. Ghamarian, M. de Mol, A. Rensink, E. Zambon, and M. Zimakova. Modelling and analysis using GROOVE. *STTT*, 14(1):15–40, 2012.

[83] H. Giese and R. Wagner. Incremental Model Synchronization with Triple Graph Grammars. In Nierstrasz et al. [151], pages 543–557.

[84] S. Gilmore and J. Hillston. The PEPA Workbench: A Tool to Support a Process Algebra-based Approach to Performance Modelling. In *Proceedings of the Seventh International Conference on Modelling Techniques and Tools for Computer Performance Evaluation*, volume 794 of *LNCS*, pages 353–368, Berlin/Heidelberg, 1994. Springer.

[85] R. Gold. Control Flow Graphs and Code Coverage. *Int. J. Appl. Math. Comput. Sci.*, 20(4):739–749, Dec. 2010.

[86] T. Goldschmidt and G. Wachsmuth. Refinement Transformation Support for QVT Relational Transformations. In *Proceedings of the 3rd Workshop on Model Driven Software Engineering (MDSE 2008)*, 2008.

[87] V. Grassi, R. Mirandola, E. Randazzo, and A. Sabetta. KLAPER: An Intermediate Language for Model-Driven Predictive Analysis of Performance and Reliability. In A. Rausch, R. Reussner, R. Mirandola, and F. Plsil, editors, *The Common Component Modeling Example*, volume 5153 of *LNCS*, pages 327–356. Springer, Berlin/Heidelberg, 2008.

[88] J. Greenyer and E. Kindler. Reconciling TGGs with QVT. In G. Engels, B. Opdyke, D. C. Schmidt, and F. Weil, editors, *MoDELS*, volume 4735 of *LNCS*, pages 16–30. Springer, 2007.

[89] J. Greenyer and E. Kindler. Comparing relational model transformation technologies: implementing Query/View/Transformation with Triple Graph Grammars. *Software and System Modeling*, 9(1):21–46, 2010.

[90] R. C. Gronback. *Eclipse Modeling Project: A Domain-Specific Language (DSL) Toolkit*. Addison-Wesley, 2009.

[91] M. Gyssens, J. Paredaens, J. Van den Bussche, and D. Van Gucht. GOOD: A Graph-Oriented Object Database Model. *IEEE Trans. on Knowl. and Data Eng.*, 6(4):572–586, Aug. 1994.

[92] A. Haase, M. Völter, S. Efftinge, and B. Kolb. Introduction to openArchitectureWare 4.1.2. MDD Tool Implementers Forum (Part of the TOOLS 2007 conference, Zürich), 2007.

[93] R. Hardin, Z. Har'El, and R. Kurshan. COSPAN. In R. Alur and T. Henzinger, editors, *Computer Aided Verification*, volume 1102 of *LNCS*, pages 423–427. Springer, Berlin/Heidelberg, 1996.

[94] S. Haschemi and S. Weißleder. A Generic Approach to Run Mutation Analysis. In *Proceedings of TAIC PART 2010*, volume 6303 of *LNCS*, pages 155–164, Berlin/Heidelberg, 2010. Springer.

[95] J. H. Hausmann. Metamodeling Relations - Relating Metamodels. In *Proceedings of the Metamodelling for MDA workshop, York (UK)*, pages 147–161. University of York, November 2003.

[96] J. H. Hausmann. *Dynamic Meta Modeling*. PhD thesis, University of Paderborn, 2006.

[97] R. Heckel. Compositional Verification of Reactive Systems Specified by Graph Transformation. In E. Astesiano, editor, *Fundamental Approaches to Software Engineering*, volume 1382 of *LNCS*, pages 138–153. Springer, Berlin/Heidelberg, 1998.

[98] R. Heckel, J. M. Küster, and G. Taentzer. Confluence of Typed Attributed Graph Transformation Systems. In A. Corradini, H. Ehrig, H.-J. Kreowski, and G. Rozenberg, editors, *ICGT*, volume 2505 of *LNCS*, pages 161–176, Berlin/Heidelberg, 2002. Springer.

[99] F. Hermann, H. Ehrig, F. Orejas, K. Czarnecki, Z. Diskin, and Y. Xiong. Correctness of Model Synchronization Based on Triple Graph Grammars. In J. Whittle, T. Clark, and T. Kühne, editors, *MoDELS*, volume 6981 of *LNCS*, pages 668–682. Springer, 2011.

[100] J. Hillston. *A Compositional Approach to Performance Modelling*. PhD thesis, University of Edinburgh, 1994.

[101] J. Hillston. Process Algebras for Quantitative Analysis. In *Proceedings of the 20th Annual IEEE Symposium on Logic in Computer Science*, LICS '05, pages 239–248, Washington, DC, USA, 2005. IEEE Computer Society.

[102] C. A. R. Hoare. Communicating Sequential Processes. *Commun. ACM*, 21(8):666–677, Aug. 1978.

[103] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice Hall International, 1985.

[104] G. Holzmann. The Model Checker SPIN. *Software Engineering, IEEE Transactions on*, 23(5):279–295, May.

[105] M. Hornkamp. A Formal, Graph-Based Semantics for UML Activities. Master's thesis, University of Paderborn, 2009.

[106] M. Hülsbusch, B. König, A. Rensink, M. Semenyak, C. Soltenborn, and H. Wehrheim. Full Semantics Preservation in Model Transformation – A Comparison of Proof Techniques. In S. M. D. Méry, editor, *Proceedings of the 8th International Conference on Integrated Formal Methods (IFM 2010)*, LNCS, pages 183–198, Berlin/Heidelberg, 2010. Springer.

[107] M. Hülsbusch, B. König, A. Rensink, M. Semenyak, C. Soltenborn, and H. Wehrheim. Full Semantics Preservation in Model Transformation – A Comparison of Proof Techniques. Technical report, Centre for Telematics and Information Technology of the University of Twente, 2010.

[108] International Software Testing Qualifications Board. ISTQB Glossary of Testing Terms V2.2. http://www.istqb.org/downloads/viewcategory/20.html, 2012. Online, accessed 12–6–2012.

[109] W. Janssen, R. Mateescu, S. Mauw, P. Fennema, and P. v. d. Stappen. Model Checking for Managers. In *Proceedings of the 5th and 6th International SPIN Workshops on Theoretical and Practical Aspects of SPIN Model Checking*, pages 92–107, London, UK, UK, 1999. Springer.

[110] F. Jouault, F. Allilaire, J. Bézivin, I. Kurtev, and P. Valduriez. ATL: A QVT-like Transformation Language. In *OOPSLA '06: Companion to the 21st ACM SIGPLAN symposium on Object-oriented programming systems, languages, and applications*, pages 719–720, New York, NY, USA, 2006. ACM.

[111] T. Jussila, J. Dubrovin, T. Junttila, T. L. Latvala, and I. Porres. Model Checking Dynamic and Hierarchical UML State Machines. In *Proceedings of the 3rd Workshop on Model Design and Validation (MoDeVa 2006)*, 2006.

[112] G. Kappel, P. Langer, W. Retschitzegger, W. Schwinger, and M. Wimmer. Model Transformation By-Example: A Survey of the First Wave. In A. Düsterhöft, M. Klettke, and K.-D. Schewe, editors, *Conceptual Modelling and Its Theoretical Foundations*, volume 7260 of *LNCS*, pages 197–215. Springer, 2012.

[113] H. Kastenberg. *Graph-Based Software Specification and Verification*. PhD thesis, University of Twente, 2008.

[114] H. Kastenberg and A. Rensink. Model Checking Dynamic States in GROOVE. In A. Valmari, editor, *Model Checking Software*, volume 3925 of *LNCS*, pages 299–305. Springer, Berlin/Heidelberg, 2006.

[115] Kent Beck. JUnit Homepage. http://junit.org/. Online, accessed 11–19–2012.

[116] L. Khaluf, C. Gerth, and G. Engels. Pattern-Based Modeling and Formalizing of Business Process Quality Constraints. In H. Mouratidis and C. Rolland, editors, *Proceedings of the 23rd International Conference on Advanced Information System Engineering (CAiSE'11)*, volume 6741 of *LNCS*, pages 521–535, Berlin/Heidelberg, 2011. Springer.

[117] P. King and R. Pooley. Derivation of Petri Net Performance Models from UML Specifications of Communications Software. In B. Haverkort, H. Bohnenkamp, and C. Smith, editors, *Computer Performance Evaluation.Modelling Techniques and Tools*, volume 1786 of *LNCS*, pages 262–276. Springer, Berlin/Heidelberg, 2000.

[118] A. G. Kleppe, J. Warmer, and W. Bast. *MDA Explained: The Model Driven Architecture: Practice and Promise.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2003.

[119] A. Knapp and J. Wuttke. Model checking of UML 2.0 interactions. In *Proceedings of the 2006 international conference on Models in software engineering*, MoDELS'06, pages 42–51, Berlin/Heidelberg, 2006. Springer.

[120] D. Kolovos, R. Paige, and F. Polack. The Epsilon Transformation Language. In A. Vallecillo, J. Gray, and A. Pierantonio, editors, *Proceedings of the First International Conference on Theory and Practice of Model Transformations (ICMT 2008)*, volume 5063 of *LNCS*, pages 46–60, Berlin/Heidelberg, 2008. Springer.

[121] B. König and V. Kozioura. Augur 2 - A New Version of a Tool for the Analysis of Graph Transformation Systems. *Electr. Notes Theor. Comput. Sci.*, 211:201–210, 2008.

[122] A. Königs. *Model Integration and Transformation: A Triple Graph Grammar-based QVT Implementation.* PhD thesis, TU Darmstadt, 2009.

[123] H.-J. Kreowski and S. Kuske. Graph Transformation Units with Interleaving Semantics. *Formal Aspects of Computing*, 11(6):690–723, 1999.

[124] G. Kutty, L. Dillon, L. Moser, P. Melliar-Smith, and Y. S. Ramakrishna. Visual Tools for Temporal Reasoning. In *Visual Languages, 1993., Proceedings 1993 IEEE Symposium on*, pages 152–159, 1993.

[125] P. Langer, M. Wimmer, and G. Kappel. Model-to-Model Transformations By Demonstration. In L. Tratt and M. Gogolla, editors, *ICMT*, volume 6142 of *LNCS*, pages 153–167. Springer, 2010.

[126] J. Lara, E. Guerra, and H. Vangheluwe. Meta-Modelling, Graph Transformation and Model Checking for the Analysis of Hybrid Systems. In J. Pfaltz, M. Nagl, and B. Böhlen, editors, *Applications of Graph Transformations with Industrial Relevance*, volume 3062 of *LNCS*, pages 292–298. Springer, Berlin/Heidelberg, 2004.

[127] M. Lauder, A. Anjorin, G. Varró, and A. Schürr. Bidirectional Model Transformation with Precedence Triple Graph Grammars. In A. Vallecillo, J.-P. Tolvanen, E. Kindler, H. Störrle, and D. S. Kolovos, editors, *ECMFA*, volume 7349 of *LNCS*, pages 287–302. Springer, 2012.

[128] M. Lawley and J. Steel. Practical Declarative Model Transformation with Tefkat. In J.-M. Bruel, editor, *MoDELS Satellite Events*, volume 3844 of *LNCS*, pages 139–150. Springer, 2005.

[129] E. Legros, C. Amelunxen, F. Klar, and A. Schürr. Generic and Reflective Graph Transformations for Checking and Enforcement of Modeling Guidelines. *J. Vis. Lang. Comput.*, 20(4):252–268, 2009.

[130] C. Lindemann, A. Thümmler, A. Klemm, M. Lohmann, and O. P. Waldhorst. Performance analysis of time-enhanced UML diagrams based on stochastic processes. In *Proceedings of the 3rd international workshop on Software and performance*, WOSP '02, pages 25–34, New York, NY, USA, 2002. ACM.

[131] LMU München. HUGO/RT. http://www.pst.informatik.uni-muenchen.de/projekte/hugo/. Online, accessed 4–3–2013.

[132] A. P. Lüdtke Ferreira and L. Ribeiro. Derivations in Object-Oriented Graph Grammars. In H. Ehrig, G. Engels, F. Parisi-Presicce, and G. Rozenberg, editors, *Proceedings of the 2nd International Conference on Graph Transformations (ICGT 2004)*, volume 3256 of *LNCS*, pages 416–430, Berlin/Heidelberg, 2004. Springer.

[133] P. L. M. Clavel, S. Eker and J. Meseguer. Principles of Maude. In J. Meseguer, editor, *Electronic Notes in Theoretical Computer Science*, volume 4. Elsevier Science Publishers, 2000.

[134] Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems*. Springer, New York, NY, USA, 1992.

[135] Matthias Biel. Literature Study on Model Transformations. http://staffwww.dcs.shef.ac.uk/people/A.Simons/remodel/papers/BiehlModelTransformations.pdf, 2010. Online, accessed 5–8–2013.

[136] J. A. McQuillan and J. F. Power. White-Box Coverage Criteria for Model Transformations. In *Proceedings of the 1st International Workshop on Model Transformation with ATL*, Aachen (Germany), 2009. CEUR Workshop Proceedings.

[137] S. J. Mellor and M. J. Balcer. *Executable UML: A Foundation for Model-Driven Architecture*. Addison-Wesley Professional, 2002.

[138] T. Mens, G. Taentzer, and O. Runge. Detecting Structural Refactoring Conflicts Using Critical Pair Analysis. *Electr. Notes Theor. Comput. Sci.*, 127(3):113–128, 2005.

[139] T. Mens and P. Van Gorp. A Taxonomy of Model Transformation. *Electron. Notes Theor. Comput. Sci.*, 152:125–142, Mar. 2006.

[140] Mentor Graphics. BridgePoint Comprehensive xtUML Tool Suite. http://www.mentor.com/products/sm/model_development/bridgepoint/. Online, accessed 5–14–2013.

[141] R. Milner. *A Calculus of Communicating Systems*. Springer, Berlin/Heidelberg, 1980.

[142] R. Milner. *Communicating and Mobile Systems: The π-calculus*. Cambridge University Press, 1999.

[143] R. Milner, J. Parrow, and D. Walker. A Calculus of Mobile Processes, I. *Information and Computation*, 100(1):1–40, 1992.

[144] M. Minas and G. Viehstaedt. DiaGen: A Generator for Diagram Editors Providing Direct Manipulation and Execution of Diagrams. In *Proceedings of VL '95*. IEEE Computer Society, 1995.

[145] M. Mohamed, M. Romdhani, and K. Ghedira. MOF-EMF Alignment. In *Autonomic and Autonomous Systems, 2007. ICAS07. Third International Conference on*, page 1, June.

[146] G. E. Moore. Cramming More Components onto Integrated Circuits. *Electronics*, 38(8):114–117, Apr. 1965.

[147] G. J. Myers, C. Sandler, and T. Badgett. *The Art of Software Testing*. John Wiley & Sons, 2011.

[148] M. Naftalin and P. Wadler. *Java Generics and Collections*. O'Reilly Media, 2006.

[149] V. Nesterow. Eine formale, graphbasierte Semantik für UML State Machines. Master's thesis, University of Paderborn, 2009.

[150] U. Nickel, J. Niere, and A. Zündorf. The FUJABA environment. In C. Ghezzi, M. Jazayeri, and A. L. Wolf, editors, *ICSE*, pages 742–745. ACM, 2000.

[151] O. Nierstrasz, J. Whittle, D. Harel, and G. Reggio, editors. *Model Driven Engineering Languages and Systems, 9th International Conference, MoDELS 2006, Genova, Italy, October 1-6, 2006, Proceedings*, volume 4199 of *LNCS*. Springer, 2006.

[152] A. Niewiadomski, W. Penczek, and M. Szreter. A New Approach to Model Checking of UML State Machines. *Fundam. Inf.*, 93(1-3):289–303, Jan. 2009.

[153] Object Management Group. UML Profile for Schedulability, Performance, and Time Specification. http://www.omg.org/spec/SPTP/1.1/PDF/, 2005. Online, accessed 4–3–2013.

[154] 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.

[155] Object Management Group. Object Constraint Language V2.0. http://www.omg.org/docs/formal/06-05-01.pdf, 5 2006.

[156] Object Management Group. Meta Object Facility (MOF) 2.0 Query/View/Transformation Specification, Version 1.0. http://www.omg.org/spec/QVT/1.0/PDF/08-04-03.pdf, 2008.

[157] Object Management Group. UML Infrastructure, Version 2.3. http://www.omg.org/spec/UML/2.3/, 2010. Online, accessed 3–11–2013.

[158] Object Management Group. UML Superstructure, Version 2.3. http://www.omg.org/spec/UML/2.3/, 2010. Online, accessed 3–11–2013.

[159] Object Management Group. UML Profile for MARTE: Modeling and Analysis of Real-Time Embedded Systems. http://www.omg.org/spec/MARTE/1.1/PDF/, 2011. Online, accessed 4–3–2013.

[160] G. O'Keefe. *The Meaning of UML Models*. PhD thesis, Australian National University, 2008.

[161] Pathfinder Solutions. PathMATE. http://www.pathfindersolns.com/products/pathmate/. Online, accessed 5–14–2013.

[162] C. A. Petri. *Kommunikation mit Automaten*. PhD thesis, University of Bonn, 1962.

[163] G. D. Plotkin. A Structural Approach to Operational Semantics. *J. Log. Algebr. Program.*, 60–61:17–139, 2004.

[164] D. Plump. Termination of Graph Rewriting is Undecidable. *Fundam. Inf.*, 33(2):201–209, Feb. 1998.

[165] A. Pnueli. The Temporal Logic of Programs. In *Foundations of Computer Science, 1977., 18th Annual Symposium on*, pages 46–57, 31 1977-nov. 2 1977.

[166] A. Rensink. The GROOVE Simulator: A Tool for State Space Generation. In J. L. Pfaltz, M. Nagl, and B. Böhlen, editors, *AGTIVE 2003 – Revised Selected and Invited Papers*, volume 3062 of *LNCS*, pages 479–485, Berlin/Heidelberg, 2004. Springer.

[167] A. Rensink. Time and Space Issues in the Generation of Graph Transition Systems. *Electr. Notes Theor. Comput. Sci.*, 127(1):127–139, 2005.

[168] A. Rensink. Nested Quantification in Graph Transformation Rules. In Corradini et al. [30], pages 1–13.

[169] A. Rensink and D. Distefano. Abstract Graph Transformation. *Electr. Notes Theor. Comput. Sci.*, 157(1):39–59, 2006.

[170] A. Rensink and J.-H. Kuperus. Repotting the Geraniums: On Nested Graph Transformation Rules. In *Proceedings of GT/VMT 2009*, volume 18 of *Electronic Communications of the EASST*. European Association of Software Science and Technology, 2009.

[171] A. Rensink, A. Schmidt, and D. Varró. Model Checking Graph Transformations: A Comparison of Two Approaches. In H. Ehrig, G. Engels, F. Parisi-Presicce, and G. Rozenberg, editors, *Graph Transformations*, volume 3256 of *LNCS*, pages 226–241. Springer, Berlin/Heidelberg, 2004.

[172] A. Rensink and E. Zambon. Neighbourhood Abstraction in GROOVE. In *Graph-based Tools 2010*, volume 32 of *Electronic Communications of the EASST*. European Association of Software Science and Technology, 2010.

[173] T. Rheker. A Bidrectional Transformation between EMF Models and Typed Graphs. Bachelor's thesis, University of Paderborn, 2008.

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

[175] Robby, M. B. Dwyer, and J. Hatcliff. Bogor: an Extensible and Highly-modular Software Model Checking Framework. *SIGSOFT Softw. Eng. Notes*, 28(5):267–276, Sept. 2003.

[176] S. Roser and B. Bauer. An Approach to Automatically Generated Model Transformations Using Ontology Engineering Space. In *Proceedings of Workshop on Semantic Web Enabled Software Engineering (SWESE*, 2006.

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

[178] D. A. Sadilek. Prototyping Domain-Specific Language Semantics. In *Companion to the 23rd ACM SIGPLAN conference on Object-oriented Programming Systems Languages and Applications*, New York, 2008. ACM.

[179] D. A. Sadilek and G. Wachsmuth. Prototyping Visual Interpreters and Debuggers for Domain-Specific Modelling Languages. In I. Schieferdecker and A. Hartman, editors, *Proceedings of ECMDA '08*, volume 5095 of *LNCS*, pages 63–78, Berlin/Heidelberg, 2008. Springer.

[180] J. Schäfer. Eine formale, graphbasierte Semantik für UML Interaktionen. Master's thesis, University of Paderborn, 2009.

[181] T. Schäfer, A. Knapp, and S. Merz. Model checking UML state machines and collaborations. *Electronic Notes in Theoretical Computer Science*, 55(3):13 pages, 2001.

[182] A. Schmidt and D. Varró. CheckVML: A Tool for Model Checking Visual Modeling Languages. In P. Stevens, J. Whittle, and G. Booch, editors, *UML 2003 - The Unified Modeling Language. Modeling Languages and Applications*, volume 2863 of *LNCS*, pages 92–95. Springer, Berlin/Heidelberg, 2003.

[183] H. Schreiber. Metamodellbasierte, teilautomatisierte Transformation von visuellen Modellen in ausführbare DMM-Laufzeitmodelle. Master's thesis, University of Paderborn, 2010.

[184] A. Schürr. Specification of Graph Translators with Triple Graph Grammars. In E. W. Mayr, G. Schmidt, and G. Tinhofer, editors, *WG*, volume 903 of *LNCS*, pages 151–163. Springer, 1994.

[185] A. Schürr and F. Klar. 15 Years of Triple Graph Grammars. In Ehrig et al. [58], pages 411–425.

[186] M. Semenyak. *Full Semantics Preservation in Model Transformation.* PhD thesis, University of Paderborn, 2011.

[187] R. F. Serfozo. An Equivalence between Continuous and Discrete Time Markov Decision Processes. *Operations Research*, 27(3):616–620, 1979.

[188] K. Smolander, K. Lyytinen, V.-P. Tahvanainen, and P. Marttiin. MetaEdit: a Flexible Graphical Environment for Methodology Modelling. In *Proceedings of the third international conference on Advanced information systems engineering (CAiSE 91)*, pages 168–193, New York, NY, USA, 1991. Springer New York, Inc.

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

[190] C. Soltenborn and G. Engels. Towards Generalizing Visual Process Pattern. In P. Bottoni, E. Guerra, J. de Lara, T. Margaria, J. Padberg, and G. Taentzer, editors, *Proceedings of the 1st International Workshop on Visual Formalisms for Patterns (VFfP 2009), Corvallis, OR (USA)*, volume 25 of *Electronic Communications of the EASST*. European Association of Software Science and Technology, 2009.

[191] C. Soltenborn and G. Engels. Towards Test-Driven Semantics Specification. In B. S. A. Schürr, editor, *Proceedings of the 12th International Conference on Model Driven Engineering Languages and Systems (MoDELS 2009), Denver, Colorado (USA)*, pages 378–392, Berlin/Heidelberg, 2009. Springer.

[192] C. Soltenborn and G. Engels. Using Rule Overriding to Improve Reusability and Understandability of Dynamic Meta Modeling Specifications. *Journal of Visual Languages and Computing*, 22(3):233–250, 2011.

[193] Sparx Systems. Enterprise Architect. `http://www.sparxsystems.com.au/`. Online, accessed 5–14–2013.

[194] D. Steinberg, F. Budinsky, M. Paternostro, and E. Merks. *EMF – Eclipse Modeling Framework*. Addison-Wesley, 2008.

[195] P. Stevens. A Landscape of Bidirectional Model Transformations. In R. Lämmel, J. Visser, and J. Saraiva, editors, *GTTSE*, volume 5235 of *LNCS*, pages 408–424. Springer, 2007.

[196] H. Störrle and J. H. Hausmann. Towards a Formal Semantics of UML 2.0 Activities. In P. Liggesmeyer, K. Pohl, and M. Goedicke, editors, *Software Engineering*, volume 64 of *LNI*, pages 117–128. GI, 2005.

[197] M. Strommer and M. Wimmer. A Framework for Model Transformation By-Example: Concepts and Tool Support. In R. F. Paige and B. Meyer, editors, *TOOLS (46)*, volume 11 of *Lecture Notes in Business Information Processing*, pages 372–391. Springer, 2008.

[198] G. Taentzer. *Parallel and distributed graph transformation - formal description and application to communication-based systems*. PhD thesis, TU Berlin, 1996.

[199] G. Taentzer. AGG: A Tool Environment for Algebraic Graph Transformation. In M. Nagl, A. Schürr, and M. Münch, editors, *AGTIVE*, volume 1779 of *LNCS*, pages 481–488, Berlin/Heidelberg, 1999. Springer.

[200] G. Taentzer. AGG: A Graph Transformation Environment for Modeling and Validation of Software. In J. L. Pfaltz, M. Nagl, and B. Böhlen, editors, *AGTIVE*, volume 3062 of *LNCS*, pages 446–453, Berlin/Heidelberg, 2003. Springer.

[201] M. Utting and B. Legeard. *Practical Model-Based Testing: A Tools Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA (USA), 2006.

[202] M. Utting and B. Legeard. *Practical Model-Based Testing – A Tools Approach*. Elsevier, Amsterdam, The Netherlands, 2007.

[203] W. van der Aalst. Verification of Workflow Nets. In *Proceedings of the 18th International Conference on Application and Theory of Petri Nets (ICATPN 97)*, pages 407–426, London, UK, 1997. Springer.

[204] W. Van Der Aalst and K. van Hee. *Workflow Management: Models, Methods, and Systems*. Cooperative Information Systems Series. MIT Press, 2004.

[205] M. Vardi. Branching vs. Linear Time: Final Showdown. In *Proceedings of the 2001 Conference on Tools and Algorithms for the Construction and Analysis of Systems, TACAS 2001*, volume 2031 of *LNCS*, pages 1–22, Berlin/Heidelberg, 2001. Springer.

[206] D. Varró. Model Transformation by Example. In Nierstrasz et al. [151], pages 410–424.

[207] D. Varró, S. Varró-Gyapay, H. Ehrig, U. Prange, and G. Taentzer. Termination Analysis of Model Transformations by Petri Nets. In Corradini et al. [30], pages 260–274.

[208] M. von Detten. Archimetrix: A Tool for Deficiency-Aware Software Architecture Reconstruction. In *WCRE*, pages 503–504. IEEE Computer Society, 2012.

[209] J. Wang, S.-K. Kim, and D. A. Carrington. Verifying Metamodel Coverage of Model Transformations. In *ASWEC*, pages 270–282. IEEE Computer Society, 2006.

[210] S. Weißleder and B.-H. Schlingloff. Quality of Automatically Generated Test Cases based on OCL Expressions. In *ICST*, pages 517–520. IEEE Computer Society, 2008.

[211] L. Wendehals. *Struktur- und verhaltensbasierte Entwurfsmustererkennung*. PhD thesis, University of Paderborn, 2007.

[212] M. Woodside. From Annotated Software Designs (UML SPT/MARTE) to Model Formalisms. In M. Bernardo and J. Hillston, editors, *Formal Methods for Performance Evaluation*, volume 4486 of *LNCS*, pages 429–467. Springer, Berlin/Heidelberg, 2007.

[213] F. Xie, V. Levin, and J. C. Browne. Model Checking for an Executable Subset of UML. In *Proceedings of the 16th IEEE international conference on Automated software engineering*, ASE '01, pages 333–, Washington, DC, USA, 2001. IEEE Computer Society.

[214] L. Xuandong, W. Linzhang, Q. Xiaokang, L. Bin, Y. Jiesong, Z. Jianhua, and Z. Guoliang. Runtime Verification of Java Programs for Scenario-Based Specifications. In L. M. Pinho and M. G. Harbour, editors, *Proceedings of the 11th Ada-Europe International Conference on Reliable Software Technologies (Ada-Europe 06)*, volume 2006 of *LNCS*, pages 94–105, Berlin/Heidelberg, 2006. Springer.

[215] E. Zambon and A. Rensink. Using Graph Transformations and Graph Abstractions for Software Verification. In *Proceedings of ICGT 2010 – Doctoral Symposium*, volume 38 of *Electronic Communications of the EASST*. European Association of Software Science and Technology, 2011.

A Appendix

A.1 Custom OCL Operations

Some OCL constraints describing the static semantics of the DMM language are rather difficult or even impossible to express using standard OCL. Fortunately, the OCL framework we are using (see Sect. 2.3.2) provides an easy way to extend an OCL evaluation environment with custom OCL operations, which can then be used within the OCL expressions defining the DMM language's static semantics as presented in appendix A.2. In this section, we will define these custom OCL operations.

For each operation, we provide a precise description of the operation's semantics – the formal definition of these operations by means of Java code can be investigated in class `de.upb.dmm.ruleset.ocl.DMMEvaluationEnvironment` of plug-in `de.upb.dmm.ruleset.ocl`.

A.1.1 `EObject.findParentObject(EClass): EObject`

This operation is used to find the nearest object in the containment hierarchy which a) transitively contains this `EObject` and b) has the same type as the `EClass` provided as the operation's parameter. For instance, calling `findParentObject()` on an invocation and passing `ruleset::Package` as a parameter will return the package which directly contains the invocation's rule.

A.1.2 `Invocation.getCompatibleRules(): Set`

This operation computes the set of DMM rules which are *compatible* with the given invocation. A rule is compatible with an invocation if all of the following conditions hold:

- The rule's name is the same as the invocation's `invokedRule` attribute value
- The number of parameters of the rule is the same as the number of parameters passed by the invocation
- The rule's context node's type is the same or a subtype of the invocation's target node's type

- The type of the i-th parameter passed by the invocation must be the same or a subtype of the type of the i-th parameter of the rule

A.1.3 `EDataType.isCompatible(EDataType): boolean`

This operation returns `true` if and only if a value of the passed datatype can be assigned to an attribute having this datatype. For instance, calling `isCompatible` on datatype `ecore::ELong` and passing datatype `ecore::EInt` will yield `true`; switching the two datatypes of this example will yield `false`.

A.1.4 `Expression.getEDataType(): EDataType`

This operation returns the `EDataType` of the expression's evaluation result. For instance, calling `getEDataType` on the expression represented by the string "4711 == 2001" will yield `ecore::EBoolean`.

A.1.5 `Rule.hasAssignmentCycles(): boolean`

This operation returns `true` if and only if the assignments in the given rule do contain *assignment cycles*. An assignment cycle is a number of assignments such that one assignment depends on the result of another assignment and vice versa. For instance, a very simple assignment cycle is contained in the two assignments `a' := b'` and `b' := a'`. The operation is implemented by computing a topological order on the rule's assignments – it returns `true` if and only if this is not possible.

A.1.6 `Rule.getUqsClusters(): Set`

A *UQS cluster* is a set of nodes which are connected to each other, and where each of the nodes has either quantification `Quantifier::ZERO_TO_MANY` or `Quantifier::ONE_TO_MANY`. This methods returns the set of such clusters (if any), i.e., it returns a set of sets of nodes.

A.1.7 `Rule.getUqsCluster(Node): Set`

This operation returns the UQS cluster containing the passed node (if any), i.e., a set of nodes. For the definition of a UQS cluster see Sect. A.1.6.

A.1.8 `Rule.getNotExistsCluster(): Set`

A *not exists cluster* is a set of nodes which are connected to each other, and where each of the nodes has role `ElementRole::NOT_EXISTS`. This methods returns the not exists clusters to which the passed node belongs (if any), i.e., it returns a set of nodes.

A.1.9 `Rule.areNestedNodesConnectedProperly(): boolean`

Each not exists cluster must be connnected to exactly one UQS cluster. The methods returns `true` if and only if this is the case for this rule. For the definition of a UQS cluster see Sect. A.1.6, for the not exists cluster see Sect. A.1.8.

A.2 DMM: Static Semantics

This section defines the static semantics of the DMM language by means of OCL constraints for the DMM metamodel's metaclasses. The OCL expressions make use of the custom OCL operations as defined in the last section. The metaclasses are organized as in the DMM language definition we have seen in Sect. 6.2 on page 59. Metaclasses which do not introduce any OCL constraints are not listed within this section at all.

A.2.1 Ruleset Structure

A.2.1.1 Ruleset

Constraints

- A ruleset must have a non-empty name
  `self.name.size() > 0`

A.2.1.2 Package

Constraints

- A package is either contained in a ruleset or in a package
  `self.ruleset.oclIsUndefined() xor self.parent.oclIsUndefined()`

A.2.2 Rule Hierarchy

A.2.2.1 Rule

Constraints

- A rule must have a non-empty name
  `self.name.size() > 0`

- A rule must have a context node
  `not self.contextNode.oclIsUndefined()`

- A rule's context node must have a non-empty name
  `self.contextNode.name.size() > 0`

- A rule's context node must have role `EXISTS` or `DESTROY`
  `self.contextNode.role = ElementRole::EXISTS or self.contextNode.role = ElementRole::DESTROY`

- A rule's unique name must be distinct within a ruleset
```
self.ruleset.rule->forAll(
  r, s | r <> s implies r.uniqueName <> s.uniqueName)
```

A.2.2.2 ParameterizedElement

Constraints

- Every node acting as a parameter within a rule must have a name
```
self.parameter->forAll(n | n.name.size() > 0)
```

- All nodes used as parameters must have pairwise distinct names
```
self.parameter->forAll(
  n,m | n <> m implies n.name <> m.name)
```

A.2.2.3 PremiseRule

Constraints

- A premise rule must not contain nodes which are to be deleted or created
```
self.node->forAll(n |
   n.role = ElementRole::EXISTS or
   n.role = ElementRole::NOT_EXISTS)
```

- A premise rule must not contain edges which are to be deleted or created
```
self.edge->forAll(e |
   e.role = ElementRole::EXISTS or
   e.role = ElementRole::NOT_EXISTS)
```

- A premise rule can only invoke other premise rules
```
self.invocation->forAll(i |
   i.oclIsTypeOf(PremiseRule))
```

A.2.2.4 PropertyRule

Constraints

- A property rule must not contain nodes which are to be deleted or created
```
self.node->forAll(n |
   n.role = ElementRole::EXISTS or
   n.role = ElementRole::NOT_EXISTS)
```

- A property rule must not contain edges which are to be deleted or created
```
self.edge->forAll(e |
   e.role = ElementRole::EXISTS or
   e.role = ElementRole::NOT_EXISTS)
```

A.2.2.5 OverridingRelation

Constraints

- A rule can only override another rule if the overridden rule's context node's type is a supertype of the overriding rule's context node's type
```
self.overridingRule.contextnode.type <>
self.overriddenRule.contextnode.type and
self.overridingRule.contextnode.type.allSupertypes
  ->includes(self.overriddenRule.contextnode.type)
```

- A rule participating in a CompleteOveridingRelation cannot also participate in a SoftOverrdingRelation
```
SoftOverridingRelation.allInstances()->forAll(r |
   r.overriddenRule <> self.overriddenRule and
   r.overridingRule <> self.overriddenRule and
   r.overriddenRule <> self.overridingRule and
   r.overridingRule <> self.overridingRule)
```

- A rule participating in a SoftOveridingRelation cannot also participate in a CompleteOverridingRelation
```
CompleteOverridingRelation.allInstances()->forAll(r |
   r.overriddenRule <> self.overriddenRule and
   r.overridingRule <> self.overriddenRule and
   r.overriddenRule <> self.overridingRule and
   r.overridingRule <> self.overridingRule)
```

A.2.2.6 SoftOverridingRelation

Constraints

- Only smallstep rules can participate in an overriding relation (see semantics below)
```
self.overridingRule.oclIsType(SmallstepRule) and
self.overriddenRule.oclIsType(SmallstepRule)
```

A.2.3 Internal Rule Structure

A.2.3.1 Node

Constraints

- Nodes acting as parameters of an invocation must not be universally quantified
```
self.quantification <> ruleset::Quantifier::ONE
implies self.rule.invocation->forAll(i |
  not i.parameter->includes(self))
```

- Universally quantified nodes must not have role `ElementRole::CREATE` or `ElementRole::NOT_EXISTS`
```
(self.quantification = Quantifier::ONE_TO_MANY or
self.quantification = Quantifier::ZERO_TO_MANY)
implies not
(self.role = ElementRole::CREATE or
self.role = ElementRole::NOT_EXISTS)
```

- A node must have a type from one of the ruleset's metamodels
```
self.rule.ruleset.metamodels->exists(m |
   m.eClassifiers->includes(self.type))
```
- Nodes carrying emphasized attributes must have quantification `Quantifier::ONE` and role `ElementRole::EXISTS` or `ElementRole::DESTROY`
```
self.emphasizedAttributes->size() > 0 implies
self.quantification = Quantifier::ONE and
(self.role = ElementRole::EXISTS or
self.role = ElementRole::DESTROY)
```
- A nested node must be connected to exactly one UQS cluster
```
self.rule.areNestedNodesConnectedProperly()
```
- All nodes of a UQS cluster must have the same quantification
```
self.quantification = Quantifier::ZERO_TO_MANY or
self.quantification = Quantifier::ONE_TO_MANY
implies
self.rule.getUqsClusters()->forAll(c |
   c->exists(n | n = self) implies
   c->forAll(n |
     n.quantification = self.quantification))
```

A.2.3.2 Edge

Constraints

- An edge must have a reference from one of the ruleset's metamodels
```
self.rule.ruleset.metamodels->exists(m |
   m.eClassifiers->exists(c |
     c.oclIsTypeOf(ecore::EClass) and
     c.oclAsType(ecore::EClass).eReferences
      ->includes(self.reference)))
```
- An edge's reference must be compatible with its source and target nodes' types
```
(self.source.type.eAllSuperTypes->includes(
  self.reference.eContainingClass.oclAsType(
    ecore::EClass))
or self.source.type = reference.eContainingClass)
and
(self.target.type.eAllSuperTypes->includes(
  self.reference.eType.oclAsType(
    ecore::EClass))
or self.target.type = reference.eType)
```
- The nodes of an edge must not have roles `ElementRole::DESTROY` and `ElementRole::CREATE`
```
not((
   self.source.role = ruleset::ElementRole::DESTROY
   and
```

```
self.target.role = ruleset::ElementRole::CREATE)
or
(self.source.role = ruleset::ElementRole::CREATE
and
self.target.role = ruleset::ElementRole::DESTROY))
```

- If one of an edge's nodes has role `ElementRole::CREATE`, `ElementRole::DESTROY`, or `ElementRole::NOT_EXISTS`, the edge must have the same role

```
((self.source.role = ruleset::ElementRole::NOT_EXISTS
or
self.target.role = ruleset::ElementRole::NOT_EXISTS)
implies self.role = ruleset::ElementRole::NOT_EXISTS)
and
((self.source.role = ruleset::ElementRole::CREATE or
self.target.role = ruleset::ElementRole::CREATE)
implies self.role = ruleset::ElementRole::CREATE)
and
((self.source.role = ruleset::ElementRole::DESTROY or
self.target.role = ruleset::ElementRole::DESTROY)
implies (self.role = ruleset::ElementRole::DESTROY
or self.role = ruleset::ElementRole::NOT_EXISTS))
```

- If one of an edge's nodes has role `ElementRole::NOT_EXISTS` and the other has role `ElementRole::DESTROY`, the edge must have role `ElementRole::NOT_EXISTS`

```
((self.source.role = ElementRole::NOT_EXISTS and
self.target.role = ElementRole::DESTROY) or
(self.source.role = ElementRole::DESTROY and
self.target.role = ElementRole::NOT_EXISTS))
implies self.role = ElementRole::NOT_EXISTS
```

- If this edge connects a node with role `ElementRole::NOT_EXISTS` to a nested node, the former node must also be nested

```
not(
(self.source.role = ElementRole::NOT_EXISTS and
self.source.quantification = ruleset::Quantifier::ONE
and self.target.quantification = Quantifier::NESTED)
or (self.target.role = ElementRole::NOT_EXISTS and
self.target.quantification = ruleset::Quantifier::ONE
and self.source.quantification = Quantifier::NESTED))
```

A.2.3.3 Invocation

Constraints

- The target node of an invocation must not have role `ElementRole::DESTROY` or `ElementRole::NOT_EXISTS`

```
self.targetnode.role <> ElementRole::DESTROY and
self.targetnode.role <> ElementRole::NOT_EXISTS
```

- The parameters of an invocation must not have role `ElementRole::DESTROY` or `ElementRole::NOT_EXISTS`
```
self.parameter->forAll(p |
    p.role <> ElementRole::DESTROY and
    p.role <> ElementRole::NOT_EXISTS)
```

- Every invocation must invoke an existing rule, i.e., a rule with the same name and compatible contextnode and parameters (see Sect. A.1.2 for more details)
```
self.getCompatibleRules()->size() > 0
```

- Premise rules must not invoke smallstep rules, only other premise rules
```
self.rule.oclIsTypeOf(ruleset::PremiseRule) implies
self.getCompatibleRules()->forAll(r |
    r.oclIsTypeOf(ruleset::PremiseRule))
```

- Property rules must not invoke smallstep rules, only premise rules
```
self.rule.oclIsTypeOf(ruleset::PropertyRule) implies
self.getCompatibleRules()->forAll(r |
    r.oclIsTypeOf(ruleset::PremiseRule))
```

- An invoked premise rule must be unique, i.e., there must only exist one compatible premise rule
```
self.getCompatibleRules()->exists(r |
    r.oclIsTypeOf(ruleset::PremiseRule))
implies self.getCompatibleRules()->size() = 1
```

- The target node of a premise rule invocation must not be quantified `Quantifier::ZERO_TO_MANY` or `Quantifier::ONE_TO_MANY`
```
self.getCompatibleRules()->exists(r |
    r.oclIsTypeOf(ruleset::PremiseRule))
implies self.targetnode.quantification <>
Quantifier::ZERO_TO_MANY and
self.targetnode.quantification <>
Quantifier::ONE_TO_MANY
```

A.2.3.4 Condition

Constraints

- Conditions must evaluate to a boolean value
```
self.expression.getEDataType().isCompatible(
    ecore::EBoolean.oclAsType(ecore::EDataType))
```

- Identifiers within a condition must not refer to the new value of attributes
```
self.expression.oclAsType(ecore::EObject)
  .eAllContents()->forAll(o |
      o.oclIsTypeOf(expression::AttributeExpression)
      implies not
      o.oclAsType(expression::AttributeExpression)
     .nextStateValue)
```

- Only enumeration conditions of form `<attribute>==` '`<enumLiteral>`' are allowed

```
let opExp : expression::OperationExpression =
self.expression.oclAsType(
  expression::OperationExpression)
in opExp.subexpressions->at(1).oclIsTypeOf(
  expression::LiteralExpression)
implies opExp.subexpressions->at(1).oclAsType(
  expression::LiteralExpression).literal.oclIsTypeOf(
    expression::EnumerationLiteral)
implies
(opExp.operator.symbol = '==' or
opExp.operator.symbol = '!=') and
opExp.subexpressions->size() = 2 and
opExp.subexpressions->at(0).oclIsTypeOf(
  expression::AttributeExpression)
```

- Only literals of the according enumeration can be used in a condition

```
let opEx : expression::OperationExpression =
self.expression.oclAsType(
  expression::OperationExpression)
in opEx.subexpressions->at(1).oclIsTypeOf(
  expression::LiteralExpression)
implies
let litExp : expression::LiteralExpression =
opEx.subexpressions->at(1).oclAsType(
  expression::LiteralExpression)
in litExp.literal.oclIsTypeOf(
  expression::EnumerationLiteral)
implies
opEx.subexpressions->at(0).oclAsType(
  expression::AttributeExpression).targetAttribute
  .eType.oclAsType(ecore::EEnum)
  .eLiterals->includes(
    litExp.literal.oclAsType(ecore::EEnumLiteral))
```

A.2.3.5 Assignment

Constraints

- An assignment must not assign a value to an attribute of a node other than the assignment's owner

```
self.assignTo.targetNode = self.oclAsType(
  ecore::EObject).eContainer()
```

- The data type of the expression must fit to the data type of the attribute within an assignment

```
self.assignTo.oclAsType(expression::Expression)
.getEDataType().oclAsType(ecore::EDataType)
.isCompatible(
```

```
self.expression.getEDataType()
.oclAsType(ecore::EDataType))
```

- An assignment must assign its expression to a new value of an attribute
  `self.assignTo.nextStateValue`

A.2.3.6 EmphasizedNodeAttribute

Constraints

- Only attributes belonging to the node's type can be used as emphasized attributes
  ```
  self.node.type.eAllAttributes()->contains(
    self.attribute)
  ```

A.2.4 DMM Expression Language

A.2.4.1 Expression

Constraints

- The data type of the sub expressions does not fit the operator within the expression
  `not self.getEDataType().oclIsUndefined()`

A.2.4.2 AttributeExpression

Constraints

- An identifier must refer to an existing attribute
  ```
  self.targetNode.type.eAllAttributes->includes(
    self.targetAttribute)
  ```

- Attribute expressions on the left side of an assignment must always refer to the attribute's value after rule application
  `self.leftHandSide implies self.nextStateValue`

- An identifier must not refer to a new value of an attribute of a node with the role `ElementRole::DESTROY`
  ```
  self.nextStateValue and not self.leftHandSide
  implies self.targetNode.role <> ElementRole::DESTROY
  ```

- An identifier must not refer to a value of an attribute of a node with the role not exists
  ```
  let rule : Rule = self.oclAsType(ecore::EObject)
  .findParentObject(Rule.oclAsType(ecore::EClass))
  .oclAsType(Rule)
  in let parentNode : Node = self.oclAsType(
    ecore::EObject)
  .findParentObject(Node.oclAsType(ecore::EClass))
  .oclAsType(Node)
  in let targetNode : Node = self.targetNode
  ```

```
in not self.leftHandSide and
targetNode.role = ElementRole::NOT_EXISTS
implies
parentNode.role = ElementRole::NOT_EXISTS and
rule.getNotExistsCluster(targetNode)->includes(
  parentNode)
```

- An identifier must not refer to a value of an attribute of a universally quantified node

```
let rule : Rule = self.oclAsType(ecore::EObject)
.findParentObject(Rule.oclAsType(ecore::EClass))
.oclAsType(Rule)
in let parentNode : Node = self.oclAsType(
  ecore::EObject)
.findParentObject(Node.oclAsType(ecore::EClass))
.oclAsType(Node)
in let targetNode : Node = self.targetNode
in not self.leftHandSide and
(targetNode.quantification = Quantifier::ZERO_TO_MANY
or
targetNode.quantification = Quantifier::ONE_TO_MANY)
implies
parentNode.quantification <> Quantifier::ONE and
rule.getUqsCluster(targetNode)->includes(parentNode)
```

- An identifier must not refer to a new value of an attribute whose value is not assigned.

```
self.nextStateValue and not self.leftHandSide
implies
self.targetNode.assignments->exists(a |
   not a.assignTo.targetNode.oclIsUndefined() and
   not a.assignTo.targetAttribute.oclIsUndefined()
   and not self.targetNode.oclIsUndefined() and
   not self.targetAttribute.oclIsUndefined() and
   a.assignTo.targetNode = self.targetNode and
   a.assignTo.targetAttribute = self.targetAttribute)
```

- An identifier must not refer to an old value of an attribute of a node with the role create

```
not self.nextStateValue and not self.leftHandSide
implies self.targetNode.role <> ElementRole::CREATE
```

- An identifier must not refer to value of an attribute of a nested quantified node

```
let rule : Rule = self.oclAsType(ecore::EObject)
.findParentObject(Rule.oclAsType(ecore::EClass))
.oclAsType(Rule)
in let parentNode : Node = self.oclAsType(
  ecore::EObject)
.findParentObject(Node.oclAsType(ecore::EClass))
.oclAsType(Node)
```

```
in let targetNode :  Node = self.targetNode
in not self.leftHandSide and
targetNode.quantification = Quantifier::NESTED
implies
parentNode.quantification = Quantifier::NESTED and
rule.getUqsCluster(targetNode)->includes(parentNode)
```