

Dynamic Meta Modeling

A Semantics Description Technique for Visual Modeling Languages

Jan Hendrik Hausmann

D I S S E R T A T I O N

Dynamic Meta Modeling

A Semantics Description Technique for Visual Modeling Languages

Jan Hendrik Hausmann
hausmann@upb.de

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.

Paderborn, October 2005

Danksagung

Zu meinem 12. Geburtstag bekam ich von meiner Patentante das Buch "Lebendiges Wissen-Computer" geschenkt, das nicht nur mein Interesse an diesem Thema geweckt hat, sondern das in seinem Ausblickskapitel auch folgende weit-sichtige Aussage trifft:

Wahrscheinlich werden eine neue Software und neue Arten, Programme zu schreiben, die größte Auswirkung auf die Leistungsfähigkeit zukünftiger Computer und damit auch auf ihre Bedeutung für die Menschen haben.

Und tatsächlich ermöglichen heute die modernen Methodiken der Softwaretechnik die Erstellung von Programmen in einer zur damaligen Zeit wohl unvorstellbaren Komplexität. Allerdings werfen auch diese Methodiken stets neue Fragen und Probleme auf. Ich freue mich sehr, dass ich mit der hier vorliegenden Dissertation nun selbst einen Beitrag zum Gebiet der Softwaretechnik leisten kann, der einige der aktuellen Probleme für die Zukunft lösen kann (und hoffentlich wird).

Den folgenden Menschen möchte ich ganz herzlich für ihre Mitwirkung zum Gelingen meiner Promotion danken:

Gregor Engels —*Doktorvater, Chef und Vorbild*— für das präzise Benennen von Stärken und Schwächen in Text und Argumentation. Weit darüber hinaus für 5 Jahre Zutrauen und Vertrauen, für meine Freiheit bei gleichzeitiger vorbehaltloser Rückendeckung. Unschätzbar unersetzlich!

Reiko Heckel und Stefan Sauer —*the original DMMsters*— für den Grundstein des Themas und unendlich viele hilfreiche Diskussionen in seiner Ausarbeitung. Ohne euch wäre noch nicht mal der Name da!

Marc Lohmann —*Weltklasse-Bürokollege*— für das geduldige Ertragen meiner Launen und für die stete Bereitschaft sich meine halb-garen Ideen anzuhören und dann die Probleme detailliert aufzuzeigen. Wie soll ich in Zukunft anders arbeiten können?

Friedhelm Meyer auf der Heide, Wilhelm Schäfer und Theo Lettmann für ihre kritische Auseinandersetzung mit meinen Ideen, eine sehr interessante Verteidigung und die großzügige Beurteilung meiner Leistung.

Sonja, Tom Niklas, Trude und Volker für die zeitlichen Freiräume, die emotionale Unterstützung und all die Liebe, die mir auch in den anstrengenden Phasen immer gezeigt hat, wofür ich das mache.

Contents

I	Motivation and Overview	1
I.1	State of the Art	2
I.2	On the Benefits of Formal Semantics	4
I.3	Objective of this Thesis	5
I.4	Structure of this Thesis	5
II	Semantics Description Techniques for Visual Modeling Languages	7
II.1	Concepts of Languages and their Definition	7
II.2	Visual Modeling Languages and their Definition	11
II.3	Survey of Semantic Description Techniques for VMLs	23
II.4	Concept of the Dynamic Meta Modeling Approach	41
III	Meta Relations	45
III.1	On the Need for Meta Relations	47
III.2	Concept of Meta Relations	51
III.3	Concrete Syntax for Meta Relations	53
III.4	Abstract Syntax for Meta Relations	55
III.5	Semantics of Meta Relations	57
III.6	Summary and Discussion	62
IV	Graph Transformations	63
IV.1	Graphs	64
IV.2	Graphs in Dynamic Meta Modeling	68
IV.3	Graph Transformation Rules	70
IV.4	Graph Transformation in DMM	76
IV.5	Controlling Graph Transformations	79
IV.6	Control in DMM—The Mechanism of Rule Invocation	82
IV.7	Discussion	101
V	The Architecture of Dynamic Meta Modeling	103
V.1	Expressing Static Semantics in DMM	105
V.2	Expressing Dynamic Semantics in DMM	107
V.3	Model Semantics in DMM	109
V.4	Modularity and Extensibility	110
V.5	Summary and Discussion	116
VI	Case Study: Formalizing UML Activity Diagrams	123

VI.1	Eliciting the Semantics of UML Activity Diagrams	124
VI.2	Excerpts from the DMM Specification of Activity Diagrams . . .	140
VI.3	Discussion of the DMM Specification of Activity Diagrams . . .	146
VII	Pragmatic Guidelines for Formulating DMM Specifications	153
VII.1	Qualities of DMM Specifications and Heuristics for their Achievement	154
VII.2	Guidelines for Formulating the SD Meta Model and Relations .	160
VII.3	Guidelines for Formulating DMM Rule Sets	166
VII.4	Summary and Discussion	172
VIII	Automatically Applying DMM Specifications	175
VIII.1	Model Checking approaches for Graph Transformation Systems	176
VIII.2	Introduction to the GROOVE Tool Set	177
VIII.3	Translation of DMM Specifications into GROOVE Specifications	182
VIII.4	Interpreting Activity Diagrams with GROOVE	187
VIII.5	Discussion of the GROOVE Prototype	194
IX	Summary and Conclusions	201
IX.1	Summary of the Contributions of this Thesis	201
IX.2	Overview of Publications on DMM	202
IX.3	Discussion of DMM	202
IX.4	Closure	205
A	Overview of Activity Diagrams	207
A.1	History of Activity Diagrams	207
A.2	The Role of Activity Diagrams in UML 2.0	208
A.3	Activity Diagram Elements	209
A.4	Advanced Activity Diagram Elements	214
B	The DMM Specification of UML Activity Diagrams	215
B.1	Overview of the SD Meta Model for Activity Diagrams	215
B.2	Package Ordering	217
B.3	Package Core Structure	219
B.4	Package Core Behavior	222
B.5	Package Core Activities	229
B.6	Package Buffernodes	244
B.7	Package Controlnodes	258
B.8	Package Core Actions	268
B.9	Package Actions	279
B.10	Package Dummy Actions	286
	Bibliography	291
	Index	315

Chapter I

Motivation and Overview

Software Engineering is the discipline of Computer Science which is concerned with the creation process of software systems. This process is inherently complex as it entails a transformation of numerous customer requirements into sophisticated program code. The size and complexity of modern software systems require different specialists to cooperate in a team to develop a high quality product within a competitive time frame.

Two aspects are crucial for such a development to succeed: communication and automation. Communication between customers and the developers drives the development project, communication within the development team is vital for its coordination and ultimately for its success. Automation on the other hand effects productivity and quality as computers can perform certain tasks in the development process with vastly higher speed and accuracy than human developers.

Models have been found to be a very useful concept which supports both of these key development aspects. For communication purposes, models allow an intuitive presentation of concepts while abstracting from details, thereby allowing people from different backgrounds to build a common understanding about a problem and its possible solutions. For automation purposes, models document the creative design decisions necessarily taken by a human developer. Based upon these models, computer systems can be used to automatically derive an elaborated solution.

To actually fulfill the pivotal role attributed to them in a development process, models need to follow a commonly agreed language definition. On the basis of such a definition all participants can precisely express concepts in the form of a model and they can expect others to comprehend the information captured therein unambiguously. Modeling languages which feature diagrammatic notations have proven to be especially successful in terms of human comprehensibility. Such formalisms are collectively addressed as Visual Modeling Languages (VMLs).

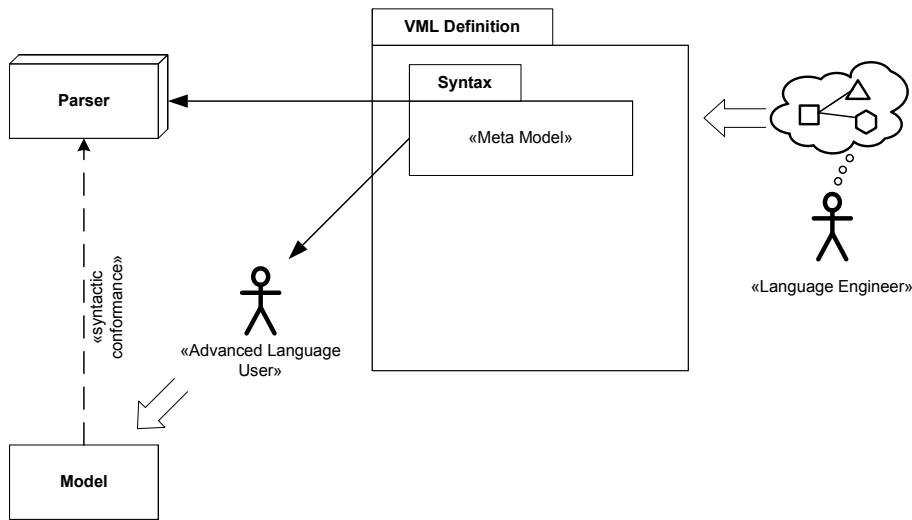


Figure I.1: Illustration of the benefits of a formal syntax definition

I.1 State of the Art

The Unified Modeling Language (UML) [Obj03e, Obj04, Obj03b] is the industry standard VML to express models in a software development process. It was incepted 1997 by combining the three (at that time) most popular modeling techniques into a unified formalism. The main goal of this standardization was to obtain a single set of well-defined modeling notations which could uniformly be used in large parts of the software industry and through all phases of a development process (a more detailed account of the UML and its features is provided in Chapter II). This goal has been successfully achieved and today UML can be regarded as the *lingua franca* in the Software Engineering community. Despite its dominant position, UML is far from being the only modeling language. A multitude of other formalisms continue to be used and new VMLs constantly emerge.

Focusing on the way a (new) Visual Modeling Language is defined, we find that the technique of *meta modeling*, i.e., the definition of the abstract syntax by means of a Class Diagram has by now been widely established. A Language Engineer (a term we will use for a person who is defining a new language) can thus provide a meta model which precisely determines how expressions (i.e., models) in the language must be constructed. The meta model is understood by (advanced) language users and forms a precise basis for automated processing. Fig. I.1 illustrates the benefits of formality in this part of the language definition. If a Language Engineer provides a language definition (creation is symbolized by the open arrows) in terms of a meta model, a precise information transfer (solid arrows) to language users is possible. Tools can process this specification to, e.g., check user-created models automatically for their conformance to the language definition.

For the definition of the *semantics* of a Visual Modeling Language no such

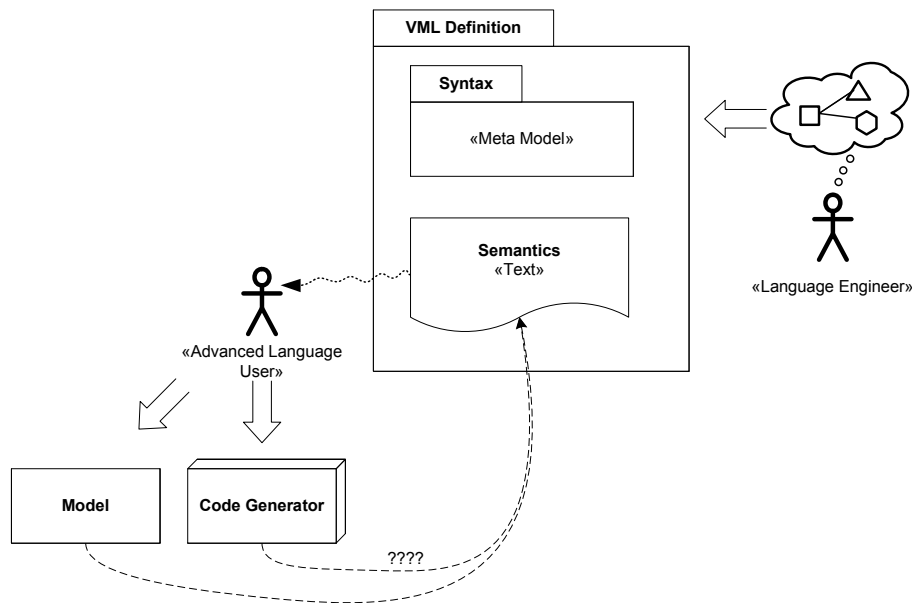


Figure I.2: Illustration of the problems of an informal semantics definition

standardized or even commonly agreed upon approach is available today. The UML (and other languages of a comparable complexity) relies on prose to convey the meaning of its various diagrams. While being very convenient for the human reader and allowing for much flexibility, using natural language text for semantics specifications has severe drawbacks: Natural language is often used imprecisely and ambiguously. In large texts (the UML specification fills more than 800 pages), omissions, inconsistencies, and outright contradictions are inevitable. Neither an automated detection nor a systematic resolution of such flaws is possible. Human inspection alone revealed more than 800 issues [FTF] during the final phase of the UML 2.0 definition. It is highly probable that a substantial amount of flaws still remains in the specification text to date.

The ambiguity of the specification text and actual flaws in the specification erode the users' confidence and invite the conception of individual language interpretations which deviate from the standard. An unambiguous settling of emerging differences is not possible. Endless debates on the correct interpretation of model elements and outright misunderstandings are the consequences of this situation. The value of models as efficient communication means is seriously decreased.

Even more serious is the impact of informal semantics definitions on automation. Semantics-dependent model processing tools (e.g., code generators, model analyzers) can never correctly implement the language's semantics but ever only *their creator's interpretation* of the semantics description. This interpretation may deviate from that of other language users. Interoperability problems between different tools and distrust in the reliability of tool-produced results are the consequence. Models can thus not fully play the role attributed to them in the development process.

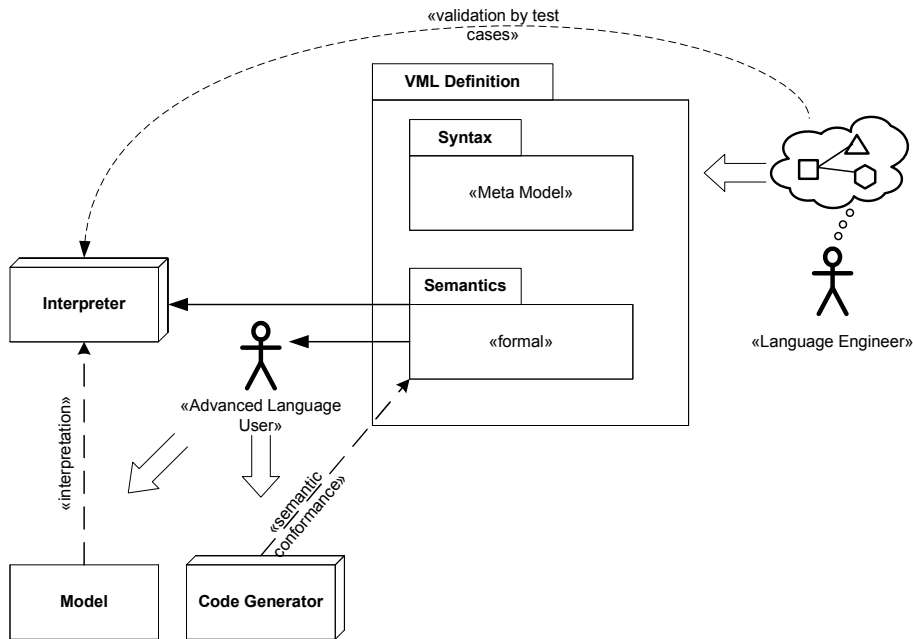


Figure I.3: Illustration of the benefits of a formal semantics definition

Figure I.2 illustrates this situation: If the semantics specification is provided as a text, it is only subject to (possibly flawed) human interpretation (indicated by the wavy arrow). This interpretation is then the base for creating models and for model-processing tools build by advanced users. Neither for models nor for tools a conformance to the language definition can be proven in a systematic way.

I.2 On the Benefits of Formal Semantics

The introduction of a formal semantics can vastly improve this situation in a number of ways (as illustrated in Fig. I.3). The most obvious difference to informal semantics is that is that formal semantics form an precise point of reference against which language users can build their understanding of the language—provided the semantics definition is presented in an understandable way.

Tools can either process the semantics definition directly (in the illustrating Figure I.3 an Interpreter is such a tool) or their implementation can be proven to be in accordance with the official semantics. Most important among these tools are interpreters, i.e., tools which take a specific model as their input and automatically derive its meaning(s) (or interpretation) according to the defined semantics. Users can employ such tools in an exploratory style of learning to refine and extend their knowledge about the language. This is similar to the learning of programming languages which is also usually done in a combination

of general definitions of constructs and their exploration in individual programs. Having these two points of reference allows for a thorough and precise understanding of the language, preventing misunderstandings in communication by models.

An indirect benefit is that the quality of the language itself is bound to improve: Based on the automated interpretation enabled by formal semantics, Language Engineers can validate by test cases whether their language definition correctly reflects their intentions. Flaws in the definition can be detected by this testing or by subjecting the definition itself to formal analysis. This effects a higher quality of the language specification.

The effect of formal semantics for modeling languages is thus that models can fulfill their purpose as unambiguous means of communication and as a precise basis for automation in software development.

A suitable technique for specifying the semantics of a Visual Modeling Language like UML do thus have to combine two major goals: For automation purposes it needs to be precise and formal. For human comprehension it also needs to be understandable. While a number of techniques have been put forward in the literature (see Chapter II for an overview), none of them has found widespread recognition (as witnessed by the OMG's explicit refusal to employ any of these techniques in the definition of UML 2.0). We believe that the reason for this failure is due the fact that none of these specification techniques is explicitly designed to address the problem of *defining* the semantics of a Visual Modeling Language (we elaborate the requirements stemming from this specific scenario in Chapter II).

I.3 Objective of this Thesis

The objective of this thesis is to supply a technique for the specification of a Visual Modeling Languages' semantics. The technique should combine formality and precision with a high degree of understandability for a human reader. In fact, the specifications resulting from its application should be fit to serve as part of a published language definition. The specification technique must provide convenient support to express core features of Visual Modeling Languages in general and UML in particular, e.g., the combination of structural and behavioral models, incomplete models, and possible user-defined language extensions.

I.4 Structure of this Thesis

The core of this thesis is the introduction of Dynamic Meta Modeling (DMM), a technique for the specification of the semantics of Visual Modeling Languages, which fulfills the above stated requirements. The presentation is divided into nine chapters (including this introduction).

In Chapter II, "SEMANTICS DESCRIPTION TECHNIQUES FOR VISUAL MODELING LANGUAGES", we elaborate the premises of the thesis in greater depth. After introducing basic terminology used in the thesis, we detail the requirements for a semantics specification technique for Visual Modeling Languages. A number of existing approaches are surveyed and evaluated according to these requirements. Finding all of them lacking, we proceed to outline our own approach of Dynamic Meta Modeling. To fill this outline the next two chapters present technical information on separate components of DMM which are combined in Chapter V.

Chapter III, "META RELATIONS", introduces a novel technique for establishing meta model relations. Such relations can be employed in the style of denotational meta modeling to provide semantics in the denotational style for structural model elements.

Chapter IV, "GRAPH TRANSFORMATIONS", explains the basic notions of the Graph Transformations and proceeds to provide a set-theoretic formalization of an innovative rule style using the mechanism of invocations to combine different rules to a complex transformation. This style of rules allows for the formulation of operational semantics, a kind of semantics ideal for the description of behavioral elements of a modeling language.

Chapter V, "THE ARCHITECTURE OF DYNAMIC META MODELING", combines these two techniques to form DMM specifications which then allow for the specification of a complete language's semantics. Additionally, DMM specifications provide a modularization concept allowing for easy maintenance and extensibility.

The applicability of the DMM technique is demonstrated in Chapter VI which contains an elaborate case study targeting the formalization of UML 2 Activity Diagrams. An inspection of the relevant UML semantics descriptions reveals a number of problems which have to be resolved prior to formalization. The presentation and discussion of the formalization's results provides insights on the practical realization of the approach's promised qualities..

How a DMM specification is to be created is the topic of Chapter VII, "PRAGMATIC GUIDELINES FOR FORMULATING DMM SPECIFICATIONS". Here, we discuss the notion of qualities of a specification and which considerations underlie their realization. We provide a small methodology which provides guidance to Language Engineers employing DMM to specify a language's semantics.

In Chapter VIII, "AUTOMATICALLY APPLYING DMM SPECIFICATIONS", we focus upon the interpretation of single models. For this we provide a prototypical DMM interpreter based on the GROOVE tool set [Ren03a] which is able to automatically construct the precise semantics for a model expressed in a language specified using DMM.

Having thus covered the definition and application of DMM, we discuss its achievements and remaining open problems of specifying a Visual Modeling Language's semantics in the concluding Chapter IX.

Chapter II

Semantics Description Techniques for Visual Modeling Languages

The focus of this thesis are Visual Modeling Languages and especially the techniques used to define their semantics. This chapter provides an in-depth introduction to these topics and thus forms the basis for the elaboration of the thesis. It details and backs claims and statements made in the overview in Chapter I.

We explain the fundamental concepts underlying the definition of languages in Section II.1. The terminology introduced there is essential for precise discussions throughout the thesis. The field of Visual Modeling Languages in general and the UML in particular are the topics of Section II.2. Here, we elicit characteristics of these languages, resulting in requirements for a semantics definition. Based upon these requirements we present a survey of existing approaches in Section II.3. While there is currently no approach to fulfill all these requirements, a combination of ideas from different approaches seems promising. We outline this combination in Section II.4 and pursue its detailed definition in the remainder of the thesis.

II.1 Concepts of Languages and their Definition

To lay out the concepts of a language's definition we need to define some fundamental terms from Linguistics in general and Formal Languages in particular, as the latter are usually of interest in Computer Science.

Let's first turn to the general term language itself:

Language is a system of finite arbitrary symbols combined according to rules of [a] grammar for the purpose of communication. Individual languages use [...] symbols to represent objects, concepts, emotions, ideas, and thoughts. [Wik05]

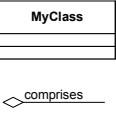
LANGUAGE TYPE	EXAMPLES	DEFINITION BY
Natural Language	"horse" "John"	Word list (dictionary)
Programming Languages	<code>until</code> <code>xy\$</code>	Keyword list Regular expressions
Visual Languages		List of shapes and connections

Table II.1: Overview of different definition techniques for a language's concrete syntax

To fulfill its communication purpose a language must not be an ad hoc construction but needs to have a stable definition which is shared between the participants of the communication. This definition comprises three parts: The *concrete syntax*, the *abstract syntax*, and the *semantics*.

II.1.1 Concrete Syntax

The concrete syntax definition provides the set of symbols used to construct expressions in a language (cf. Table II.1). In natural languages and programming languages this set comprises words constructed from the characters of the western alphabet and enumerated in dictionaries. The creation of new symbols (identifiers) in programming languages furthermore underlies rules provided by regular expressions. In Visual Modeling Languages we usually find an assortment of shapes and connecting lines as the concrete symbols. Complex syntax representations may call for additional definition means like grammars or multiple layers of definition.

II.1.2 Abstract Syntax

The abstract syntax definition sets out rules for the combination of language elements (words) to complex constructs (sentences). Such rule sets are called grammars. Grammars can be categorized according to their expressive power. The so called Chomsky hierarchy distinguishes between unrestricted, context-sensitive, context-free, and regular grammars. Natural languages conform to only partially formalized and generally unrestricted grammars. The definition of programming languages is usually split up into a context-free part (expressed in the Backus-Naur form) and context-sensitive well-formedness conditions (traditionally but misleadingly called *static semantics* in the field of programming languages, cf. [HR00, Mos01]). Attributed Grammars can be used to express such conditions for programming languages [Kas91], but often they are provided textually only and made precise by reference implementations of compilers. Visual Languages nowadays mostly use a combination of meta models and OCL


LANGUAGE TYPE	EXAMPLES	DEFINITION BY
Natural Language (English)	A relative clause is a clause introduced by a relative pronoun.	Description
Programming Languages	<pre><if_statement> ::= <bool_exp> <statement> <statement></pre> <p>The declaration of a variable needs to precede its first value assignment.</p>	BNF (context-free parts) Attribute Grammars, Text/Compiler (context-sensitive parts)
Visual Languages	 <pre> classDiagram class Class class Association Class "1" -- "*" Association : target Association "*" -- "1" Class : source </pre>	Meta model & OCL, Graph grammars

Table II.2: Overview of different definition techniques for a language’s abstract syntax

expressions [Obj03c] to define their abstract syntax. Graph grammars have also been used to define the abstract syntax of visual languages [Sch77]. The concept of attribute grammars has been taken up in the PROGRES system [Sch90]. Table II.2 provides an overview of the definition methods for abstract syntax.

The benefits of using such formal definition techniques for the definition of abstract syntax are witnessed by the existence of tools which can automatically process specifications given in certain formalisms. Scanners can automatically tokenize the elements of a given expression on the basis of a keyword list and regular expressions. Parsers can proceed to reveal the abstract structure of an expression based on BNF grammars. In fact, such tools can be constructed generically by so called scanner/parser generators (see [Com05] for an overview of available generators).

For Visual Modeling Languages, the parser is usually integrated in a syntax-directed editor which guides the construction of expressions. For formal abstract syntax definition techniques, the construction of generic editors which are customized by a concrete language definition is possible. Examples include the Kent Modeling Framework [kmf] (definition by meta models), DiaGen [MK00, MV93] (definition by hypergraph grammars) and VL-Eli [KS02] (definition by special attributed grammars). The whole approach of Software Factories for Domain Specific Languages (DSLs) [GSCK04, Fow05a] is based upon the concept of generating such specific tools on the basis of a formal (meta model based) language definition.

The abstract syntax serves as the basis for semantics definitions. In the remainder we will thus regard expressions usually not in their concrete form (unless we want to present examples) but in their abstract representation. Fig. II.1 illustrates the difference for a UML class (top of figure) and a UML action with an outgoing control flow. The left hand side of the figure shows the concrete

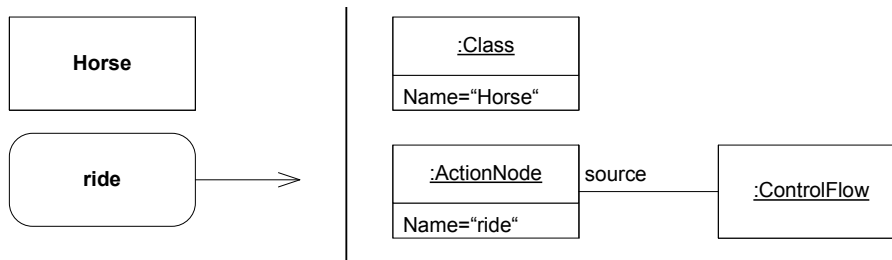


Figure II.1: Examples for the difference between concrete and abstract syntax of UML elements

representations, the right hand side their abstract counterparts (expressed as instances of the UML meta model).

A direct implication is that we consider only such expressions for semantics definition which are legal in the language and thus allow for such an abstract representation.

II.1.3 Semantics

While the syntax of a language is concerned with the form of its expressions, the semantics of a language is concerned with their *meaning*. Semantics thus form a bridge between the concepts expressed *by* the language and the symbols used *to* express them. This general outline immediately gives rise to another problem: As concepts, ideas, and perceptions of reality primarily exist in the mind of a person, how do we relate them to (written) symbols? If we try to explain them, what are the semantics of the language we use for explanation? Such questions have been the basis for a philosophical examination of the term "semantics" and its implications, resulting in *Tarski's indefinability theorem* [Tar44], which states that a language cannot be used to explain its own semantics. The implication of the theorem is that an assumption-free definition of semantics cannot exist:

Further important results can be obtained by applying the theory of truth to formalized languages of a certain very comprehensive class of mathematical disciplines; only disciplines of an elementary character and a very elementary logical structure are excluded from this class. It turns out that for a discipline of this class the notion of truth never coincides with that of provability; for all provable sentences are true, but there are true sentences which are not provable. ([Tar44], Sect. 13)

Computer Science usually takes a more pragmatic view on this dilemma and simply assumes that a sufficiently well understood formalism exists to express semantic concepts. If all else fails, pure mathematics is often regarded as the least common denominator for definitions. We do thus assume that a so called *semantic domain* exists which captures concepts to be expressed by a language. Syntactic terms are then assigned a meaning by relating them via a *semantic mapping* to semantic elements [HR04, CEK⁺00].

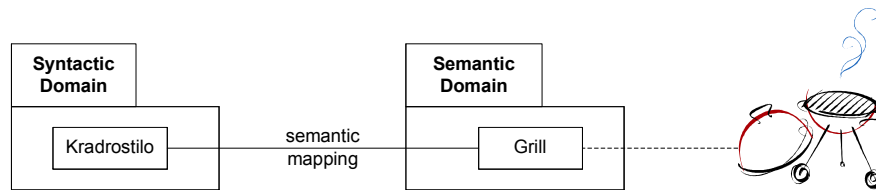


Figure II.2: Example illustrating the roles of syntactic domain, semantic domain, semantic mapping, and represented concept

In Fig. II.2 we illustrate these terms with an example from natural languages: The syntactic term "Kradrostilo" (a legal word in the artificial language Esperanto) is assigned a meaning by relating it to the German term "Grill". This construction assumes that the term in the semantics domain ("Grill") is sufficient to capture and convey the general concept of the cooking device which is addressed by it.

We can furthermore distinguish between *static semantics* and *dynamic semantics*. Static semantics express the structural meaning of a language term ("what something *is*"). Dynamic semantics are concerned with the behavior expressed by a language term ("what something *does*"). Precisely defining the latter is usually harder than the former.

We can summarize that language definitions in general comprise definitions of concrete and abstract syntax as well as semantics in the form of a semantic domain and a semantic mapping. The forms in which these components are expressed vary greatly.

We now focus more concretely on that segment of languages which we are interested in: Visual Modeling Languages.

II.2 Visual Modeling Languages and their Definition

We begin the discussion of Visual Modeling Languages by giving an overview of the dissemination and diversity of this type of language in software development (Subsect. II.2.1). Focusing on the common core concepts of VMLs (Subsection II.2.2) we find UML to be the embodiment of these concepts. Moving on to the definition techniques for VMLs, Subsect. II.2.3 lays out the concepts of meta modeling as used in the UML. Goals of and requirements for a formal semantics definition technique for VMLs are then collected in Subsection II.2.4.

II.2.1 Overview of Visual Modeling Languages

Modeling has found wide-spread acceptance in software development as it allows for an abstract representation of certain aspects of either a problem or a pro-

posed solution. Taking different domains of system development into account (e.g., database design, telecommunication systems, embedded software, user-interface design, business process support) we find very different aspects which are important to these domains and which are consequently addressed by a multitude of modeling notations: Database design requires conceptual data modeling, served by Entity Relationship diagrams (ER-diagrams, [Che76, AE96]), telecommunication systems are concerned with inter-component communication as visualized by Message Sequence Charts (MSCs, [ITU93]), embedded systems emphasize real-time aspects as expressed by, e.g., Timed Automata [AD94], user-interface design is mainly concerned with reactive behavior emphasized in variants of Statecharts [Hor99], and finally business process support requires the modeling of alternative and interwoven flows of control and information which can be visualized with Event-Process-Chains [KNS92]. Besides these rather practically oriented formalisms there are also Visual Languages stemming from theoretical considerations, e.g., Petri Nets [Pet62, Rei85].

Intensive research on VMLs yielded numerous variants of the above enumerated formalisms to provide ever more detailed features. On the other hand, the exact nature of what a model actually is and which characteristics a Visual Language must have to be suitable for the expression of models remains an open question for high-level philosophical debates, conducted, e.g., in [Kue05, Bez04, Fav04, Lud03, Sei01]. The whole field of Visual Modeling Languages has in fact by now grown so large that several scientific conference series [MB01, GK01, RH04, Min04, BMS04] and journals (Sosym, JVL) are solely devoted to documenting the progress in this area.

II.2.2 Characteristics of Visual Modeling Languages

The necessary pragmatic reduction of the discussion from the complete field of Visual Modeling Languages and its open philosophical ends to commonly and practically used core concepts is simplified by the existence of the Unified Modeling Language (UML) [Obj01, Obj04].

The UML was explicitly constructed to address the proliferation of modeling languages by capturing the commonly found concepts of VMLs into a single formalism. Störle characterizes the achievements of the UML as a *consolidation* of the field of VMLs, its *integration and standardization* and the selection of *core concepts of visual modeling* ([Stö05c], p.23¹). The success of the UML in achieving this consolidation is undeniable; by now (roughly 8 years after the publication of the UML 1.1 [UML97]) it has become the lingua franca of the Software Engineering community, being used in the industry and academia alike.

In the following discussions we will thus regard those features of Visual Modeling Languages as "common" which are integrated in the UML. UML is also by far the most complex VML in the field due to its unifying nature. Additional problems need to be addressed in its definition which can be sidestepped in smaller notations which focus on a single aspect only. Consequently, the UML is used as the primary subject of discussion in this thesis and examples are presented using UML notations. The results in the remainder of this chapter

¹translated from German

and the technique introduced in this thesis do, however, also apply to other visual languages which convey similar concepts to those found in the UML.

Characteristics of the UML (especially those with an impact on the semantics definition) are elicited in the UML Semantics FAQ [KER99], a document combining the opinions of some of the most influential academics in the UML community at that time:

UML does have some specific characteristics, which makes the task of semantics definition interesting:

- a) *A substantial part of UML is visual/diagrammatic.*
- b) *UML is not for execution, but for modeling, thus incorporating abstraction and underspecification techniques.*
- c) *UML is combined of a set of partially overlapping subnotations.*
- d) *UML is of widespread interest.*

At least b) has to be amended by now as UML 1.5 introduced the so called UML Action Semantics [Obj03e], which "are now defined in as much detail as a programming language" [Kob04]. We can generalize that UML models *can* comprise underspecification and abstraction, but not necessarily.

In addition to c) Harel and Rumpe emphasize the need to consider structural as well as behavioral subnotations in formulating semantics [HR00]:

One common misconception in the world of system modeling languages is to take semantics as a synonym for behavior. Both the behavior and the structure of a system are important views thereof; both are represented by syntactic concepts and both need semantics.

Bran Selic, one of the principal authors of the UML standard, formulates an important characteristic of these behavioral views [Sel04]:

UML behavioral semantics only deal with event-driven, or discrete, behavior.

The strive for universality in UML's application also causes two rather uncommon characteristics: The UML's definition is intentionally incomplete and open to change. To allow for varying interpretations of controversial details, the UML comprises the concept of *semantic variation points*. Such variation points designate details in the specification which are intentionally left open. At best, choice lists for possible interpretations are provided (e.g., events received during a transition can be regarded as errors, they can be discarded or deferred). At worst, crucial details are left completely unspecified ("*The precise lifecycle semantics of aggregation is a semantic variation point.*" [Obj04], p.53).

Going even beyond this local adaptability is the *language extension* concept of UML. To allow for different application domains to endorse UML even though standard UML does not fully address their modeling needs, UML allows for user-defined language extensions. A language extension (technically combined in a so called *profile*) can restrict the core UML to a subset suitable for the domain, add new symbols to extend/replace standard UML notation, and introduce completely new concepts to the language. The term extension is thus misleading as extensions as well as restrictions can be effected by a profile. While the syntactical alterations to standard UML underlie several restrictions ([Obj04],

Chap. 18), the only restriction on semantic alterations is that *“the specialized semantics do not contradict the semantics of the reference meta model²”*. Details on the profile mechanism may be found in [CKM⁺99], a discussion of different extension mechanisms is provided by [RCA01].

To summarize: We focus our discussion primarily on the UML because it unifies the core concepts of many other Visual Modeling Languages. Its characteristics are that it is a visual language which expresses static and discrete dynamic concepts in partially overlapping views. It is intended for a wide audience and may contain underspecification both in the definition and in its expressions. The language is furthermore open to syntactic and semantic extensions.

Before we elicit concrete requirements toward a semantics definition of the UML from these characteristics, we first explain the UML’s syntax definition in more detail.

II.2.3 Definition of the UML’s Syntax by Meta Modeling

The Unified Modeling Language is an industry standard published by the Object Management Group (OMG). Currently, version 1.5 [Obj03e] is still the official standard (bearing the formal designation “Published Specification”), but the version 2.0 [Obj04] of the language is in the last stage of its inception and its specifying documents are already available. Unless noted otherwise we will already refer to the upcoming standard UML 2.0 as “the UML”.

The UML 2.0 specification comprises an infrastructure part (which defines a language kernel) and a superstructure part (which provides the front-end language). In this thesis we usually refer to the superstructure part as the UML definition. The superstructure specification is an 800+ pages document which contains a structured list of all language elements found in UML. The concrete syntax for each element is provided by means of example renderings and descriptions of presentation options³. Each element can underlie certain context-sensitive restrictions, expressed in the Object Constraint Language (OCL)[Obj03c]. The abstract syntax of the language is defined by a meta model and its semantics are given by explanations in natural language. We will take a closer look at the technique of meta modeling first as it is a truly remarkable feature of UML.

As explained above, the inception of UML was clearly aimed at an integration of existing concepts. In its definition method, however, the UML used a rather innovative technique. This technique, the definition of abstract syntax by means of a meta model, was not only readily accepted by the community but it soon became a common way of expressing the abstract syntax of modeling languages even beyond the UML (e.g., meta models are used to define Petri-nets in [VP03,

²Obviously the reference *language* is meant here as the meta model itself does not specify semantics.

³The concrete syntax of the UML is—contrary to common belief—*not* standardized. As emphasized in [Boc03d], UML models can be expressed using different symbols or text entirely.

dLETE04], an ASM meta model can be found in [KN03]). Meta modeling was not invented with the inception of the UML, though. Previously it was used to compare and integrate different OO analysis and design approaches (see [Ode97]) with the COMMA meta model (Common Object Methodology Metamodel Architecture) as its most prominent result. And even before that it was a technique to facilitate interoperability between different CASE tools (e.g., [Fla02] gives an overview on the development and application of the CDIF meta model). UML's contribution is the wide dissemination of this concept as a language definition technique.

Two main properties can be identified as the cause for this rapid success: adequacy and accessibility. Visual Modeling Languages do not usually have a tree as the underlying structure in their abstract syntax, but they conform to the more general concept of graphs (cf. [EH00, GPP98]). While tree-based structures can be easily captured in (context-free) grammars with each rule expanding one element of the tree, graphs can only be described (in general) by context-sensitive graph grammars which are much harder to grasp. Meta models changed the basic approach of syntax definition from an operational (production rules of a grammar) to a declarative one. By presenting this declaration visually, they were able to visualize the underlying graph structure in an intuitive way. Thus, declarative meta models are more *adequate* for expressing graph structures than operational (grammar) approaches.

The second property which made meta models a success is an idea known from meta-programming [DM95] or reflective languages [Coi96, dRS84]: the expression of language features using the language's own notation. For UML this means that the expression of its meta model uses the notation of Class Diagrams. The huge advantage of this approach is that the potential recipients of the definition (i.e., advanced UML users delving into the details of the specification) already have a firm understanding of the notation and are spared the effort of learning a special definition notation. It is presumably this easy *accessibility* for the intended audience which contributed most to the rapid adoption of meta modeling as an established method for defining a VML's abstract syntax.

A further important achievement of the UML's meta model is that it allows for the integration of the separated subnotations of UML. Since all elements of the UML syntax are defined by the same meta model, meta associations can be used to express relations between elements even if they are strictly separated in the notation. E.g., the meta class *Activity* (used to represent concepts in the behavioral notation UML Activity Diagrams) can be connected to an operation of a class (used in the structural Class Diagrams) to express that the activity specifies the operation's behavior. Semantics specifications can thus largely disregard the different subnotations and be based on a single integrated meta model.

Technically, however, meta modeling gives rise to a number of rather grave issues. While the notation of the meta model is the same as that of UML's Class Diagrams, the meta model is actually defined according to the OMG's Meta Object Facility (MOF) [Obj02a, Obj03a]. MOF thus defines a meta-meta model which provides the concepts used for the definition of the classes in the UML meta model. The UML meta model is thus not really a UML Class Diagram, it just happens to look a lot like one.

Fig. II.3 provides an overview of the OMG's four-leveled meta stack. The example is taken from the UML specification and illustrates simultaneously the concept and some problems of the architecture: On the topmost level (M3) the Meta Object Facility's meta meta model is located. It provides the constructs which can be used in the creation of a language's meta model. One such element is `Class`. The usage of classes in a meta model is shown on the M2 level. Here, a fragment of the UML meta model is displayed which itself defines the language constructs to be used at the model level. The displayed constructs are `Class` and `Attribute`, i.e., we are looking at a fragment of the Class Diagram specification. Models (M1) can now employ these elements to express structures in a problem domain, e.g., for video rentals. The M0 level finally contains the objects which the model is supposed to represent, in case of a software model these are usually supposed to be run-time elements.

Some problems are also demonstrated by the figure as there are actually two instances of the `Video` class, one on M1 and one on M0. The detailed relation between these two notions of instance was left unclear in UML 1.x. In UML 2 the problem was principally resolved with replacing the `Instance` class from the UML meta model by an `InstanceSpecification` class. That change has obviously not been propagated to this architectural overview figure and thus the figure also serves to demonstrate the inconsistencies in the UML specification.

The original construction of UML and MOF (in their 1.x versions) has drawn a lot of criticism [AK01, BG01, AKHS00, VP03, CEK⁺00, AES01a, Atk99] which can be summarized in four distinct topics:

- a) The four-level hierarchy was in general poorly understood⁴. A main contribution to this problem is the fact that actually three levels of the hierarchy (in Fig. II.3) comprise identically looking class constructs, while the fourth (M0) completely lacked constructs for its expression. This overloading of notational constructs is one of the inherent dangers of a meta modeling approach.
- b) MOF was not an exact subset of UML's Class Diagrams. Several details differed between both notations, making MOF-based meta models prone to misinterpretations. Consequently, the alignment of UML and MOF was a major aim of the MOF 2.0 and UML 2.0 infrastructure formulation [Obj03b, Obj03a].
- c) MOF did neither provide a precise instantiation notion (i.e., it remains unclear when *exactly* an M_2 meta model is an instance of MOF), nor did it shed any light on the instantiation of a meta model described using MOF (i.e., the relations between M_1 models and their M_2 meta model definitions) [CEK⁺00, AK01, Atk99, AES01b].
- d) Since MOF is a technology to define meta models, it is used to define itself, i.e., MOF is defined recursively. The exact interpretation of MOF and its meta models thus depends on the fact that the user already has an understanding of MOF and meta modeling. This circularity is a fundamental

⁴There is even evidence for severe confusion in the OMG on the precise roles of the different meta levels: The whole UML infrastructure specification switched levels during its development. Compare, e.g., draft version 0.6 [Ale01] with the final version [Obj03b] of the UML 2.0 Infrastructure specification

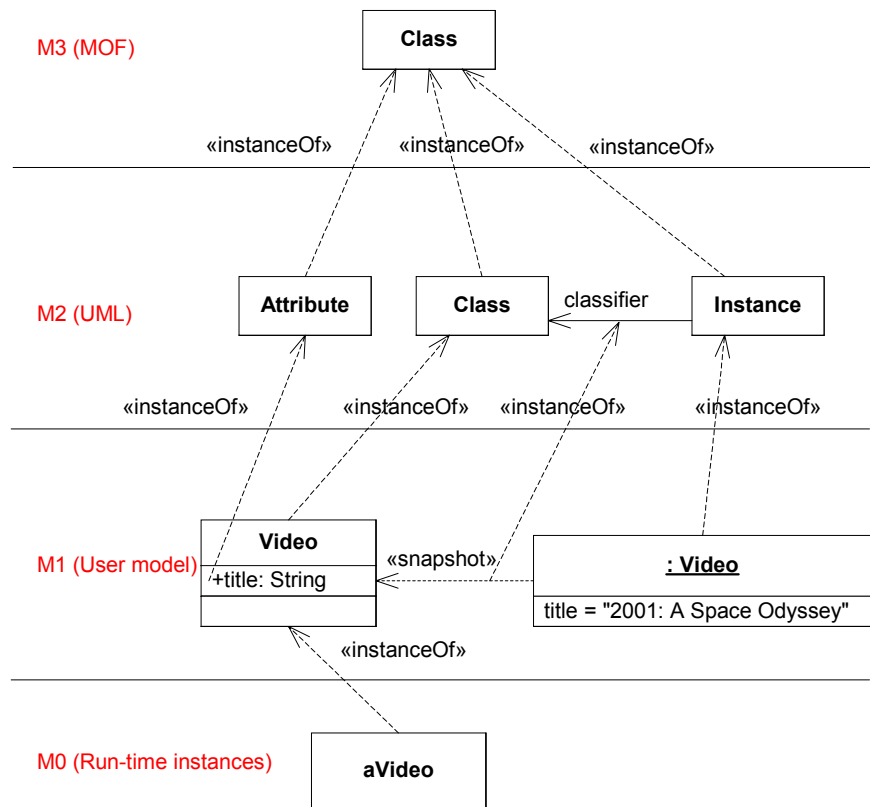


Figure II.3: Illustration of the four layer structure of the UML/MOF framework (reproduced from [Obj03b])

issue in meta modeling approaches [AKHS00, BG01, VP03].

Concerning d), different approaches were put forth to provide a firm base for the definition of meta models, by basing them on set-theory [Baa02], Object-Z [GKM98], or specially designed calculi [VP03, CEK01]. Other researchers adopted a more constructive approach by iteratively extending meta models with tool support in a bootstrapping fashion [kmf, SLKN01].

Substantial contributions toward a more precise (and possibly level-spanning) concept of meta instantiation have been made by Colin Atkinson and Thomas Kühne [Atk99, AKHS00, AK01, AK02b, Kue05, Küh05] who defined the notions of *strict meta modeling* [Atk99, AK01] and provided a detailed suggestion for renovation of the MOF/UML architecture [AK02c]. The *precise UML group*⁵ also pursued this goal. Based on an IBM-sponsored study on the feasibility of re-architecting the UML [CEK01] they developed the Meta Modeling Language MML and elaborated its implications for meta modeling [AES01b, AES01a]. A core idea was that a meta model should not only provide elements for the level below it, but also define the relationship of this level to its instances (i.e., M_n defines the constructs for M_{n-1} and the instantiation between M_{n-1} and M_{n-2}). While these works propose remedies for c) they come at the price of an even more complicated hierarchy notion, thus worsening a).

The forthcoming version 2.0 of MOF [Obj03a] mainly targets the alignment problems summarized under b). Its architecture basically remains unchanged. If anything, it has been obfuscated even further by re-using the UML infrastructure (i.e., the UML language core) in the MOF, distinguishing between an essential MOF (called EMOF) and the complete MOF (CMOF) and explicitly adding reflection facilities. There are furthermore several other Requests for Proposals to extend MOF even more (e.g., the Queries, Views, and Transformations RfP [Obj02b]). Addressing the understandability problem a) and its resolution in MOF 2.0, the specification provides the following statement:

Suffice it to say MOF 2.0 with its reflection model can be used with as few as 2 levels and as many levels as users define.

We doubt whether this insight serves to alleviate the user's confusion.

Summarizing we can state that meta modeling enjoys a high acceptance on the user's side and that the UML meta model achieves an integration of the different UML notations. The technical realization of meta modeling in form of the Meta Object Facility fails to convincingly solve the technical issues arising out of the inherent circularity of the approach.

Having thus covered the syntax definition of the UML we now proceed to discuss its semantics definition.

⁵See also www.puml.org

II.2.4 Requirements for a Technique for Semantics Definitions

While meta modeling has become an accepted technique for the definition of the abstract syntax, the UML's informal approach to semantics definition remains an issue of heated criticisms. The three main arguments are:

- ◆ The UML's semantics is *unsystematic*. Being organized along the syntactic elements, semantics information is spread all over the place and there is no systematic way to collect all information pertaining to a special situation. There is also no systematic way to derive precise interpretations of special cases or to resolve the frequently occurring contradictions.
- ◆ The UML's semantics text is subject to *human interpretation*. The interpreter's expectations, experiences, and demands will influence this interpretation.
- ◆ The UML's semantics definition cannot be subject to *automated processing*. The formality of syntax definitions has allowed for the creation of a high number of very useful and generic language processors. All applications dealing with the semantics have to be hand coded, based solely on their creator's interpretation of the specification.

The essence of the criticism is that there is often no way to either prove or disprove that a certain interpretation of a UML diagram is correct with respect to the language standard. UML models can thus not be regarded as precise and reliable means of communication in a development process.

Formal semantics descriptions promise to improve all criticized aspects of this situation. Numerous suggestions for such formal semantics descriptions have been brought forward by the academic community. The OMG, however, decided to stick with the "precise natural language" [Obj03b] approach for the description of the UML 2.0. Obviously, there are more factors to consider than the ones enumerated above. In the following paragraphs we take a closer look at the strengths and weaknesses of the natural language style of semantics definition. Our goal here is to derive requirements for a semantics description technique which should ideally be able to replace the current style of semantics definition in future UML versions and other VMLs of a comparable nature.

The first important distinction is between the *form* and the *content* of a semantics definition. The content of a semantics definition are the concepts it describes. The form is the way these concepts are described. There is little doubt that an 800+ pages long document which is the result of a strenuous agreement process⁶ in which more than fifty parties participated [Kob04] simply cannot be free of odd design choices, inconsistencies, omissions and outright contradictions (some of which are enumerated in [AR02, HS01, RW99, SG99, FQL+03], many more can be found in [FTF]). These, however, are all faults which lie with the content of the specification and which are largely (albeit not completely) independent of the technique used for its description (i.e., its form). It is to be expected that a formal specification would (at least initially) also reflect these

⁶Our favorite quote from such a discussion is from UML's (at that point rather exasperated) chief author Bran Selic: "Otherwise, I will simply defer the issue until God comes down to Earth and reveals the Truth." [FTF], issue 6902

inherent faults in the diverging semantic concepts of the various UML authors. Equally all claims that the UML either omits important concepts for some domains [HS99] or is already overburdened with too many features [Mey97] target the content of the specification only. In this thesis we are not concerned with the content of a language specification unless it has implications for the specification technique.

An existing connection between form and content is that formal specifications allow for an easy and systematic, sometimes even automatic detection of certain types of errors in a specification. Natural language cannot be processed this way. The detection of flaws in the UML specification thus relies on human inspection only. A semantics description technique which allow for automated analysis of the specification would thus increase the quality of the UML specification. To allow for such processing, the specification technique needs at least to be *formal* (i.e., all of its part must be in some well-defined and itself precisely defined format). Which further properties need to be fulfilled to enable certain kinds of analysis is hard to discern from this general point of view. Usually a high number of powerful control features makes analysis harder. We do thus request a generic *analyzability* in that the semantics definition technique is minimal in its features.

A problem clearly related to the form of the semantics definition is the preciseness of its interpretation. Currently, even the UML's authors themselves can get into heated debates on the interpretation of the specification text. A semantics description that is *precise* would eliminate such arguments. Preciseness captures the fact that for a given model of the language its meaning can be systematically and precisely constructed. The best proof for such preciseness is the existence of what is often called a *reference implementation* of the UML [Kob04, Ste04, HK04, Tho03]. The term (and the idea behind it) stems from the way programming languages are published: A specification text is shipped together with a reference implementation of a compiler for the language. Existing ambiguities in the specification text are removed by the compiler which precisely determines which programs are admissible and which are erroneous.

Reference implementations have another important advantage: They allow users to learn about the general technique as well as trying out specific examples by directly compiling/executing them. Especially the latter facility is very helpful in picking up new notations as "*people seem to be a lot better at dealing with specific examples first, then generalizing from them, than they are at absorbing general abstract principles first and applying them in particular cases*" [Tai97]. Such a facility is also of immense help to Language Engineers as writers of a language specification since they can validate whether their specification ('what they write') actually matches their intentions ('what they want to write').

Note, however, that precision in the semantics specification technique only implies that it is possible to unambiguously say whether a certain syntactical structure has a particular meaning or not. It does *not* imply that there is only one such meaning. This distinction is stressed in [HR00, HR04, KER99] to alleviate fears that in providing a precise way to specify the semantics of the UML the inherent abstractness of UML would get lost, ultimately resulting in a programming rather than a modeling language (cf. [Tho03, Ste04, Fow05b]). A semantics specification technique should rather be prepared for the fact that

UML elements can in general have a whole set of possible meanings (deriving from their abstractness as well as their diverging interpretation in different domains). The reference implementation which we have in mind is thus significantly different from approaches toward executable UML [MB02] in that it not necessarily creates in a deterministic behavior sequence as its result.

Focusing only on the shortcomings of the UML's specification technique without acknowledging its successes would be foolish, though. After all the (flawed and imprecise) 1.x versions of the language attracted a very large user base, probably surpassing that of any other VML⁷. The vast advantage that the informal style of specification by explanations holds over all formal specification methods is that it is easily accessible for anyone speaking English. No prior knowledge of mathematics, formal calculi, or special programming languages is required to read the specification⁸. This intuitive accessibility is the main reason for the OMG to stick with natural language descriptions to convey semantics. Any technique which seriously tries to compete with (or even replace) the current style of semantics definition needs to make *understandability* for a wide audience its prime requirement. For modeling languages, this point is also made, e.g., in [KW01, KHH⁺97, Sch96a, HR04, KER99] and looking to programming languages we find rather similar problems. Marcotty et al. formulated them in conclusion of their 1976 survey of (programming) language specification techniques [MLB76]:

Formal definitions must never be thought of as self-contained arenas with no user contacts. The interface with users is the key area where most of the effort is needed. The metalanguage of a formal definition must not become a language known to only the high priests of the cult.

The notion of understandability is strongly related to the expected audience. The one group which will certainly come into contact with the semantics definition technique are the writers of the specification. We call this group *Language Engineers*. In the case of the UML these are primarily the members of the OMG's committees concerned with incepting or improving the language standards. On the reception side of the specification we find a group called *Advanced Language User*. Under this heading we capture all users of UML who have a deeper interest in its 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. We do in fact not expect the average UML user to be in this group. For the basic users of the language explanatory texts (more like those provided in introductory textbooks today [Stö05c, Stö05b, FS00, Pen03] than the actual standard's text) should complement the formal definition. Language Engineers are actually a subset of the Advances User group as every advanced user may rightfully set out to define his own UML extensions and thus be confronted with the task of writing semantics specifications.

⁷Exact numbers are hard to come by, but a recent study of UML usage by Dobing and Parsons [DP05] found almost 50% UML users in the questioned group of system developers. Even though this exact number is not proven to be reliable, it still indicates a high degree of UML dissemination in the relevant target group.

⁸although all of these help in actually understanding its details.

What then can we postulate as "understandable" for the Advanced Language User group?

- ◆ *Visual specifications* are preferred over textual ones. Users of Visual Modeling techniques can be assumed to have a preference for visual notations, unless textual alternatives offer clear benefits.
- ◆ As modeling languages are used in the field of Software Engineering, solid familiarity with the concepts of *Object-Orientation* can be assumed. Special formal calculi will, however, only ever be known to a particular subset of the audience.
- ◆ *UML familiarity* can also be assumed, even if the language under consideration is not UML.

All of these assumptions strongly indicate that a meta modeling approach (like employed for the definition of the abstract syntax of the UML) would have a high degree of understandability for a semantics specification, too (an idea also emphasized in [KER99, KW01, EFLR99, Reg02]).

Natural language is also very flexible in the way semantic concepts can be introduced. In some places, the UML specification text is rather descriptive, in others highly operational. Overall, it tries to express the semantics in a way which is *adequate* in that the mapping of syntactic to semantic concepts remains relatively straightforward. Any semantics description techniques which try to reduce the high-level features of UML to very basic mathematic formalisms will face a huge *semantic gap* and will probably not be adequate (cf. [Sel04]).

The flexibility of natural language has another effect: It can address all characteristics of Visual Modeling Languages (cf. Subsect. II.2.2), i.e., static and dynamic diagrams, high and low abstraction levels, semantic variation points etc. It is *universal* and can even cover very informal concepts like Use Cases. While this level of universality can never be reached with a formal specification technique, such a technique should be aimed at a high degree of universality and should be able to address (almost) all of UML's characteristics and features.

Requirements for a VML's semantics specification technique:

To actually find a suitable alternative to the current style of specification, we look for a **formal** and **precise** description technique which has a **high understandability** for Advanced Language Users. It also needs to be **analyzable**, **adequate**, and **universal**.

These requirements will guide our evaluation of existing approaches (in the next section) as well as our construction of the DMM approach.

II.3 Survey of Semantics Description Techniques for Visual Modeling Languages

Surveying the body of existing proposals to provide a formal semantics to UML is not an easy task as witnessed by [HR00]:

It is very difficult to compare papers written on the semantics of the UML, since the comparison must take into account the subsets of the notation dealt with, the assumptions on the kind of systems it is intended for, the relationships between the constructs treated, the levels of detail used in defining the language, and the notations and representations used in the papers themselves.

A basic distinction can be made between approaches which pursue specific goals (e.g., verifying model properties, analyzing consistency, or generating code) by providing a formalized semantics and approaches which aim for a general semantics description technique. The former are usually focused on their respective goals and techniques and can not be expected to yield universal solutions. We give an overview comparison of such approaches in Subject. II.3.1.

Approaches with the explicit aim to provide a general semantics description technique for (visual) languages are discussed in Subsections II.3.2 and II.3.3. We orient the discussion along the classical division between denotational (or compilation) semantics and operational (interpreter) semantics. In Subsection II.3.4 we present a hybrid operational/denotational approach.

II.3.1 Overview of Specific Formalizations

There is a vast difference between providing one specific formalization of (a sub-part of) UML and designing a technique which can universally address UML's semantics definition. A lot of papers pursue the former goal. Bran Selic calls these approaches "concrete semantics" in [Sel04]. We adopt this term. In particular we subsume approaches under this heading which

- ◆ are targeted toward a specific *application purpose*. This specific purpose usually guides the formalization process in a way as to obtain a usable formalization (for the application). If, e.g., model checking is the purpose then the formalization will be restricted in a way as to avoid too fast state space expansion.
- ◆ have a well known formal technique as their semantic domain. Targeting such existing formal notions allows for the intended application purpose as tools or theoretic results are readily available. The readability of the formalism in the semantic domain is usually of no concern for the authors and generally very poor with respect to Advanced Language Users of the UML.
- ◆ focus more on the semantic domain than on the mapping. As in many cases the semantic gap is rather small (the semantic domain and the examples being chosen to fit rather intuitively) no special effort is spent on the

formulation of the mapping. Usually informal descriptions are provided only.

- ◆ target specific notations of the UML only. To avoid the problem of a wide semantic gap, many approaches look for the UML notation which looks most like a nail to the hammer they have (or vice versa). Broadening the scope to other UML notations is usually dubbed a topic of "future work". Such approaches severely lack the universality required for a general semantics definition.

While concrete semantics may provide valuable insights to their specific purposes, they are far from meeting the requirements we set out. Especially understandability, adequacy, and universality are usually not addressed in these works. We do thus only provide a summarizing overview of concrete semantics approaches in Table II.3. For each approach we list its application purpose (if no explicit purpose is given, we state the purpose as 'formalization'), the technique of its semantic domain, the technique employed to describe the semantic mapping, and the UML diagrams covered. Please note that this coverage does not imply complete support for all features of the respective diagram. In fact, most approaches place severe restrictions on the general expressiveness of UML.

Similar to the theoretically oriented concrete semantics there are a number of approaches which generate program code from UML models. While most commercial case tools restrict the code generation to static structures (i.e., mostly class definitions), ideas for generation of more complex program code exist (e.g., [fuj, EHSW99]). These can also be classified as semantics descriptions, albeit with serious shortcomings in relation to our requirements: As the target language is a special programming language, the implementation independence of UML gets lost by definition. Abstract models cannot be expressed, multiple possible semantics are reduced to a single deterministic execution. Understandability heavily depends on the user's knowledge about the target language. We thus not cover code-generating approaches in depth here.

We can summarize that despite the number of concrete semantics proposed in the literature, none of them promises to fulfill the requirements we have. The presented works provide valuable insights for model analysis, consistency checking etc. But their formalization of UML is focused upon this purpose only. UML is a very general and high-level language, incorporating semantic concepts from very different schools of thought. Expressing its semantics in a concrete formalism which is restricted (for analysis purposes) or geared toward one specific paradigm only must fall short.

It is our opinion that the problem of actually defining the semantics of UML is complex enough to warrant its own technique. Based upon such a general semantics definition, one may then progress to derive (in a formal and documented way) more restricted semantics representations, allowing for the application of specific analysis methods.

AUTHOR(S)	PURPOSE	SEMANTIC DOMAIN	SEMANTIC PING	MAP-	UML COVERAGE
Bock and Gruninger [BG04, BG05]	formalization	Process Specification Language	description/examples		Activity Diagrams
Böger, Cavarra, Riccobene [BCR00, BCR03, CRS04]	formalization/simulation	Abstract State Machines (ASMs)	description		Activity Diagrams, Statecharts
Damm et al. [DJPV03]	formalization	Symbolic Transition Systems	pre-compilation and formal logic		Real Time Statecharts, Class Diagrams
Di Nitto et al. [NLS ⁺ 02]	workflow execution	OPSS process specifications	description		Activity Diagrams, Class Diagrams, Statecharts
Eshuis and Wieringa [EW04, Esh02, EW01]	verification	Clocked Transition System	description		Activity Diagrams
Kim and Carrington [KC00b, KC00c, KC99]	formalization	Object-Z and Timed Refinement Calculus	Formal mapping [KC00a]		Class Diagrams, Statecharts
Knapp [Kna99] and Störle [Stö03]	formalization, detection of specification flaws	Traces and Temporal Logic	Description		Collaboration/Interaction Diagrams
Knapp et al. [KMR02]	consistency checking	Timed Automata	description		Statecharts, Collaboration Diagrams

(continued)

AUTHOR(S)	PURPOSE	SEMANTIC DOMAIN	SEMANTIC MAPPING	UML COVERAGE
Küster et al. [Küs04, EHK01]	consistency checking	CSP	rule-based transformation	Statecharts, Sequence Diagrams
Kwon [Kwo00]	verification	SMV input language (finite state machines)	formal mapping	Statecharts
Lano and Bicarregui [LB99]	formalization, consistency	Real-time Action Logic	description	Class Diagrams, Sequence Diagrams, Statecharts
Li et al. [LLH04, LLH02]	consistency checking	CSP	description	Class Diagrams, Sequence Diagrams, Use Case Diagrams
Ober [Obe03]	formalization	Abstract State Machines (ASMs)	translation tool	Class Diagrams, Actions
Øvergaard [Øve99]	formalization	Traces and logic	description	Collaboration Diagrams
Øvergaard [Øve00]	formalization	BOOM (textual specification framework, based on the π -calculus)	meta model mappings [Øve98]	Class Diagrams
Paige et al. [POB02]	consistency checking	PVS specifications (The Prototype Verification System, a theorem prover)	description	Collaboration Diagram, Class Diagram
Pezzè and Baresi [BP01a, BP01b]	analysis	High Level Times Petri Nets	description and reduction rules	Class Diagrams, Statecharts

(continued)

AUTHOR(S)	PURPOSE	SEMANTIC DOMAIN	SEMANTIC PING	MAP-	UML COVERAGE
Porres and Lilius [LPP99, PL99, Por01]	verification	PROMELA (input language for the SPIN model-checker)	vUML tool		Class Diagrams, Statecharts
Reggio et al. [RACH00, RACH99, AR02]	formalization, consistency checking	CASL (algebraic specification framework) and LTSs	description		Class Diagrams, Statecharts
Stevens [Ste01]	formalization	Labeled Transition Systems	description		Use Cases
Störle [Stö05a, SH05, Stö04c, Stö04a]	formalization	Petri Nets (various dialects)	description		Activity Diagrams
van der Beeck [vdB01, vdB02]	formalization	Structured Operational Semantics rules	-not applicable-		Statecharts
Van der Straeten et al. [SJM04]	consistency	traces and LTS	description		Statecharts, Sequence Diagrams
Varró [Var02, Var03]	verification	Extended Hierarchical Automata	Model Transition Systems		Statecharts
Xie et al. [XLB01, XB02]	verification	S/R automata (input language for the COSPAN model checker)	tool		Statecharts

Table II.3: Overview of concrete semantics descriptions for UML

Besides these concrete semantics formalization approaches there are also more fundamental and general proposals for semantics definition techniques. We discuss these in the following subsections, starting with the denotational paradigm.

II.3.2 Denotational or Compilation Semantics Descriptions

The approach of *denotational semantics* originated in the field of Programming Languages. Its foundations were formulated by Scott and Strachey [SS71] in the early 70s. The denotational approach has the following characteristics [Mos03]:

- ◆ Clear separation of syntactic domain, semantic domain, and semantic mapping.
- ◆ The semantic domain is formed by mathematical functions
- ◆ The mapping is defined inductively, using λ -notation to specify how the denotations of components are to be combined.

Denotational semantics allow for a very rigorous semantics definition in terms of commonly agreed mathematics but they also require considerable mathematical skills. For instance, loops and recursion are usually denoted to least fixed-points of continuous functions on Scott-domains. In their pure form denotational semantics thus remain very much a theoretician's tool as they severely lack understandability for a broad audience.

The clear underlying framework and its terminology have, however, shaped the way many people think about semantics. We have also adopted this terminology in our discussion of concrete semantics in the previous section. Since the approaches discussed there (and in the remainder of this section) lack the mathematical rigor in both their semantic domain and the mapping definition, they are not really denotational in the original sense of the approach. One might rather call them "translation" or "compilation" semantics.

Compilation semantics thus consists of translating expressions in a language which is to be defined (the syntactic domain) into a language which one assumes is already known (the semantic domain). How well a compilation semantics fulfills our requirements mainly depends on its choice of formalism for the semantic domain (the overview of concrete semantics in the previous subsection already dismissed a number of such choices). In the following paragraphs we focus on three promising approaches for formulating a semantic domain suitable for Visual Modeling Languages:

First, the expression of a model's semantics in terms of *graph transformations* is discussed. Being a well-known and visual formalism, graph transformations have the potential to form a highly understandable semantic domain, thus meeting our prime requirement. Then we briefly touch upon the topic of *core semantics*, which promotes the idea of the semantic domain being a subset (or core) of the original language. Finally we discuss the so called *denotational meta modeling* which is the currently most influential style of semantics definition for the UML.

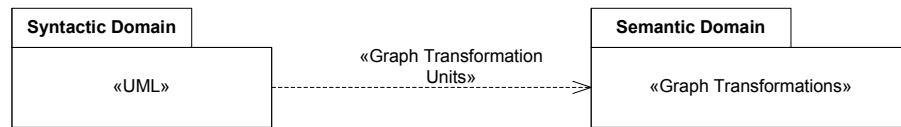


Figure II.4: Architecture of the Bremen approach to define UML semantics

Graph Transformations as a Semantic Domain

Graph Transformations are a visual formalism which allows to specify the manipulation of object structures in a very intuitive way. The mathematical foundations of Graph Transformations, their features, and numerous examples are presented in Chapter IV, see also [Roz97, EEKR99, EKMR99].

The combination of visual renderings with potentially intuitive interpretations and mathematical rigor make Graph Transformations an interesting candidate for behavioral specifications. This general fact has long been recognized, cf. [Pad82, EEKR99, GR01].

One approach which exploits this general idea for the semantics description of UML is that of Kuske, Gogolla and Ziemann (illustrated in Fig. II.4 and called the Bremen approach in the following discussion). In [Kus01] the semantics of a Statechart model is expressed by a set of graph transformation rules. Since both formalisms are based on the concept of states and discrete changes of state, the semantic gap is rather small in the basic case. To correctly translate the high-level features of UML Statecharts (nested states, priorities, etc.) Kuske employs Graph Transformations controlled by Transformation Units [Kus00]. Thus both the semantic domain and the semantic mapping are expressed by Graph Transformations. The approach has later been generalized to cover other UML diagrams and their integration [KGKK02, GZK02, ZHG05].

In its general form, the approach takes a model composed of different UML diagrams as input and produces a set of Graph Transformation rules as output. This process is illustrated by an example taken from [GZK02] and depicted in Figs. II.5 and II.6. The figure shows a Class Diagrams and a Statechart which specify aspects of an office model. Fig. II.6 shows a rule which is generated by the approach. It combines the information of the different diagrams in that it specifies that executing the operation `printer.print()`, a new `Printout` object is being created and an event for triggering proofreading is enqueued with the `Boss` object.

An advantage of using Graph Transformations as the semantic domain is that given a start graph for the model a whole behavior sequence can be played out to the user by applying the generated rules [EHKZ05, EB04]. Such animations come pretty close to our idea of example model validation. Open questions are how non-determinism and incompleteness can adequately be represented in such approaches.

The Bremen approach also comprises several drawbacks. Its most fundamental problem is the simplicity of the Graph Transformation approach employed for the formulation for the semantic domain. Since only simple graph trans-

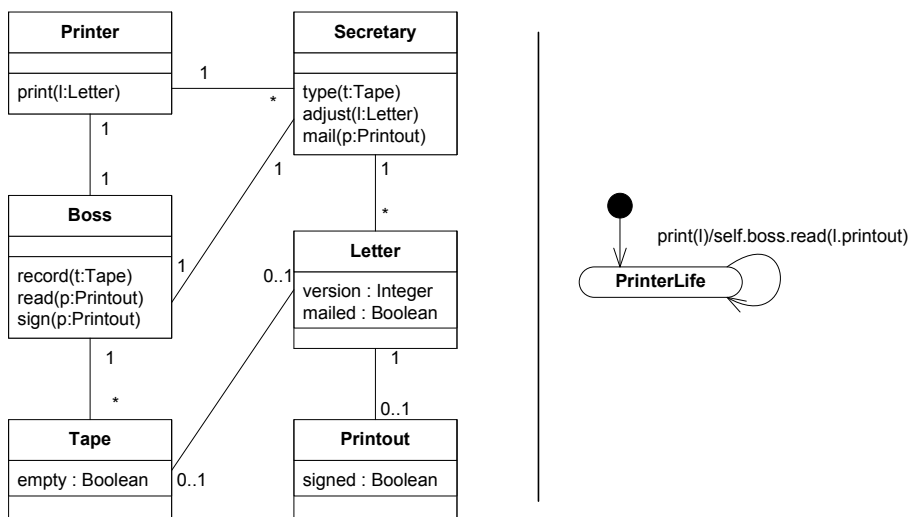


Figure II.5: Example for the Bremen approach: Class Diagram and Statechart

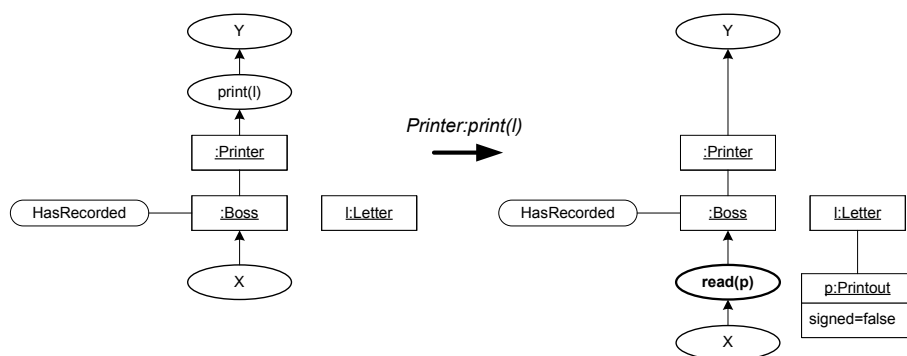


Figure II.6: Example for the Bremen approach: Resulting rule for operation print

formation rules are used here, all manipulations for a certain behavior have to be either combined in a single rule or synchronized by means of control objects (as done in [ZHG05]). This causes some rather peculiar interpretations of UML models (e.g., all operations have to happen in an atomic step)—a sign for inadequacy of the approach.

On the other hand the simplicity of the semantic domain enforces a complex translation mechanism to encode the high-level features of UML. The approach of employing Graph Transformation Units to capture these translations (as used in [Kus01]) does not find any mention in later publications [ZHG05]. These just speak of an "automated translation" being based on the USE system [RG00]. The translation description, however, is where the language semantics really reside in this approach. The resulting Graph Transformation System provides the semantics for a *single model* only (note, how the rule in Fig. II.6 is formulated over concrete model elements like **Boss** and **Printer**). Understanding the general mechanisms of a language and possibly extending them is prohibited in this approach by obscuring the translation specification. Without a clear mapping notion the approach is also not precise.

We can conclude that Graph Transformations seem a promising choice for the formulation of the semantic domain especially for dynamic semantics but the way they are employed in the Bremen approach does not meet the criterion of adequacy. The semantic mapping in that approach also lacks precision and understandability.

Core Semantics

Core Semantics approach the dilemma of finding an appropriate semantic domain by defining a subset of the syntactic domain (i.e., the language to be explained) as the *core language*. The semantics mapping can then be considered as a reduction from the whole language to the language core (cf. Fig. II.7). Core semantics approaches for the definition of UML can be found in [GPP98] (reducing UML Statecharts to simple automata) and [GR99] (reducing complex to simple Class Diagrams). Both of these works use Graph Transformations to describe the reduction. The general notion of defining a language core for UML (without an explicit reduction) is often proposed (e.g., [SK02, EK99, ESW⁺05]) with Class Diagrams and Statecharts being the favorite candidates to express structure and behavior respectively [HG97, HK04].

The inherent problem of Core Semantics is that a subset of the (as of yet semantically undefined) language is assumed to have a clear and well-known semantics. While this is just a variation on the general assumption of compilation semantics, the circularity of Core Semantics makes it harder to accept to many people. Thus, such reductions are usually regarded as pre-compilations only, reducing the "syntactic sugar" before attacking the core language with a "real" semantics description⁹.

⁹The reverse way is known as *bootstrapping* in Programming Language design: First, a compiler for the core language is implemented in an external language, then this core language

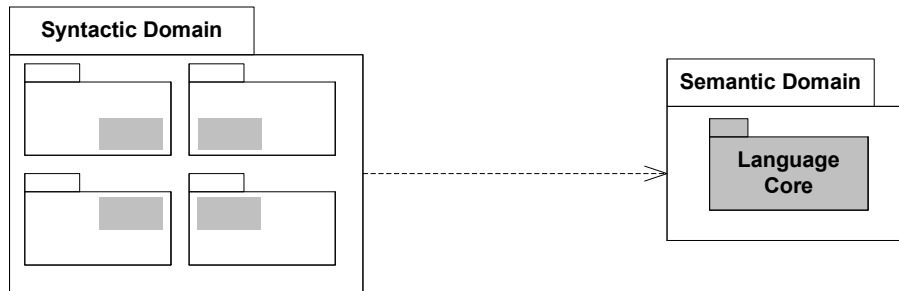


Figure II.7: Overview of the Core Semantics approach

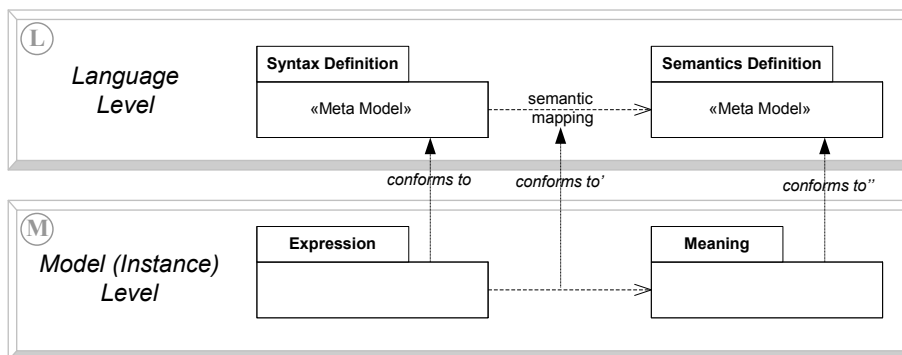


Figure II.8: The architecture of denotational meta modeling

Besides this problem, core semantics promise a high degree of understandability by essentially applying the meta modeling idea to the definition of semantics. We deem this to be a very interesting concept, yet the above cited works only cover very selective features of UML and do not allow for an evaluation of adequacy and universality of this kind of semantics definition.

Denotational Meta Modeling

Denotational Meta Modeling is a term coined by the members of the pUML group which actively promoted this approach to semantics definition for Visual Modeling Languages [KGR99, KER99, EK99, CEK⁺00, CEK01]. Its basic construction is outlined in Fig. II.8. The denotational meta modeling approach assumes the language under consideration to be formulated by a (syntactic) meta model. This meta model defines the set of expressions of the language (models). The approach then proposes to formulate the semantic domain in a similar structure, using a *semantic domain meta model* to define the set of semantic concepts which are relevant for the expression of a model's meaning. The semantic mapping is defined on the meta model level, relating syntactic to semantic elements. This mapping is then propagated to the instance level and allows for a precise construction of a given model's semantics.

is used to define the compilation of additional language features.

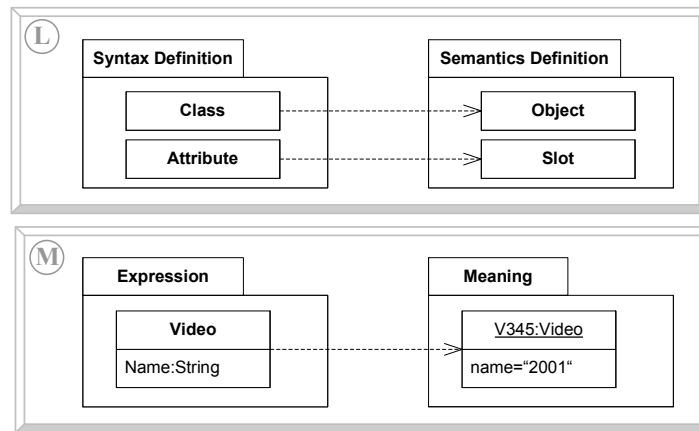


Figure II.9: An example for denotational meta modeling

A common example for the denotational meta modeling approach can be found in Fig. II.9. Here, the syntactic element `Class` is related to the semantic concept `Object`, the concept of `Attribute` to that of a `Slot` (which can hold actual values). On the model level we find a class modeling the concept `Video` (the example is identical to that of Fig. II.3) which expresses properties about a set of objects (a sample object is shown in the lower right hand package of the figure).

Note that the choice of semantic concepts in the denotational meta modeling framework usually corresponds to "runtime notions", i.e., terms used to describe the execution of programs. We thus find concepts like `Object`, `SignalOccurrence`, and `ActionExecution` in the semantic domain meta model. For users with a background in OO programming, these notions are well-known which increases the understandability of the denotational meta modeling approach. There are, however, no strict requirements for this choice of semantic concepts.

If runtime notions are chosen as the semantic concepts, one needs to take care in using the term 'instance' in the denotational meta modeling framework. On the one hand, language expressions can be seen as instances of their defining meta model (i.e., the vertical relations in Figs. II.8 and II.9). In [Kue05] this kind of instance is called a *linguistic instance*. On the other hand, the semantic mapping relates defining model elements with runtime or instance notations (the horizontal relations in the figures).

A main achievement of the denotational meta modeling approach is the clear separation between the type or language level (which defines the semantic mapping in general) and the instance or model level (which assigns a concrete meaning to one specific expression). While even mathematical denotational semantics provide their definitions in terms of general syntactic entities, the distinction between type and instance level is not made so explicit there. We will emphasize this distinction by using the notation of the slightly raised boxes to distinguish meta levels throughout this thesis. The language definition level is marked with an 'L', the instance (model) level with an 'M'.

Another advantage is that—compared to the concrete semantics approaches

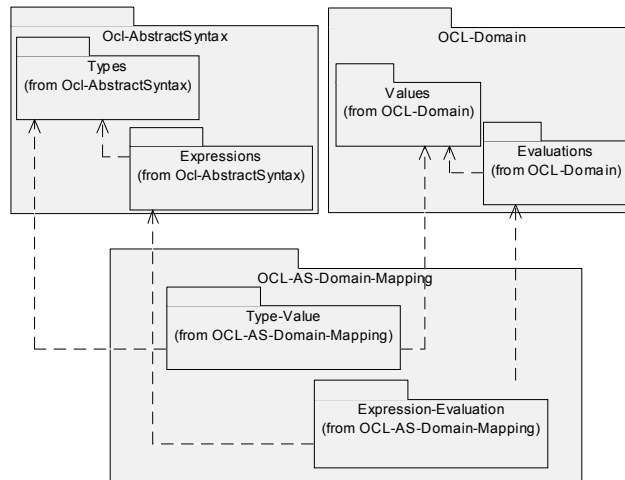


Figure II.10: Architecture of the semantics specification of OCL 2.0 (reproduced from [Obj03c])

in Subsect. II.3.1—denotational meta modeling does not prescribe a specific semantic domain, i.e., it provides no concrete constructs on which the syntax elements have to be mapped. The Language Engineer can actually construct a set of semantic concepts he deems adequate for the description of a language. Denotational meta modeling only provides the technique to express his choices. This enables an expression of semantics using concepts known to the intended users. The concept of meta modeling is also known to this group from the syntax definition of the UML. We can thus award high scores for potential understandability.

As the pUML group formed the academic core of the U2Partners consortium which contributed the basis for the UML 2.0 specification, the OMG has started to follow the approach of denotational meta modeling. The most notable result is the OCL 2.0 specification [Obj03c] (see Fig. II.10 for its architectural overview) which uses denotational meta modeling. The denotational meta modeling approach is currently the closest to being considered for a formal specification of UML's semantics.

Even if the UML specification does not (yet) directly employ denotational meta modeling, it benefited from the pUML group's clarification of terms. Direct results can be seen in the fact that `Instance` is not a class in the UML 2.0 meta model anymore (because it really is a semantic concept) and that the Action Semantics of UML 1.5 have been renamed to Actions (because they are syntactic elements only).

Denotational meta modeling also incorporates two drawbacks: One is of a rather technical nature, the other a severe conceptual issue.

Technically, the realization of the mapping between syntactic and semantic domain has to be considered an open problem. In the OCL 2 specification, special mapping classes were used (grouped in the package `OCL-AS-Domain-Mapping` in Fig. II.10) which were associated to the syntax and semantics elements. The

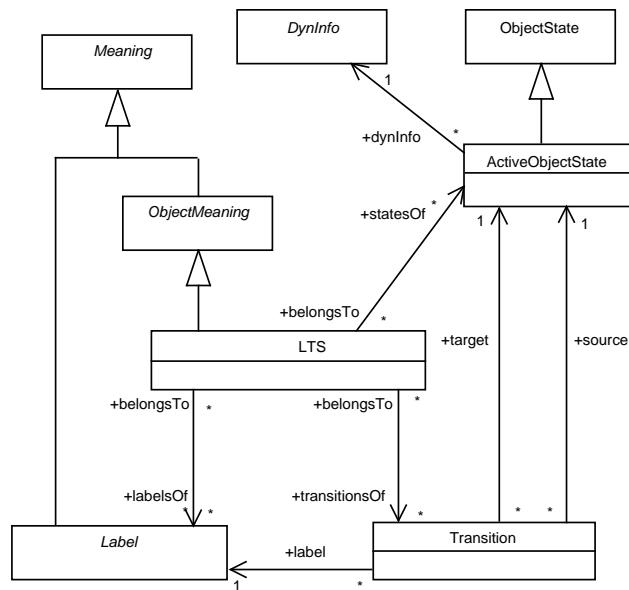


Figure II.11: Semantic domain meta model of the Reggio/Astesiano study: LTS and Active Object (reproduced from [RA01])

main drawback here is that both domains get "lumped together" by using associations. In fact the package membership is all that distinguishes syntactic, semantic, and mapping elements.

Conceptually the biggest drawback of denotational meta modeling is its inability to express behavior in a proper way. In the definition of the Meta Modeling Language (MML, the first substantial subset of UML to be defined according to denotational meta modeling [CEK⁺00]) dynamic aspects are mentioned only briefly. Two studies on the integration of dynamic semantics in MML have been published: In [RA01] Reggio and Astesiano propose a concentration of behavior notions to the (syntactic) element **Active Class**. The semantic domain meta model for this element provides the notions of **Active Object** and **LTS** (Labeled Transition System) (see Fig. II.11). The semantics mapping specifies that the states of the LTS are the system states of the Active Object and that the transitions of the LTS correspond to communications of the active class. No more details are provided on the semantics mapping of this dynamic core. The notion of active class is furthermore not related to UML's usual behavioral features (i.e., Statecharts etc.). It thus remains unclear how this dynamic core ties into a more general framework.

Kleppe and Warner [KW01] criticize the Reggio/Astesiano study furthermore for not properly integrating the dynamic and the static features. In their opinion, "a static view is just a view at one moment in time of a dynamic view, thus the semantics of UML must be build from a viewpoint that integrates static and dynamic aspects". They proceed to provide a semantic domain meta model which is based upon the notion of **Values** which are aggregated to **Snapshots** (cf. Fig. II.12). Fundamental behavioral mechanisms of UML (actions and signals)

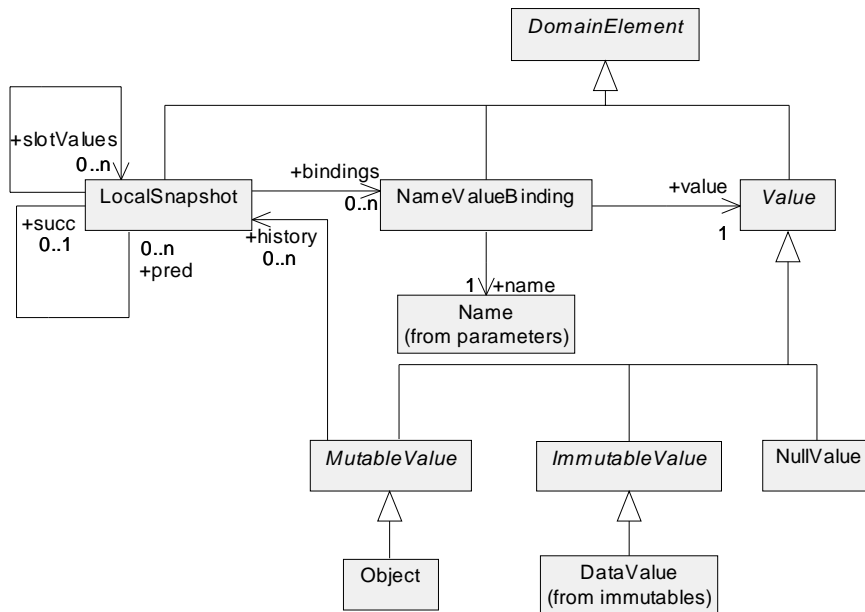


Figure II.12: Semantic domain meta model of the Kleppe/Warmer study: Values and Snapshots (reproduced from [KW01])

are related to this dynamic core. Fig. II.13 provides an excerpt of the package specifying the semantics of actions (`actionExpressions.instance.concepts`). The semantic equivalent of an action expression is the class `ActionExpExec` which relates a `before` to an `after` snapshot. It may furthermore produce a `Value` as its result. While this is a correct description of an action execution from a structural point of view, it simultaneously falls woefully short to explain anything *about* the actual behavior (i.e., "What does the action do?"). Actually this information has to be encoded in very complex OCL constraints on the semantic mapping. Being complex already for (basic) actions, the study falls short to lay out the semantics of the more high-level behavioral diagrams by reducing state machines to a simple signaling mechanism and claiming activity and collaborations diagrams to be "*straightforward*".

Despite all its successes and obvious advantages, the restriction of denotational meta modeling to specifying behavior in a descriptive way only is a fundamental drawback. It makes for complicated specifications and shifts the emphasis from the semantic domain to the semantic mapping and thus to complex logic formulae (in the form of OCL). We do recognize that denotational meta modeling goes a long way to the satisfaction of our requirements but it is not yet an ideal solution.

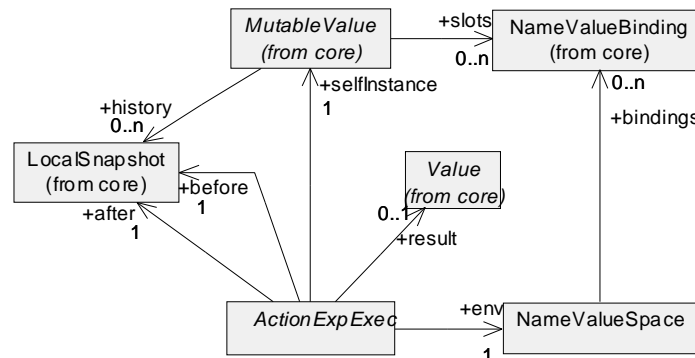


Figure II.13: Semantic domain meta model of the Kleppe/Warmer study: Actions (excerpt, reproduced from [KW01])

II.3.3 Operational or Interpretation Semantics Descriptions

Operational Semantics are also a general style of semantics descriptions and form an alternative to denotational semantics. While denotational semantics approach semantics definitions by stating what an element *is* (in terms of the semantic domain), Operational Semantics describe what an element *does*. Operational Semantics are given in terms of *rules*. Each rule consists of *preconditions* which have to be met for the rule to apply and *effects* which transform the current state in some way. There is a number of different operational specification frameworks available (see [Mos01, Mos03] for an overview). The most prominent approach is Structured Operational Semantics (SOS), proposed by Plotkin [Plo81]. The history of the SOS approach and its connections to other styles of semantics definitions are elaborated in [Plo04].

[Mos01] provides the following characterization of Operational Semantics:

Computations are modeled as sequences of (possibly labeled) transitions between states involving syntax, computed values, and auxiliary structures.

Figure II.14 provides an abstract representation of the idea of Operational Semantics. It is a bit unusual (compared to illustrations usually found) in that it tries to place the ideas of Operational Semantics in the structure imposed by denotational semantics. We can see that the notion of state in Operational Semantics consists of syntactic elements, values, and auxiliary elements. The semantic mapping on the language level is thus a simple identity function between the syntactic elements. The core of an operational semantics are its rules which specify transformations of states. On the instance level, an expression determines a start state (the state solely constructed from its syntactical elements). Applying the rules of the Operational Semantics yields new states which gradually extend and finally replace the syntactic structures by auxiliary constructs and values. The final states of this transition system only contain values, they represent the result(s) of the specification. This type of transition system is called a Labeled Transition System (LTS). The semantics of a program are thus

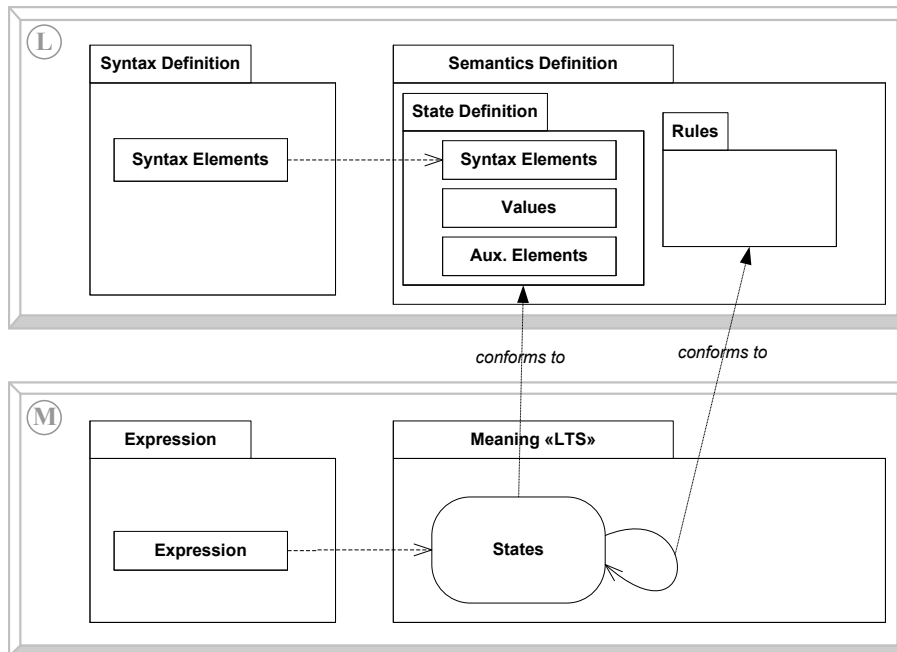


Figure II.14: Overview of the Operational Semantics approach

$$(1) \frac{E \longrightarrow E'}{cond(E, E_1, E_2) \longrightarrow cond(E', E_1, E_2)}$$

$$(2) \quad cond(true, E_1, E_2) \longrightarrow E_1$$

$$(3) \quad cond(false, E_1, E_2) \longrightarrow E_2$$

Figure II.15: Operational Semantics: Example rules

the LTS it induces.

A typical example for SOS rules is provided in Fig. II.15. These three rules evaluate a conditional expression which upon the boolean value of E either executes E_1 or E_2 (i.e., in usual pseudo-code: `if E then E1 else E2`). The rules illustrate the step-wise fashion in which operational semantics alter the underlying state. Rule (1) is a conditional rule which states that the results of other rules (for evaluating boolean expressions) can be propagated to the conditional construct. Once these evaluations yield a final boolean value, rules (2) or (3) may apply and either reduce the conditional expression to its ‘then’ part (E_1) or its ‘else’ part (E_2). Note that the lower two rules delete all obsolete syntactic elements.

As the user can “observe“ the steps of evaluation/computation in the LTS, operational semantics are also said to specify an abstract *interpreter* for a given language. Such interpreter semantics have an advantage in understandability as

$$\begin{aligned}
OR-1 & \frac{(-, l, sr, e, \alpha, td, i, ht) \in T, sr \subseteq \text{conf}(s_l), s_l \xrightarrow{e} s'_l}{[n, (s_{1..k}), l, T] \xrightarrow{e} [n, (s_{1..k})_{[s_i/\text{next}(ht, td, s_i)]}, i, T]} \left(\begin{array}{l} ex \in \text{exit}(s_l), \\ en \in \text{entry}(\text{next}(ht, td, s_i)) \end{array} \right) \\
OR-2 & \frac{s_l \xrightarrow{e} s'_l}{[n, (s_{1..k}), l, T] \xrightarrow{e} [n, (s_{1..k})_{[s_l/s'_l]}, l, T]} \\
OR-3 & \frac{s_l \xrightarrow{e} s_l, [n, (s_{1..k}), l, T] \xrightarrow{e} [n, (s_{1..k}), l, T]}{[n, (s_{1..k}), l, T] \xrightarrow{e} [n, (s_{1..k}), l, T]}
\end{aligned}$$

Figure II.16: Formalizing UML Statecharts by SOS rules, examples reproduced from [vdB02]

users can follow complex computations in a stepwise fashion. Operational Semantics are thus usually regarded as more intuitive than denotational semantics for the expression of behavior.

Operational Semantics also provide a firm base for reasoning about programs: As the rules work along the structure of the specification, which in programming languages is a tree, proofs can be formulated inductively. Induction works over the structure (a property holds for all abstract syntax trees) or (in rarer cases) over the rules themselves (a property holds for all transitions).

Applying Operational Semantics to Visual Modeling Languages poses a number of challenges:

Trees vs. Graphs. Models do not generally have a tree as their underlying structure. The technique of structural induction does thus not work as its recursions are not guaranteed to terminate.

Missing start states. Models do not necessarily provide a start state. While Programming Languages provide mechanisms to determine the start of their computation (e.g., in a `main` procedure), models usually have no such deterministic starting point.

Visual expression of semantics. The visual nature of models is not adequately reflected in SOS rules. Take, e.g., van der Beeck's formalization of Statecharts in SOS as an example ([vdB02], see also the reproduced example rules in Fig. II.16). While it provides the semantics of Statecharts in a very precise way, its rules are purely textual in nature and can not be considered as understandable from our point of view. A remedy is proposed by Corradini et al. [CHM00], who provide a graphical representation of Operational Semantics using Graph Transformation rules as a basis. Their approach is called Graphical Operational Semantics (GOS).

Incompleteness of models. Models can be incomplete. SOS usually assumes that the information given by a program is sufficient to derive its final value(s). Since this does not necessarily hold for models, GOS [CHM00] provides two ways of defining the relation between an extension E (a LTS) and an operational specification S : E satisfies S , if it is closed under derivable transitions, i.e., transitions which correspond to rules of S must obey these rules. This notion allows for arbitrary steps in E outside of the

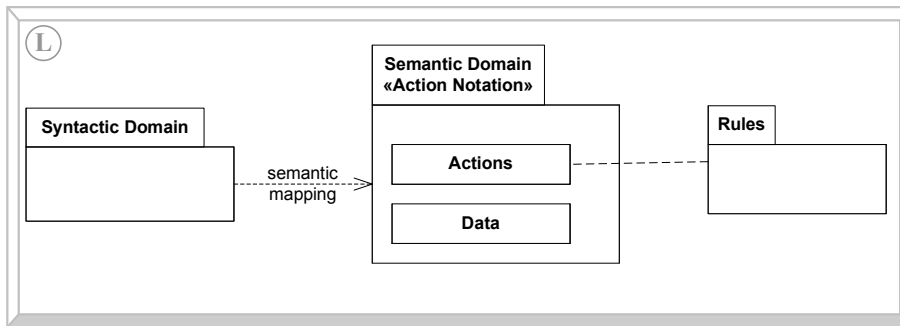


Figure II.17: Overview of the Action Semantics approach

specification's scope. On the other hand E is *generated* from S if all transitions in E correspond to rules in S .

Extensibility of the specification. Extensibility is also an issue for Operational Semantics. The introduction of new concepts often causes the reformulation of large parts of a rule set [Mos00].

Representation of data structures. A large part of the UML is devoted to modeling complex data structures. Representation of such data structures (i.e., everything beyond simple values) is difficult in Operational Semantics. An example for a possible integration of types into an Operational Semantics framework is given in Chapter 8 of [Mos03]. Types are treated as tuples of values here and rules dealing with types are restricted to side-effect free big-step rules of the Modular Operational Semantics style [Mos04b].

We can thus summarize that the interpretational style of operational semantics represents behavior in a step-wise fashion which is intuitive for many people. In that it is clearly superior to the denotational approach. For application to Visual Modeling Languages, the development of Graphical Operational Semantics is most relevant as it already addresses some issues stemming from the application of operational semantics to VMLs. Other questions remain open, however.

The hybrid approach of Action Semantics tries to combine the different strengths of operational and denotational approaches. We discuss Action Semantics in the next subsection.

II.3.4 Hybrid Semantics Descriptions

As denotational approaches are very successful in many regards but proved to be very unsatisfactory for anything but purely functional behavior, attempts were made to combine denotational semantics with operational semantics. One result is the Action Semantics approach as defined by Mosses [Mos92].

The basic idea of Action Semantics is visualized in Fig. II.17. Syntactic elements are mapped (denotational style) to a semantic domain which is formulated in a language called Action Notation. Action Notation provides constructs to describe data and actions. Actions are predefined entities which express behavior in a number of so called *facets* (control flow, data flow, data processing, communication etc.). The behavior of these actions is defined using Structural Operational Semantics rules. The denotation thus expresses that a certain language element has a behavior (by relating it to an action or possibly a combination thereof), the operational part of the specification expresses what this behavior actually does. Note that in the figure we set the rules apart from the semantic domain as they form the semantics of the Action Notation and not directly the semantics of the syntactic domain. See [Mos96] for an extensive overview of Action Semantics, including examples and practical applications, e.g., for automated compiler generation.

An application of Action Semantics to Visual Modeling Languages has not yet been undertaken. There was, however, a usage of the term 'Action Semantics' in some versions of the UML specification [Obj03e]. While there is no direct relation between Mosses' Action Semantics and the UML's Action Semantics, some similarities can be observed [Mos04a]. In the UML, so called *Actions* were pre-defined to express atomic behavior. In that they were rather similar to Mosses' Action Notation. The UML Actions did, however, neither provide a concrete notation nor a formal semantics. In the move to UML 2 the term 'Action Semantics' was consequently reduced to 'Actions' only.

II.3.5 Conclusions from the Survey

We can conclude from our survey that no existing approach to the definition of a Visual Modeling Language's semantics completely meets our criteria. Especially the requirements of understandability and adequacy are hard to achieve. Yet, the survey presented a number of promising ideas which have complementary strengths and might thus be assembled in a way as to combine their advantages.

II.4 Concept of the Dynamic Meta Modeling Approach

The approach which achieved a high degree of understandability and promised the most potential for adequate semantics definitions was the denotational meta modeling approach. It could express static semantics in a precise, yet mostly visual way without constraining the semantic domain to a set of predefined elements. Its weak point was the inadequate expression of dynamic semantics. Action Semantics have shown a way how to overcome this inherent weakness of denotational approaches by combining them with operational semantics.

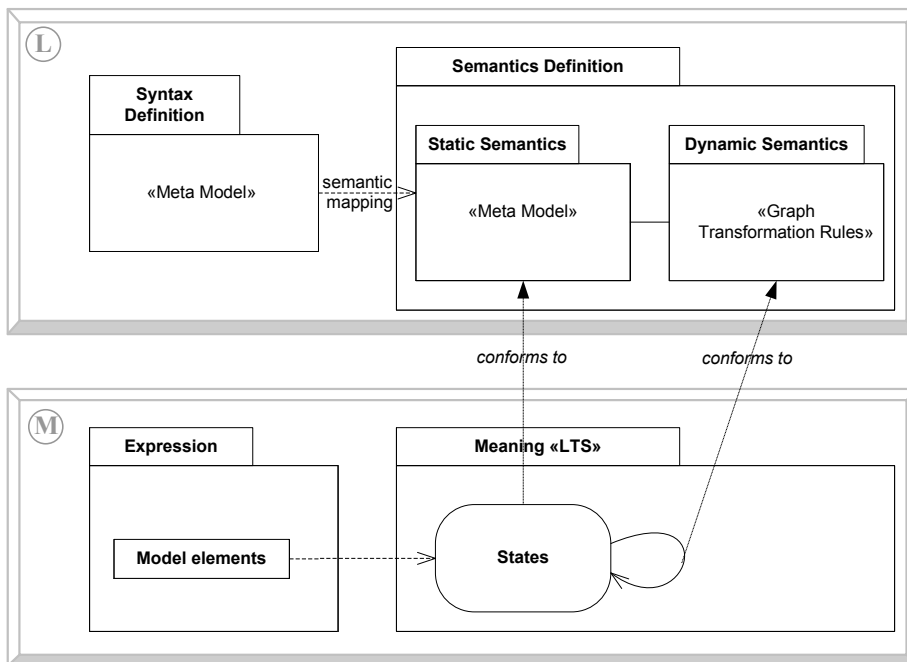


Figure II.18: Outline of the DMM approach

We follow this line of thought and provide a semantics description method for Visual Modeling Languages which combines a denotational meta modeling framework for expressing static semantics with operational rules capturing the behavior of elements. Fig. II.18 illustrates the outline of our approach. Since denotational meta modeling leaves the choice of adequate semantic elements to the Language Engineer, the operational semantics of these elements can not be predefined. Rather, a technique for their definition must be supplied which also conforms to our criteria.

The approach of Core Semantics propagates the use of UML diagrams for understandable specifications of behavioral semantics but it is plagued with the problems of circular definitions. Graph Transformations on the other hand are a formal technique which allows for the precise specification of behavior while promising good understandability. We believe that a combination of both approaches allows for a combination of their advantages: The specification of behavioral semantics should look like a UML diagram, thus exploiting the user's existing knowledge. Technically, however, the specifications should be Graph Transformations, thus having a precise semantics and avoiding all meta-circularity issues arising in Core Semantics and the MOF.

This combination of denotational meta modeling and operational graph transformation rules, presented as UML diagrams is the approach which we elaborate in this thesis. To emphasize its support for behavioral elements, our approach is called *Dynamic Meta Modeling* (DMM for short)¹⁰.

¹⁰The presentation of ideas in this chapter does not accurately reflect the historic development of the different approaches. In fact, DMM has been developed in parallel with most

Dynamic Meta Modeling promises a high degree of understandability in its semantics specifications with a formal and precise background. To actually judge whether DMM fulfills these promises and to evaluate it against the complete set of criteria we set out, we need to flesh out the approach with technical details first. In the following two chapters we describe the two main technical components of DMM: First, we turn to a technique to formalize the semantic mapping. Chapter III introduces a novel approach to specify such inter-meta model mappings called Meta Relations. Then we focus on Graph Transformations in Chapter IV. We explain the basic formalism, evaluate different variants of Graph Transformations and propose an innovative mechanism to provide the features which we require to combine adequacy, understandability, and analyzability for DMM.

above mentioned works toward the definition of a VML's semantics. Meetings and discussion at different scientific venues lead to a mutual pollination of ideas by good or bad examples.

Chapter III

Meta Relations

Denotational Semantics (see Subject. II.3.2) comprise the concept that elements of a syntactic domain are connected via a semantic mapping to elements of a semantic domain. In denotational meta modeling both of these domains are expressed by meta models (cf. Fig. III.1). There is, however, no standard mechanism defined in the Meta Object Facility (MOF) [Obj03a] to express such a mapping between meta models. Thus, to express a semantic mapping between, say, the UML meta model and some semantic domain meta model, one has to rely either on external mechanisms or on (ab)using associations. For various reasons (see Section III.1) we find these solutions unsatisfying.

In this chapter, *Meta Relations* are introduced to fill this gap. Meta Relations are a diagrammatic language which can be seamlessly integrated with the meta modeling framework of the MOF and which allows expressing connections between different meta models without actually merging them. Meta Relations are not only a useful concept for expressing denotational semantics. They can be applied to many other areas like code generation (cf. [HK03] for an example of a UML - Java mapping based on Meta Relations), reconciliation (cf. [HHS02b]), and MDA in general (as discussed in [Hau03]). Here we focus on their use in semantics definitions only.

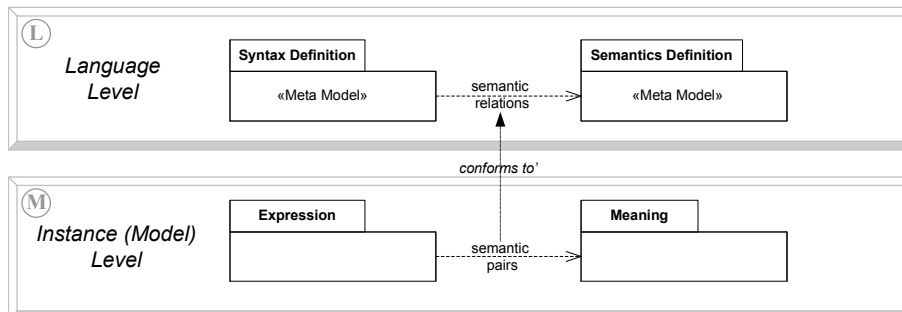


Figure III.1: Denotational Meta Modeling and the role of relations

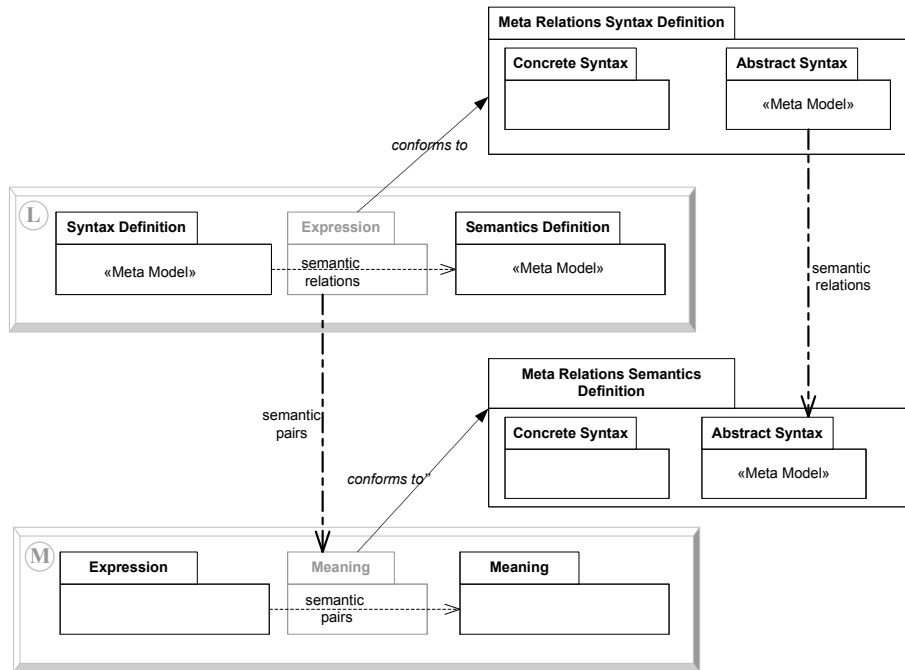


Figure III.2: Illustration of the connections between the definition of Meta Relations and their usage in language definition

One of the achievements of denotational meta modeling is the explicit distinction between the type and instance level in semantics definitions (see Fig. III.1). Consequently, what is usually called the *semantic mapping* also has an explicit definition part (connecting general language elements to general semantic entities) and an instance part (connecting the elements of one concrete language expression to its meaning). We call this instance level mappings *pairs* and the type level concepts *relations* in this thesis. If this explicit distinction is not required, we use the term *mapping*. The capitalized terms ‘Relation’ and ‘Pair’ are used to refer to our specific realization of the general concept of relations.

In the definition of Meta Relations it is important to separate the different levels of concepts which we are talking about. On the one hand, Meta Relations are a language with the usual distinction in syntax and semantics. On the other hand, expressions in this language are used in the definition of other languages. Making things even harder is the fact that Meta Relations are both used in and defined by a denotational meta modeling approach.

Fig. III.2 illustrates the different parts of the Meta Relation definition and their use for language definition. This usage (semantic mapping in the L-level of the language to be defined) is an expression in the language of Meta Relations (as indicated by the light package notation). To define these expressions, Meta Relations need a definition of concrete syntax (see Sect. III.3) and a meta model specifying their abstract syntax (provided in Sect. III.4).

Using Meta Relations to express a semantic mapping constrains the way se-

semantic pairs relate language expressions to meaning on the M-level. From the viewpoint of Meta Relations as a language, these semantic pairs are the meaning to its expressions, i.e., a concrete semantics. The general semantics of the language Meta Relations are provided in the denotational meta modeling style. Sect. III.5 provides the semantic domain meta model of Meta Relations. We also need to make explicit the semantic mapping between the syntactic and semantic meta model of Relations. Here, the Meta Relation definition encounters the usual circularity of meta modeling approaches as we define this mapping using Meta Relations themselves (in the figure the dashed and dotted arrow). In addition to this application of the general denotational meta modeling framework, we also supply a special concrete syntax for depicting the instances of a Relation's meaning.

Note that from the Meta Relations point of view, the language level of the language to be defined (i.e., the contents of the L box) is the instance level/expression side. Similarly the language's instance level (M box) forms the Meta Relations instance level/meaning side. The connection between the two is thus expressed by semantics Pairs (dashed and dotted arrow in the figure). In the illustrations we are sticking with the placement of elements from the viewpoint of the language to be defined and place the Meta Relations definition at an angle to it to indicate its orthogonality to these levels.

III.1 On the Need for Meta Relations

In this section we argue that there is a need for a special construct to map meta models. The application scenario is provided by denotational meta modeling (cf. Fig. III.1). We will at first derive requirements for the mapping technology from the general requirements for a semantics definition in Subsect. III.1.1 and survey existing approaches according to these requirements in Subsect. III.1.2. We close the section by formulating our concept of Meta Relations in contrast to the afore discussed approaches.

III.1.1 Requirements for Mappings

In Subsection II.2.4 we pose requirements toward a technique suitable for the expression of a VML's semantics. Since semantic mappings form a part of our proposed solution DMM, they need to adhere to these requirements as well. A suitable mapping technique should thus be precise and formal, yet allow for underspecification. It should also be highly understandable and in the denotational meta modeling framework this implies a visual technique aligned to the MOF framework of meta modeling.

Adequacy applies to a mapping technique in that it must be able to handle commonly encountered situations with dedicated elements. One such situation is the *nested mapping*. Nested mappings occur when the domain (i.e., syntax) meta model uses aggregation. The tight relation between syntactic classes formed by aggregations is often reflected in the semantic domain and needs to be preserved by the semantic mapping. Regard Fig. III.3 for an example. Here, we map a

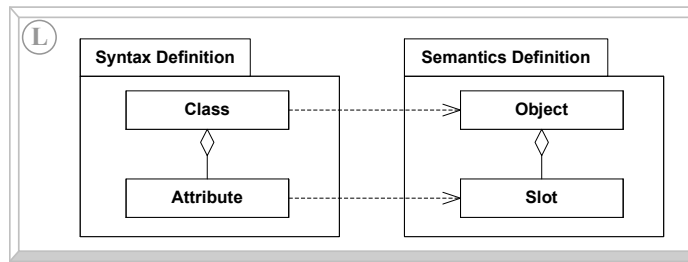


Figure III.3: Example for nested mappings

class to an object and an attribute to a slot. We intuitively expect, however, that *each* attribute of a class is mapped to one slot of *its* objects and not to slots of arbitrary other objects. The domain and range of a nested mapping are thus derived from a pair of another relation. This pair is called the *scope* of the nested relation. As such situations occur frequently, adequate support for nested mappings is desirable in a mapping technology.

Relations must be able to express two different aspects of the Pairs they define: On the one hand they express constraints on the construction of a single Pair, i.e., restraining the type of Objects which might partake in Pairs of the Relation. On the other hand Relations need to express completeness properties of the *set of their Pairs*, resulting in (right/left) total Relations.

A specific requirement is that mappings must not interfere with the meta models on either side of the mapping. There are a lot of (syntactic) meta models around and knowledge and tools have been build on these meta models. The addition of semantic mappings must respect these previous investments by forming a *neutral* addition to the existing meta models.

A further distinction between mapping technologies is whether the pairs defined by a relation are temporarily computed ad-hoc or retained as persistently stored instances. The former is usually the preferred in fully computable scenarios (i.e., in scenarios where the definition allows for a precise determination of the unique corresponding element given one element of a Pair), the latter if the pairs contain additional information not derivable from the relations (e.g., a user's selection between different alternatives). As we explicitly want to support the mapping of incomplete models, we cannot assume fully computable semantic relations. A technique for the expression of semantic mappings in DMM has to provide *persistent pairs*.

III.1.2 Existing Approaches

In practice (see the overview of approaches in Sect. II.3.1) we commonly encounter natural language descriptions of mappings (which are informal and imprecise) and operational transformation approaches (rule-based, tool-based, or programming language-based) which usually assume complete computability and fail to produce persistent instances. We are mainly interested in *descriptive and visual* (i.e., model-based) approaches to the formulation of meta model

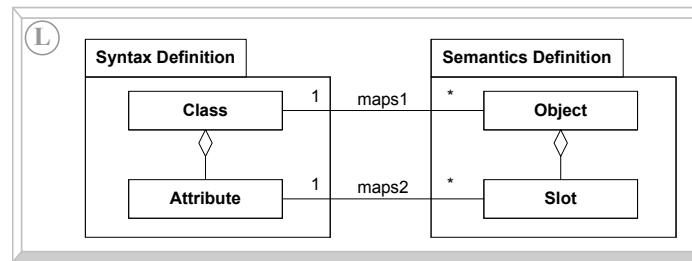


Figure III.4: Using associations to express mappings

mappings here and discuss them in the following paragraphs.

Our first choice of looking for a suitable model-based mapping technique is the MOF, the OMG's meta modeling language. In [Obj03a], p.51 we find that the only kind of relationship which MOF supports is an association. Associations are bidirectional, have well-known semantics, and OCL constraints can be used to express further properties. Examples for the use of associations as model mappings can, e.g., be found in the OCL 2.0 specification [Obj03c] or the denotational meta modeling approaches discussed in Subsect. II.3.2. In fact, the absence of standardized alternatives makes associations probably the most commonly used construct to express inter-model relations.

The use of associations as meta model mappings is rather problematic, though. As [Obj01] specifies and, e.g., [Ste02] elaborates, associations are meant to represent structural or behavioral connections *between elements of one model*. Tools (and modelers) interpret them in this way and, e.g., generate code accordingly. Using associations to express the semantic mapping results in fusing the syntactic and semantic domain to a single, integrated meta model. The clear distinction of the denotational approach between the domains is reduced to separation by packaging only (cf. Fig. II.10). The technical feasibility of encoding mappings in the models themselves depends on all participants respecting the different roles packages and associations play in this integrated meta model. Practically, this distinction is all too often ignored and semantic concepts are simply connected directly to syntactic meta model elements (compare, e.g., the so called *Abstract Semantics* of CMOF [Obj03a] p.58ff). In principle, the usage of associations to realize mappings ignores that mappings are supposed to be *between* meta models, not *in* them. It thus severely violates the requirement of neutrality.

A further technical disadvantage of using associations as mappings is that completeness criteria can only be expressed by using multiplicities which are defined on the set of *all* instances of a class. To understand why this is not adequate for mappings consider the following scenario: Taking up the scenario from Fig. III.3, we want to map *Class* and *Attribute* to *Object* and *Slot* respectively. We want to express that a class may be mapped to an arbitrary number of objects. For each of these objects, however, we need to make sure that it has slots for *all* attributes of its class. Looking at the only possible solution of this scenario by associations in Fig. III.4, we find that associations are able to precisely capture the former constraint only. Each class may be related (via *maps1*) to an ar-

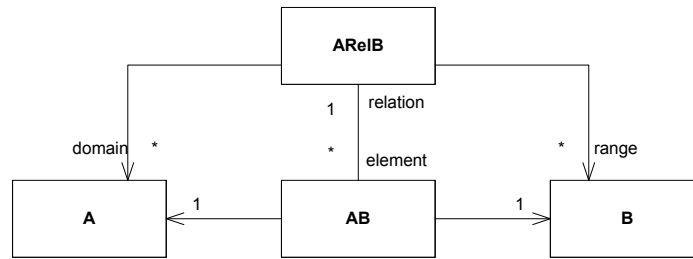


Figure III.5: Model pattern for Relations

bitrary number of objects. For the nested relation `maps2`, however, we cannot strengthen the cardinality at the `Slot` end beyond the non-committal '*' as there are (in a global scope) attributes (of classes without instances) which are not mapped at all and attributes which are mapped to a multitude of slots. While we can use OCL constraints to ensure that attributes are only mapped to slots of its class's objects, we simply cannot specify with associations that *for each* mapped class *all* of its attributes have to be mapped, too.

To understand this inability, we have to take the point of view of a single association instance (a link), since this is the context for evaluating the OCL constraints defined for the association. From this point of view we can navigate to the endpoints of the link (i.e., instances of the domain and range classes), but never back to other mapping links as the models should be unaware of the mappings. Thus from this point of view we can never establish the completeness of all pairs of a relation.

A specific pattern to overcome these limitations of associations is presented by Akehurst and Kent in [AK02a] (see Fig. III.5). It promotes the introduction of a relationship (`AReIB`) and a pair class (`AB`) for each intended mapping of model elements (`A` and `B`). OCL specifications are used to define domain, range and additional constraints of the relationship. Special predefined OCL constraints also ensure that the instances of the pattern correctly express nested relations. The trick here is that an instance of the `AReIB` class can capture all domain and range elements and determine the completeness of the set of Pairs (i.e., instances of `AB`).

While this pattern provides the means to express nested mappings and completeness criteria, it comprises applicability problems: Every time a relation according to this pattern is built, two new classes are created and all OCL definitions that accompany the pattern have to be replicated with respect to the new names of the classes. The resulting structure is quite complex, and it is hard to separate the original model classes from the classes capturing the relation. Again we are faced with the problem of mixing the content of a meta model with its mappings by using identical elements to express both.

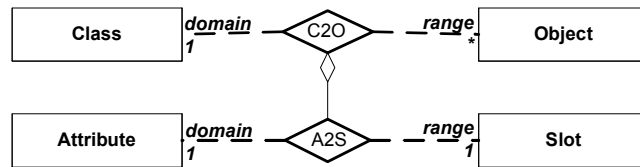


Figure III.6: Example for the use of Meta Relations: Definition

III.2 Concept of Meta Relations

The idea we are following in this thesis is to introduce Meta Relations as a first-class feature in the meta modeling language (in the OMG's framework that would translate to an extension of the MOF). This avoids the problems of other approaches: All Meta Relations are visually and technically distinct from standard meta model elements. They can be provided with the necessary means to express completeness criteria and to support nested relations in a convenient way. Further constraints can be placed upon a Meta Relation to address specifics of the mapping. Relations are always binary and may connect either two single classes or tuples expressing a combination of classes. Associations between Relations can be used to capture references to existing mappings. These enable modular and redundancy-free mapping specifications which are better maintainable.

Figs. III.6 to III.9 give an exemplary overview of the central concepts of Meta Relations (we will provide the exact definitions in the succeeding sections): Fig. III.6 shows the definition of Relations for the Attribute-Slot example. While Relations C2O exposes no difference to the realization of relations with associations, Relation A2S can now conveniently express that every Attribute needs to be mapped to exactly one Slot (in the context of one C2O Pair). The reason we can provide this strengthened cardinality is because in the accompanying OCL constraints we formulate that the domain/range of A2S is formed by all attributes/slots of a specific class/object pair.

```
context A2S
  -- Attribute to Slot
  domain=scope.domelement.Attribute
  range=scope.ranelement.Slot
```

The definition of the domain and range objects first navigates to the `scope` Pair. The name `scope` is pre-defined to address the Pair which defines the scope of the nested relation (as defined by the aggregation between the Relations). The names `domelement` and `ranelement` allow access to its respective endpoints. Thus the domain of the nested Relation A2S is defined by accessing a Pair of the C2O Relation, navigating to its domain end (yielding an instance of Class) and selecting the set of its Attributes.

As this definition relies on a specific Pair of the scope Relation C2O, the domain/range of A2S is recalculated for every instance of this relation. And for these specific sets, the relation must indeed be bijective. Note the difference to associations, which can only express cardinalities in a global scope.

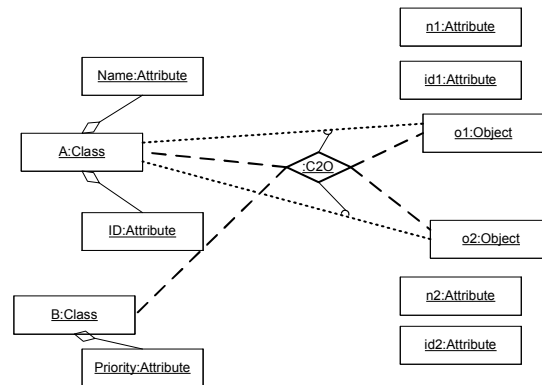


Figure III.7: Example for the instances of Meta Relations: Instance of the C2O Relation

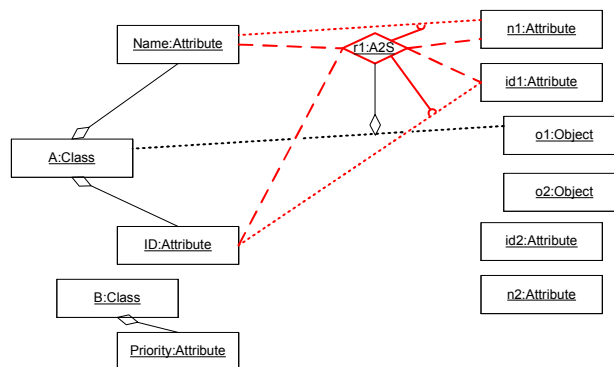


Figure III.8: Example for the instances of Meta Relations: Instance of the A2S Relation

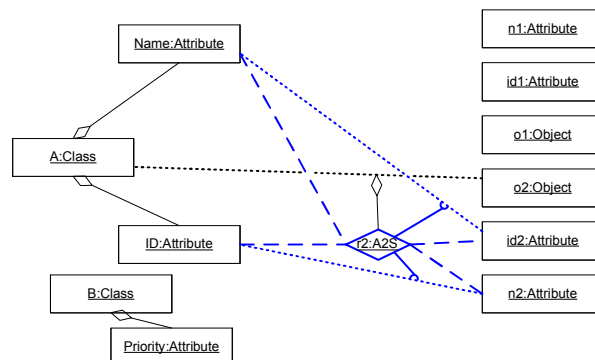


Figure III.9: Example for the instances of Meta Relations: Instance of the A2S Relation

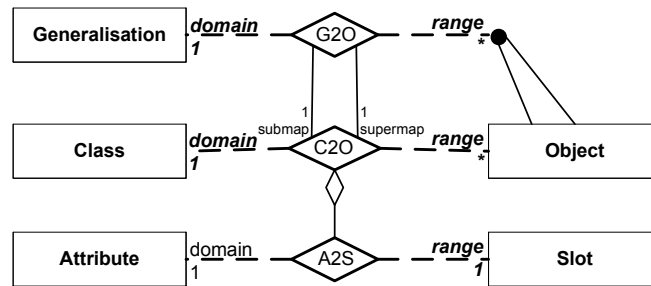


Figure III.10: Example for the concrete syntax of Meta Relations

Figs. III.7 to III.9 illustrate how the instances of Meta Relations are structured to manage the compliance to these nested definitions (all three figures show excerpts from the same instance, the splitting has been done for presentation purposes only): Of relation C20 there is only one instance (see Fig. III.7) which has a global scope, i.e. it has all Classes (A and B) in its domain and all Objects o1 and o2 in its range. Its pairs (dotted lines) show that both objects are of type A, none of B. This complies with the relation definition which calls for a class per object but allows for zero objects per class.

For both pairs of C20 there are now *separate* instances of A2S (r1 and r2) (also separately depicted in Figs. III.8 and III.9). Each of these relation instances has the attributes of A as its domain and the slots of *its* object as its range. This information is derived by accessing the S2O-Pair which forms the *scope* of this Relation instance.

Note how the Pairs of r1 and r2 fulfill the constraints of the A2S relation: both relation instances are bijective. Yet, there are unmapped attributes in the global scope (Priority).

The instance level structure is thus adapted from the pattern introduced above. An instance of a Relation is thus not a single element but a pattern of elements.

After this brief introduction to Meta Relations we now proceed to lay out their technical details in more detail, starting with the concrete syntax.

III.3 Concrete Syntax for Meta Relations

We describe the concrete syntax' symbols informally here only. See Fig. III.10 for an accompanying example rendering.

A Meta Relation is depicted as a dashed line between the classes it connects. These classes define the type of the domain and range objects participating in the Relation's Pairs (which may be further restricted by accompanying OCL specifications). A diamond shape in the middle of the Relation contains its name. Associations between Relations can be attached to the diamond shapes. The multiplicities and role names of the Relation ends are visualized in the same way as a UML AssociationEnd. If associations and relations appear in the same diagram, we usually provide the labels at the relation ends in bold and italics

to set them apart. Tuples are visualized by a black dot which is attached to one Relation end. The role name and multiplicity of the Relation end are then shown near the tuple. The tuple itself has connections to its elements. Nested Relations (Relations that are defined in the context of another Relation) are depicted by connecting the nested Relation by an aggregation association to its context Relation.

The example in Fig. III.10 thus expresses that a class `Class` is related by Relation `C2O` to the class `Object`. The role names designate `Class` as the domain of the Relation and `Object` as its range¹. The multiplicities specify that Relations `C2O` and `G2O` are inverse functional and that `A2S` is bijective. Note that `A2S` is defined in the context of `C2O`, thus a Pair of `C2O` determines which `Attributes` and `Slots` form the domain and range of (instances of) this Relation. The association between `G2O` and `C2O` is a normal association that enables `G2O` to access the `C2O` Pairs that map its endpoints to their respective objects. The additional OCL constraints defining the specifics of how this association is employed are omitted here.

III.3.1 Concrete Syntax of Relations Instances

Besides the Relations (i.e., the type level), users also need to inspect Pairs (i.e., the instance level) occasionally: As semantic mappings usually relate syntactic elements to (unlimited) sets of semantic entities, users need to select appropriate examples from this set to enable validation/verification tasks. Similarly, the results of these tasks need to be inspected in relation to the original model's syntax. Thus a visual notation for Pairs is also required.

Fig. III.11 shows the concrete syntax for this instance level. `RelInstances` are rendered as a diamond shape. It contains the name of the instance and its type (the name of the Relation definition). In line with the UML object notation these names are separated by a colon and are underlined. All domain and range objects are connected to this diamond by dashed lines again. Pairs are depicted as dotted lines between the domain and range objects. All Pairs belonging to a `RelInstance` are connected to it by solid lines with a half circle forming the connector between the lines. If a `RelInstance` is defined by a nested relation, the `RelInstance` is connected to the defining Pair of the scope relation again using the composition notation. This re-use of visualization concepts from the definition level allows for an easy transfer of concepts and makes for an intuitive understanding of the diagram.

Even in this small example it becomes apparent that such instance level diagrams have an easily cluttered structure. Simplifications are needed to make these diagrams more accessible. One important simplification is the *projection on relation types*. This means that a view of an instance level diagram can focus on a single relation instance and show only its domain and range objects as well as its Pairs. The domain and range objects can furthermore be more conveniently visualized by capturing them in a shaded shape. This way the

¹To avoid cluttered diagrams, one of the related meta models can be designated as the domain/range model thus making all relation ends connected to this meta model the domain/range ends by default.

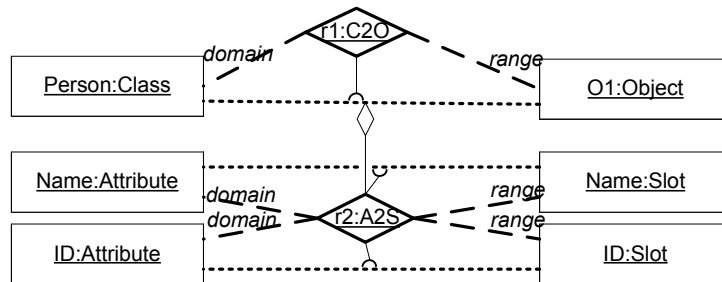


Figure III.11: Example for the instance notation of Meta Relations

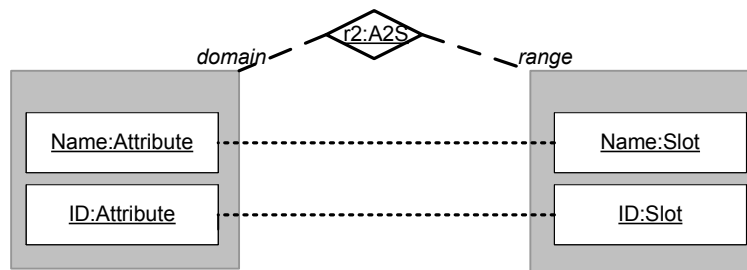


Figure III.12: Simplified projection on r2 of Fig. III.11

intersection of the dashed lines from the pairs and the domain/range definition is avoided. Fig. III.12 shows a projection on r2 of the diagram in Fig. III.11.

This diagram now closely resembles the intuitive notation that is mostly used when introducing relations in a mathematical context. It shows two sets (depicted as shaded shapes), elements in these sets and the connections (pairs) between them in conformance to a given relation definition. We believe that this kind of visualization is very intuitive for most users, yet we still retain a formal level of definition.

III.4 Abstract Syntax for Meta Relations

The meta model defining the abstract syntax of Meta Relations containing relations is given in Fig. III.13. Following the MOF architecture [Obj03a] it reuses elements of the UML 2.0 Infrastructure [Obj03b] (from the Core:Constructs package) whenever applicable and combines them with new elements to express our relation concepts.

Relation is the main class of the new model. This class captures the definition of a connection between two model elements. It therefore refines the concept of a DirectedRelationship from the Constructs package. Since Relations have instances and contain OCL constraints for them, they inherit from Classifier. Similar to associations, Relations are connected to two ends which contain information about the way the instances of the relation can be constructed. While the domain and range ends only indicate the type of the connected elements, OCL

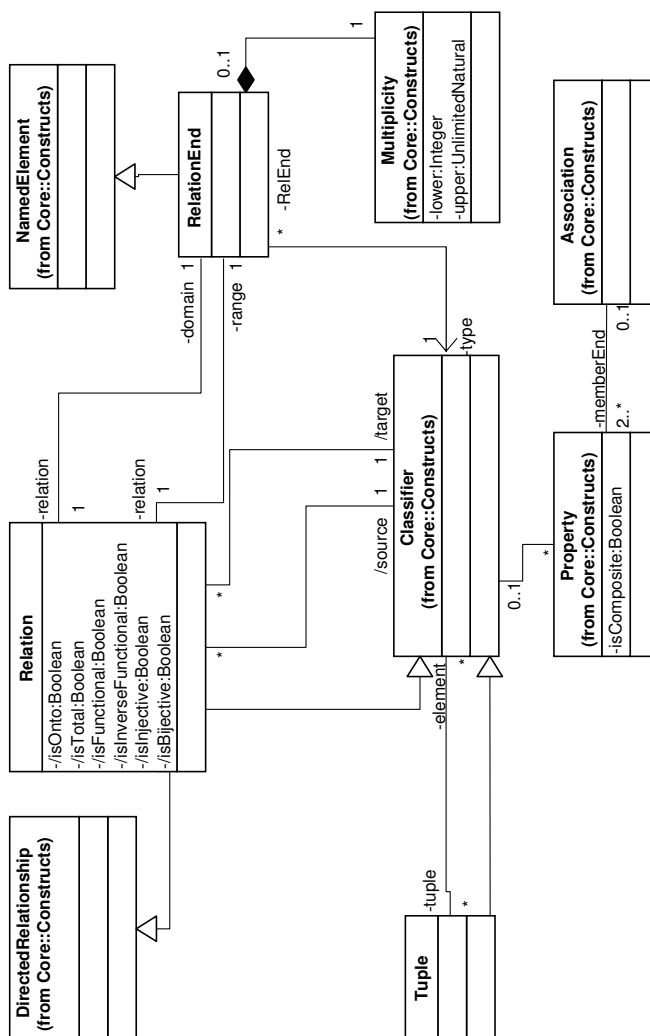


Figure III.13: Meta model defining the abstract syntax of Meta Relations

constraints may be used to further refine these sets. The source and target links are redefined from `DirectedRelationship` and indicate the classifiers connected to the domain and range `RelationEnds`.

```
context Relation
    source=domain.type
    target=range.type
```

For mathematical relations there are well-known properties like functional, total etc. When modeling relations we encode this information in the multiplicity constraints at the `RelationEnds` (e.g., an upper limit of 1 on the range end of a relation indicates that every domain element may participate at most in one pair of the relation; this corresponds to the definition of functional relations). Thus, the meta class `Relation` only has derived attributes for easier access. Further properties like *is bijective* can be combined out of these basic features.

```
isFunctional = (range.multiplicity.upper=1)
isInverseFunctional = (domain.multiplicity.upper=1)
isTotal = (domain.multiplicity.lower=1)
isOnto = (range.multiplicity.lower=1)
```

Note that in contrast to [AK02a] these definitions are not derived from the actual instance of the relation but are definitions on the specification level. The attribute `isOnto` thus states that an instance of the Relations *should* cover all range elements but situations may arise (e.g. by adding new elements to the model) in which the constraint does not hold. This may trigger inconsistency detection and reconciliation mechanisms (see, e.g., [HHS02b]).

A `RelationEnd` is similar to an `AssociationEnd` as known from Class Diagrams. It contains a role name by which the relation can address the Classifier that forms the type of the `RelationEnd`. `RelationEnds` contain multiplicities. These specify in how many Pairs of a Relation a domain or range instance may or must participate. Note that these specifications are not global like with associations but rather dependent on the concrete instance of the relation. This is necessary to provide the facilities for nested relations. It is also the fundamental difference to `AssociationEnds`². If no multiplicity is explicitly specified, "*" is used as the default value.

A tuple is a necessary addition to the meta model to enable the mapping of structures. It is a classifier itself and may thus participate in relations. A tuple comprises an ordered set of elements. We do not provide additional details here because tuple is just an auxiliary construct which is, e.g., also defined in [Obj03c].

III.5 Semantics of Meta Relations

As Relations are a purely static concept, their semantics can be expressed adequately by the denotational meta modeling framework. We thus supply a semantic domain meta model and the semantic mapping to define the meaning

²In the UML 2.0 submission `AssociationEnds` have been integrated in the more general concept of `Property`, see [U2P03] for details.

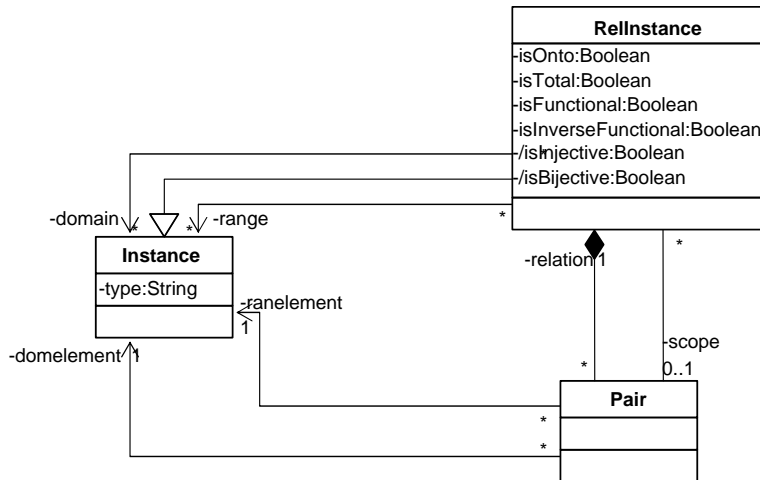


Figure III.14: Semantic domain meta model for Meta Relations

of Meta Relations. Note that as Meta Relations are intended to be used in denotational meta modeling, they are used to express their own semantic mapping. We discuss a way out of this meta-circularity later on.

III.5.1 Semantic Domain

The semantic domain meta model is given in Fig. III.14. It comprises three classes: Instance, RelInstance, and Pair, which we detail in the next paragraphs.

Instance is a generic class that captures the fact that an element of the semantic domain corresponds to some defining element (its type).

A RelInstance is an instance of a Relation. It contains the set of all domain and range elements (according to the specification given in its defining Relation). A RelInstance contains a set of Pairs. These Pairs must be unique in the RelInstance. A Pair can form the scope of a RelInstance (though never for the one it belongs to).

```

context RelInstance
  inv:
    self.Pair->forAll(x,y|(x.domelement=y.domelement)
      and (x.ranelement=y.ranelement)
      implies x=y)
    not self.Pair->includes(scope)
  
```

A Pair embodies the combination of elements from domain and range included in the RelInstance. Each Pair is linked to its containing RelInstance and one element from the domain (domelement) and the range (ranelement). It has got to be ensured that only elements from the sets defined by the RelInstance are connected.

```

context Pair inv:
  
```

```

relation.domain->includes(domelement)
relation.range->includes(ranelement)

```

The tricky bit here is that Relations as we understand them do not conform to the usual concept of instantiation where a classifier describes a set of (isolated) instances that conform to its specification. Rather, a set of patterns made up from a `RelInstance` and `Pairs` contained in the `RelInstance` is defined by the `Relation`. The specification present in the `Relation` has an impact on different parts of this pattern. The domain and range sets of the `RelInstances` conform to the description provided by the `Relation`. The `Pairs` contained in the `RelInstances` conform to additional constraints provided by the `Relation`. The set of all `Pairs` in one `RelInstance` conforms to the multiplicities specified at the `RelationEnds`. Composition associations between `Relations` will result in `scope` links between a `Pair` that stems from the `RelInstance` of the containing `Relation` and a `RelInstance` conforming to the nested `Relation`.

These connections can now be made precise and explicit by using Meta Relations to specify the semantic mapping.

III.5.2 Semantic mapping of Meta Relations

Fig. III.15 provides the mapping between the abstract syntax and semantic domain for Meta Relations. Note that only parts of the meta models are shown here to avoid cluttering the figure.

The most basic Relation is C2I, the Classifier to Instance mapping. This mapping expresses the general semantics of classifiers which says that each classifier describes a set of instances. The instances have a type attribute that indicates the classifier they conform to.

```

context C2I
  -- Classifier to Instance
  domain= Classifier.allInstances
  range= Instance.allInstances
  inv: ranelement.type=domelement.name

```

The Relation capturing the semantics of Relation is R2RI, Relation to Relation-Instance. It contains several constraints to ensure that the specifications in the abstract syntax are correctly related to the elements of the semantic domain.

```

context R2RI
  -- Relation to RelInstance
  domain= Relation.allElements
  range= RelInstance.allElements
  inv: domelement.isOnto=ranelement.isOnto
      domelement.isTotal=ranelement.isTotal
      domelement.isFunctional=ranelement.isFunctional
      domelement.isInverseFunctional=ranelement.isInverseFunctional

```

The pairs conforming to a `RelInstance` must obey the multiplicity constraints imposed by the defining `Relation` (we only show one of these constraints, the others are constructed similarly).

```

ranelement.domain->forAll(i| ranelement.pair->select(p|
  p.domelement=i)->size>domelement.range.multiplicity.lower)

```

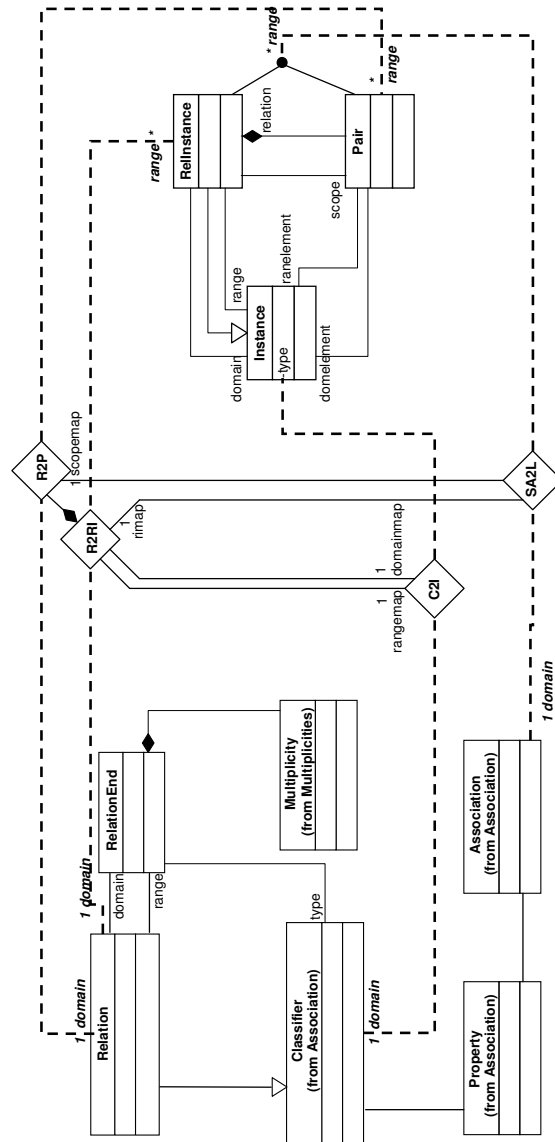


Figure III.15: Mapping syntactic and semantic domain meta models of Meta Relations

To check whether the domain and range objects are correctly typed, an access to the C2I mapping is necessary. This is denoted by the associations between R2RI and C2I which can be accessed from R2RI using the role names `domainmap` and `rangemap`. Since invariants in a relation will be evaluated in the context of a `Pair` (of the R2RI relation), the navigation `self.domelement` will result in a `Relation` and `self.ranelement` in the corresponding `RelInstance`.

```
domainmap.domelement->includes(self.domelement.domain.type)
rangemap.domelement->includes(self.domelement.range.type)
domainmap.ranelement->includes(self.ranelement.domain)
rangemap.ranelement->includes(self.ranelement.range)
```

The relation R2P connects `Relations` and `Pairs`. It expresses that the invariants of the `Relation` constrain the way the `Pairs` in its `RelInstance` are built. Thus this relation can only happen in the context of an R2RI mapping. The containing `Relation` can always be addressed from the contained relation using the role name `scope`.

```
context R2P
  -- Relation to Pair
  domain=scope.domelement
  range=scope.ranelement.Pair
```

The R2P relation thus connects the concept of a `Relation` with the `Pairs` defined by it by referring to the (already established) R2RI `Relation`. From a `Pair` of R2RI forming the scope of an R2P `Relation Instance`, we can derive which `Relation` is to be mapped (its `domelement`) and which `Pairs` should be mapped (all `Pairs` of the `RelInstance` forming the `ranelement` of the scope `Pair`).

To give a formalization of the constraint evaluation, the abstract syntax and semantic domain of OCL expressions have to be defined as meta models. This is accomplished by the OCL 2.0 specification [Obj03c]. Yet the structures defined there are quite complex and using them here would require considerable explanations without providing much insight to the relation technique. They are thus omitted here.

The fourth `Relation` connecting abstract syntax and semantic domain is called SA2L, `Scope Association to Link`. A composition between two `Relations` is mapped to a `Pair` and a `RelInstance`.

```
context SA2L
  -- Scope Association to RelationInstance-Pair Link
  domain= collect(a:Association| (a.property.isComposite=(true)
    and (a.property.type=Relation))
  range= collect(t:tuple|t.RelInstance.scope=t.Pair)
```

The `Pair` contained in the tuple forming the range must be the scope of the `RelInstance`. By referring to an R2RI and an R2P mapping it is ensured that only instances of the correct relations are connected.

```
inv: tuple.relInstance.scope=tuple.pair
scopemap.domelement.property->exists(p|p.isComposite=true
  and p.association=self.domelement)
scopemap.ranelement->includes(self.ranelement.pair)
rimap.domelement.property.association->includes(self.domelement)
rimap.ranelement->includes(self.ranelement.relInstance)
```

III.6 Summary and Discussion

Meta Relations as introduced in this chapter allow for a visual specification of mappings between meta models. They provide facilities for nested mappings and associations between Relations. Their semantics is precisely defined. Meta Relations thus fulfill our requirements for a technique to specify semantic mappings in Dynamic Meta Modeling. Their suitability for this task is demonstrated by their use in defining the semantic mapping in their own definition.

Several aspects of Meta Relations remain to be addressed, though: While Meta Relations provide a visual notation, the core of their specification lies in the associated OCL formulae. We do not currently see viable diagrammatic alternatives to OCL although several visualizations are suggested in the literature, e.g. [BKPT01, Ken97]. The conciseness of textual logic formulae is the deciding factor here. To judge the impact of this decision on the overall framework of DMM we have to distinguish between our user groups. Language Engineers who write language specifications can be expected to be proficient in OCL as it is an established part of the OMG's modeling framework. Advanced Language Users as readers of such a specification on the other hand can obtain most information from the visual part of the specifications. The OCL specification needs to be inspected for very intrinsic details only. Thus we see only a very limited impact on the overall understandability of the approach.

Two other issues are left open: formalization and operationalization of Meta Relations. The former is an issue if one wants to avoid the circularity of the meta modeling definition. As the core of Meta Relations lies in their OCL formulae, we can rely on existing OCL formalizations [Baa02, FM03, HHB02] to provide the required external foundation.

A tool which operationalizes Meta Relations specifications is not yet available. Such a tool would compute a (partial) domain/range model, given its opposite. There are, however, two approaches very similar to our Meta Relations³ which are backed by transformation tools. One is the work of Akehurst and Patrascioiu [AKP03] which is based upon the relationship pattern. The other is the QVT Partners work [QVT] which features a notation which is very similar in spirit to Meta Relations but adorned with operational modules to enable transformations. If the latter approach is able to deliver the expected general mapping notation and transformation engine for the OMG's meta modeling framework, we might consider replacing Meta Relations with this standardized technique in the long term.

³The similarity is due to common roots of all three approaches

Chapter IV

Graph Transformations

The technique of Graph Transformations forms the backbone of DMM's operational component. Thus, in this chapter we turn to the notion of Graph Transformations in general and the particular variant of it which allows us to use them in DMM as outlined in Sect. II.4.

Graph Transformations are a well established theoretical notion with a simple and intuitive core: Given a graph (a structure of nodes and connecting edges), a graph transformation produces another graph which is altered in some way. The alteration performed by the transformation is captured by a *Graph Transformation rule*.

Graph Transformations have been a topic of intense theoretical research for more than thirty years now. Based upon fundamental concepts by Pfaltz [PR69], Montanari [Mon70], Schneider and Ehrig [EPS73], and others, a wealth of slightly different formalisms has emerged. An overview of these different variants of Graph Transformations and their applications can be gained from the Handbook of Graph Grammars [Roz97]. The differences between the approaches concern the definition of the *underlying graph notion*, the way a rule is applied, and the way a single rule application can be *controlled and combined* with other rule applications. In the following sections we investigate each of these three aspects. We discuss existing approaches from the literature, pick the features useful for the DMM approach, and provide formal definitions for the innovative mechanism of *rule invocations* which is introduced specifically for DMM.

The usefulness of Graph Transformations (and their detailed features) for the DMM approach is measured according to the following criteria:

Understandability We choose Graph Transformations as the underlying formalism because they already provide a high degree of understandability in their basic form. This basic form is also easily visualizable in UML's Communication Diagram notation. We need to make sure that this claim still holds if the base formalism is extended with special features.

Adequacy For the practical application of a formalism it is not only important that it is able to express certain concepts but that the expression of com-

monly encountered situations is possible in a convenient way. As we need to manipulate graphs which are essentially meta model instances, adequate support for object-oriented concepts is mandatory. For realistically sized examples adequacy also extends to the maintainability of the formalism. If local changes can only be realized by altering large parts of an already existing specification, no realistic use of a formalism can be expected.

Analyzability A benefit of basing DMM on the formal specification concepts of Graph Transformations is the possible application of existing theoretic results to enable the analysis of semantics specifications. Different GT variants support different analysis notions. In general, the more basic and restricted a GT approach is, the more analysis results can be gained from it.

Tool Support There are a number of tools which allow to create, apply, or analyze Graph Transformation rule sets. The GT approaches and features used in these tools differ widely. While DMM is not geared toward a particular tool, utilizing this existing support is an aim of our approach. DMM should thus only employ those features of GTs which are either well-known and broadly supported or which can be reduced to basic GTs in a simple way.

IV.1 Graphs

The most basic notion of a graph G is a set of nodes N and a set of edges E with $E \subseteq N \times N$. Note that this definition disallows multiple edges between identical nodes. So called *multi-graphs* explicitly allow multiple edges by defining a Graph as $G = (N, E, s_E, t_E)$ with functions $s_E : E \rightarrow N$ and $t_E : E \rightarrow N$ designating the source and target node of each edge. All edges in this graph variant are unidirectional and connect exactly two nodes. *Undirected* graphs and *Hypergraphs* (with edges connecting more than two nodes) also exist but are not considered here.

Labeled graphs add labels to nodes, edges, or both. Labels are used for different purposes: Early GT approaches used them for denoting type information (now specified by typed graphs, see below), we use them to denote the names of elements. We assume a combined label alphabet Λ which provides labels for edges, nodes, and further elements alike. The amount of labels is not bounded and we assume an operation `new` to provide a yet unused label. Λ contains a special label \perp which denotes an undefined name. Comparisons between \perp and any other symbol will always yield *false*, \perp compared to \perp yields *true*. An extended alphabet $\Lambda' (= \Lambda \cup \{\bullet\})$ introduces the special symbol \bullet called the *wildcard* which yields *true* when compared to any other label of Λ' , even for \perp . A basic graph for the purpose of our approach is an edge and node labeled multi-graph.

Definition 1 (Graph)

$G = \langle N, E, l_N \rangle$ with
 N the finite and non-empty set of nodes

E the finite set of edges, $E \subseteq N \times \Lambda' \times \Lambda' \times \Lambda' \times N$
 $l_N : N \rightarrow \Lambda'$ the labeling function for nodes.

For convenience, we define the functions

$s_E : E \rightarrow N$ the function indicating the source node of an edge,
 $t_E : E \rightarrow N$ the function indicating the target node of an edge,
 $l_E : E \rightarrow \Lambda' \times \Lambda' \times \Lambda'$ the labeling function for edges.

Note that the name of an edge (designated by l_E) is a triple of labels. This reflects the structure of UML's Class Diagrams, in which associations can be distinguished either by an association name or by a role name at either end. Thus the name of an edge has the structure (role name source side, association name, role name target side). The definition implies that two edges running between identical nodes must be distinct in at least one of these labels. In practice, differences in all three label components are desirable.

Labeling is (in general) non-injective as there are situations in which different nodes may carry the same name. The most common example is the occurrence of multiple anonymous nodes in a rule which are all labeled by \perp .

Definition 2 (Edge-Label Preserving Graph Morphism (elp-morphism))

An elp graph morphism is a structure and edge label preserving morphism between graphs.

$m(G, H) = m_N$ with
 $m_N : N^G \rightarrow N^H$ the node mapping function¹,
and $\forall \langle a, l_1, l_2, l_3, b \rangle \in E^G : \langle m_n(a), l'_1, l'_2, l'_3, m_n(b) \rangle \in E^H$
with $l_1 = l'_1, l_2 = l'_2, l_3 = l'_3$

Note that due to the existence of the wildcard symbol in Λ' , the equality of labels does not imply complete identity.

IV.1.1 Typed Graphs

For application in a UML context, the notion of a *typed graph* [CMR96, CEL+96, BEdL+03, EEPT05] is important. In typed graphs, a structure is imposed on the way different nodes and edges can connect. A *type graph* is employed to describe this structure. Thus, the definition of a typed graph is

Definition 3 (Typed Graph)

$G_T = \langle G, type \rangle$ with
 G a graph as defined above,
 $type : G \rightarrow TG$ the typing elp-morphism and
 TG a graph with the additional requirement that l_N^{TG} is injective²

In an object-oriented interpretation, TG is the type definition level and the typed graph G_T is an instance of that definition. Note that it is usually sufficient to provide the typing of nodes only as the edge typing is implied by the label preservation.

¹The upper index always indicates the superordinate element, i.e. the graph name in this case.

²Injectiveness of the node labeling guarantees unique class names.

IV.1.2 Inheritance in Typed Graphs

Inheritance is a central concept in object-orientation. Inheritance in Typed Graph Transformations is studied by Ehrig et al. [BE_dL⁺03, EPT04]. The following definitions are adapted from there.

Definition 4 (Type Graph with Inheritance)

Inheritance in the type graph TG is expressed by special edges I

$G_I = \langle TG, I, A \rangle$ with

TG a graph with the additional requirement that l_N^{TG} is injective

$I \subseteq N^{TG} \times N^{TG}$ the set of inheritance edges which must not form a circle

$A \subseteq N^{TG}$ the set of abstract nodes.

The edges of I denote the existence of an inheritance relation between two classes. Tuples of this relation connect sub classes to super classes (i.e., the direction of these edges corresponds to the UML generalization symbol). The simple relational definition of an edge suffices here, as inheritance edges neither have labels nor are multiple inheritance edges between two classes possible. Circular inheritance relations must not occur. The semantics of the constructs A and I are that any node in A may not have a direct instance in a graph (i.e., nodes in A represent *abstract* classes). Any node n which is connected to a node m by an inheritance relation (i.e., $\langle n, m \rangle \in I$) inherits all properties from m . Instances of n may thus have edges which are not defined for n in the type graph but which are defined for m . Inheritance is transitive, thus a node may have a tree of subclasses (called its *clan*).

Definition 5 (Inheritance Clan for a Type Graph with Inheritance)

For a node n in a type graph with inheritance its inheritance clan is defined by

$$\text{clan}_I(n) = \{n' \in N \mid \exists \text{ path } n' \xrightarrow{*} n \text{ in } I\}$$

A *closure* of a type graph with inheritance "flattens" the inheritance hierarchies by promoting normal edge definitions along the inheritance relations. The result is a graph without inheritance edges, i.e., a graph as defined above. In [BE_dL⁺03] two closures are distinguished: the *abstract closure* retains the abstract nodes (it does not, however, distinguish them in any way from normal nodes) while the *concrete closure* does not contain any nodes from the set A anymore. We adapt the definition with respect to our labeling concept.

Definition 6 (Abstract Transitive Closure)

Given $\langle TG, I, A \rangle$ with $TG = \langle N, E, l_N \rangle$ the abstract transitive closure is the graph $\overline{TG} = \langle N, \overline{E}, l_N \rangle$ with

$$\begin{aligned} \overline{E} = \{ & \langle n_1, l_1, l_a, l_2, n_2 \rangle \mid \exists o \in E : \\ & n_1 \in \text{clan}_I(s_E(o)) \\ & \wedge n_2 \in \text{clan}_I(t_E(o)) \\ & \wedge (l_1, l_a, l_2) = l_E(o) \} \end{aligned}$$

Definition 7 (Concrete Transitive Closure)

Given $\langle TG, I, A \rangle$ with $TG = \langle N, E, l_N \rangle$ the concrete transitive closure of $\langle TG, I, A \rangle$ is the graph $\widehat{TG} = \langle N \setminus A, \overline{E}|_{N \setminus A}, l_N|_{N \setminus A} \rangle$, with \overline{E} as defined above.

Depending on the context they appear in (i.e., either as a instance graph or as part of a rule) typed graphs are either typed over \overline{TG} (rule graphs) or \widehat{TG} (instance graphs).

IV.1.3 Attributes

A further OO concept that needs to be represented in the graph formalism is that of attributes. Classes may define *attributes*, i.e., names which instances of this class can bind to a value. Typically, the set of possible values is restricted by the definition of a data type for the attribute. Supporting the notion of attributes in Graph Transformations has been proposed in [LKW93, GHV03]. The most flexible and precise way to handle attributes is the integration of algebras into the definition a graph [LKW93]. Here, the type graph contains arbitrary algebraic signatures to define the data types and their operations. These algebraic structures need to be taken into account for all succeeding definitions (e.g., morphisms and single-pushouts are redefined in [LKW93], typed and attributed graphs are studied in [EPT04], typed attributed graphs with inheritance are mentioned in [BE_dL⁺03] and studied in [EEPT05]).

A second possible treatment of attributes avoids these redefinitions: Data types can be represented as a subset of nodes (DT) in the type graph. These nodes are being connected via special attribute declaration edges to normal (class) nodes. For each such data type $s \in DT$ there is a domain D_s containing all possible values of the data type. Instance graphs will contain these values in special nodes and connect them to the normal (object) nodes. This representation of attributes does not extend the usual notion of graph but rather constrains it slightly (some nodes may only have certain edges, cf. [GHV03] for details). While this handling of attributes is limited in its capabilities (for comparison as well as manipulation of these nodes), it allows for integration of the concept without changes to the underlying formalism. We make use of this notion of attributed graphs in [HHS02b, HHS02a] where DMM rules relying heavily on attribute manipulations are presented. In this thesis attributes do not play a prominent role but they are supported by DMM.

The treatment of attributes by Steinert and Plump in [PS04] can be seen as a way to combine the advantages of both techniques. They employ algebraic notation for the formulation of so called *rule schemas* but instantiate the variables in these schemas to plain value nodes before application of the rule. Thus they avoid the generation of large rule sets and retain the full power of algebraic attribute specifications. The price for this is a pre-processing step before rule application. A more general investigation of this pre-processing approach to attribute handling has recently been performed by Hoffmann [Hof05]. There, a concept of variables is introduced which does not only cover attribute manipulations but which forms a complete meta-level in that variables might also represent nodes and edges.

IV.2 Graphs in Dynamic Meta Modeling

The graph notion that is underlying the DMM approach is a typed, attributed, node and edge labeled multi-graph that allows for node inheritance in the type graph and uses special datatype nodes (DT) for the representation of attributes. We will call this combination a *DMM type graph*, a *DMM instance graph*, and a *DMM rule graph* respectively.

Definition 8 (DMM Type Graph)

$$\begin{aligned}
 G_{DMMT} = \{ \langle N, E, l_N, I, A, DT \rangle \text{ with} \\
 \langle \langle N, E, l_N \rangle, I, A \rangle \text{ a type graph with inheritance} \\
 DT \subset N \text{ the set of data types} \\
 l_N(N) \subseteq \Lambda \setminus \{\perp\} \\
 E \subseteq ((N \setminus DT) \times \Lambda \times \Lambda \times \Lambda \times (N \setminus DT)) \cup \\
 ((N \setminus DT) \times \{\perp\} \times \{\perp\} \times \Lambda \times DT) \}
 \end{aligned}$$

Note that data type nodes can only be the target of special edges which carry the name of the attribute at the target end label (and have no other labels). Node labels in type graphs may not contain the special symbols \bullet and \perp .

Definition 9 (DMM Instance Graph)

$$\begin{aligned}
 G_{DMMI} = \{ \langle G, type \rangle \text{ with} \\
 type : G \rightarrow \widehat{TG} \text{ the typing elp-morphism} \\
 TG \in G_{DMMT} \\
 l_N(N^G) \subseteq \Lambda \\
 l_E(E^G) \subseteq \Lambda \times \Lambda \times \Lambda \\
 \forall s \in DT^{TG} : type_N^{-1}(s) \subseteq D_s \}
 \end{aligned}$$

Definition 10 (DMM Rule Graph)

$$\begin{aligned}
 G_{DMMR} = \{ \langle G, type \rangle \text{ with} \\
 type : G \rightarrow TG \\
 TG \in G_{DMMT} \\
 l_N(N^G) \subseteq \Lambda' \\
 l_E(E^G) \subseteq \Lambda' \times \Lambda' \times \Lambda' \\
 \forall s \in DT^{TG} : type_N^{-1}(s) \subseteq D_s \}
 \end{aligned}$$

In contrast to DMM instance graphs, rule graphs allow the use of abstract nodes and wildcards as label names. The formulation of rules is thus more flexible.

Graphs are usually represented as diagrams rather than textually. We do thus provide rendering/parsing relations between graphical elements and their conceptual counterparts as presented above. Adhering to our goal of combining Graph Transformations with UML, we chose graphical symbols which also appear in the UML Class and Instance Diagram notation.

Table IV.1 contains an exemplary overview of the correspondences. Label strings are placed in quotes and Greek characters are used for node identities. Both


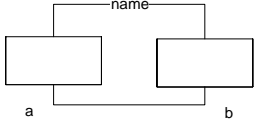

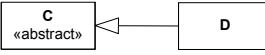
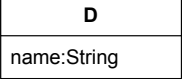
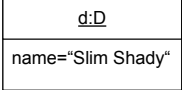
Element	Example (conceptual)	Example (graphical)
Nodes	$N = \{\alpha, \beta\},$ $l_N = \{\langle \alpha, "A" \rangle, \langle \beta, "B" \rangle\}$	
Edges	$N = \{\alpha, \beta\},$ $E = \{\langle \alpha, \perp, "name", \perp, \beta \rangle,$ $\langle \alpha, "a", \perp, "b", \beta \rangle\}$	
Typed Nodes	$N = \{\delta\},$ $l_N = \{\langle \delta, "d" \rangle\},$ $type = \{\langle \delta, \Delta \rangle\},$ $TG = \langle \{\Delta\}, \{\}, \{\langle \Delta, "D" \rangle\} \rangle$	
Inheritance	$N = \{\Gamma, \Delta\},$ $A = \{\Gamma\},$ $l_N = \{\langle \Gamma, "A" \rangle, \langle \Delta, "B" \rangle\},$ $I = \{\langle \Gamma, \Delta \rangle\}$	
Attributed Nodes (in type graphs)	$N = \{\Delta, \Sigma\},$ $DT = \{\Sigma\},$ $l_N = \{\langle \Delta, "A" \rangle, \langle \Sigma, "String" \rangle\},$ $E = \{\langle \Delta, \perp, \perp, "name", \Sigma \rangle\}$	
Typed Attributed Nodes	$N = \{\delta, "Slim Shady" \},$ $E = \{\langle \delta, \perp, \perp, "name", "Slim Shady" \rangle\},$ $l_N = \{\langle \delta, "a" \rangle\},$ $type = \{\langle \delta, \Delta \rangle, \langle "Slim Shady", \Sigma \rangle\}$ TG given in the previous row	

Table IV.1: Correspondence of graph concepts and their graphical rendering

special labels render to an empty string. This does not prohibit parsing, as \perp is used in type and instance graphs and \bullet in rule graphs only.

As the graphical correspondence in Tab.IV.1 shows, our graph notion allows for a visual representation of type, rule, and instance graphs in terms of UML Class and Instance Diagrams.

A different question which needs to be discussed is the reverse direction: Can all UML Class Diagrams be represented by our graph notion? Several features of UML are not directly supported by DMM graphs. Most important among these omissions is the concept of constraints and its special case of multiplicities. While type graphs impose structural restrictions on their instances, they do not provide the finer grained control of multiplicities, let alone those of OCL. A UML Class Diagram may thus express that some instance of class A must be associated to at least one and at most three instances of class B but the corresponding type graph can only express that there is a connection between A and B. Consequently, if a UML Class Diagram is represented as a DMM type graph, all legal instances of the Class Diagram can be represented as DMM instance graphs. However, there are legal DMM instance graphs (typed over

the type graph) which do not have a legal correspondence on the UML side. We need to keep this over-approximating nature of our graph notion in mind.

Another difference which needs to be overcome is that associations in UML Class Diagrams are usually undirected while our graph notion supports directed edges. There are basically two ways to deal with this difference: Either all UML associations are interpreted as a pair of (conversely) directed edges in a DMM graph or an arbitrary but consistent direction is fixed for associations in a UML Class Diagram and all Instance Diagrams and rules typed over it. While the first solution is conceptually more clean, it bloats the graph representations. We thus assume that UML associations contain an (implicit) direction which is respected by all instances.

IV.3 Graph Transformation Rules

If information is encoded in a graph structure, a way to manipulate this structure is needed. For graphs, Graph Transformations (GT) form this manipulation facility. Usually, one is not only interested in a single concrete manipulation but in a general pattern for them. Thus, the concept of *Graph Transformation rules* (GTR) is central to Graph Transformations.

IV.3.1 Basic Terminology

To enable concise discussions, we introduce some common terms: A *Graph Transformation rule* r consists of two graphs: The so called *left-hand side* (L) of the rule designates the portion of the host graph which is to be manipulated by the rule. The *right-hand side* (R) depicts the post-state after application of the rule. A production morphism between both graphs can be used to relate identical elements in both graphs. In practice, this morphism is usually implied by equality of the labels.

Elements in both the left and the right hand side graph are called the *application context* or the interface of the rule (in the example in Fig.IV.1, the application context consists of the nodes labeled A,B,C and the edge labeled u). Elements in $L \setminus R$ are deleted by an application of the rule (in the example node D and edges v,w) and elements in $R \setminus L$ are newly created (in the example edge x). We refer to rules textually as $L \xrightarrow{r} R$.

The graph on which to apply the rule is called the *host graph* (G). A *matching* mt has to be identified between the elements of L and G for the rule to apply. The image of L in G under mt is called the *occurrence* of L (marked gray in the figure). If the changes encoded in the rule are applied to G , a new graph H is derived from G . The post-application matching function mt' is identical to mt for elements of the application context.

The detailed formulation of these common notions differ between the different approaches to Graph Transformation.

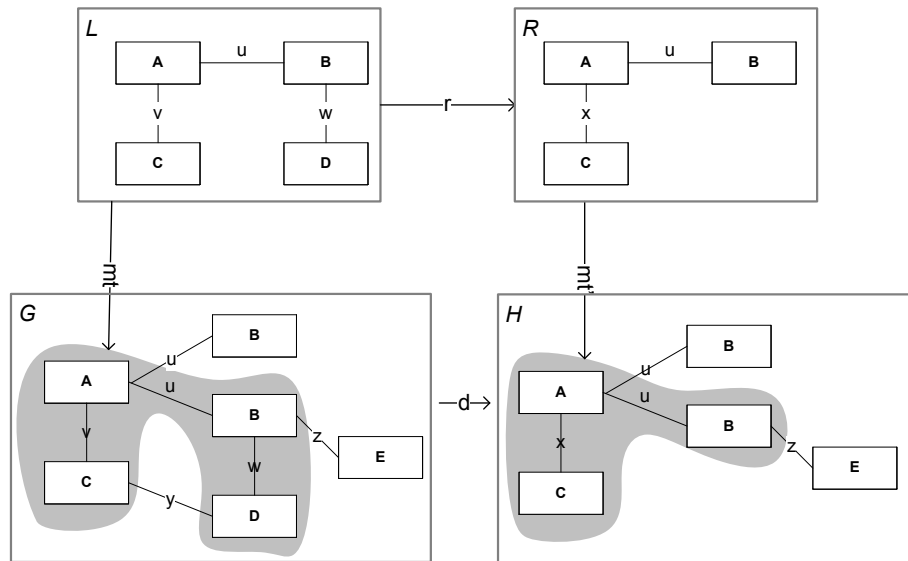


Figure IV.1: Basic graph transformation rule concepts

IV.3.2 Double Pushout vs. Single Pushout

The basic notion of Graph Transformation rules has been formalized in a number of ways. The most wide-spread approaches are the algebraic formulations of the Single-Pushout approach (SPO) [LE90, Löw93, EHK⁺97] and the Double-Pushout-Approach (DPO) [EPS73, EHK⁺97]. Besides theoretic differences, the main practically relevant distinction (assuming injective matchings) between the two approaches is the handling of dangling edges. If a node rn in the left hand side of a rule matches a node hn in the host graph, hn may have more edges attached to it than rn . If the rule application now calls for a removal of hn , the question arises what happens to edges in the host graph which have lost one of their anchoring nodes (they are called *dangling edges*) but which are not explicitly deleted by the rule application. Under the DPO approach, such a rule application is interdicted while the SPO approach calls for an implicit deletion of the dangling edges. The example in Fig. IV.1 demonstrates this difference: Under the DPO scheme, the derivation d would not be possible, as the y edge is not matched by rule r . Under SPO, however, it is implicitly deleted (as shown in Fig. IV.1).

While the DPO approach ensures that all Graph Transformations are reversible (which enables elegant proofs), it also forces modelers to specify rules for every possible edge grade of a node which is to be deleted. The SPO approach is more flexible in its matching and is thus preferred for practical applications. We also follow the SPO approach in DMM.

Choosing SPO over DPO does not prohibit the application of theoretic results gained for DPO. In [EHK⁺97] a generic embedding construction for DPO into SPO is given. Thus, theoretic results for either approach may—in principle—be transferred to the other, albeit with some restrictions.

IV.3.3 Rules

Definition 11 (Graph Transformation rule)

A Graph Transformation rule $r : (L, R)$ consists of two DMM-rule graphs L and R and a name r .

Note that for all following definitions, we assume that the different graphs are defined over a common base set such that, e.g. $L \cap R$ is defined. We furthermore assume that all involved rule and instance graphs are typed over a common type graph (with inheritance).

A GT rule r can be divided into three different partitions: elements to be deleted, elements to be newly created, and application context.

$$\begin{aligned} r_{del} &= \{N_{del}, E_{del}\} \text{ with } N_{del} = N^L \setminus N^R \text{ and } E_{del} = E^L \setminus E^R \\ r_{new} &= \{N_{new}, E_{new}\} \text{ with } N_{new} = N^R \setminus N^L \text{ and } E_{new} = E^R \setminus E^L \\ r_{ac} &= \{N_{ac}, E_{ac}\} \text{ with } N_{ac} = N^L \cap N^R \text{ and } E_{ac} = E^L \cap E^R \end{aligned}$$

A necessary restriction is that nodes in N_{new} must never be typed as abstract nodes.

IV.3.4 Rule Application under SPO

Definition 12 (Matching of a rule)

The matching of a rule $r = (L, R)$ against a graph $host \in G_{DMMI}$ is a mapping of the left hand side rule graph:

$$\begin{aligned} match(r, host) = mt : L \rightarrow host \text{ an injective morphism with} \\ \forall n \in N^L : type_N(mt_N(n)) \in clan_I(type_N(n)) \end{aligned}$$

The injective matching used in this thesis is a restriction of Graph Transformations in general. When allowing for non-injective matchings, an element in a host graph may actually satisfy two roles in one rule. This can be a convenient feature to handle special situations (e.g., reflexive and non-reflexive transitions in a Statechart might be handled by the same rule). It can also incur unwanted matchings and conflicts³. Since recognizing potential problems with non-injective matches is not trivial (especially taking subtyping into account), we restrict our approach to injective matching only.

A matching might induce a set of edges to be deleted (if one of their anchoring nodes is being deleted):

Definition 13 (Deleted Edges)

The set of deleted edges de induced by a matching is defined as

$$de(match(r, host)) = \{e \in E^{host} \mid \exists n \in N_{del}^r : s_E(e) = mt_N(n) \vee t_E(e) = mt_N(n)\}$$

Note that this definition does not distinguish between edges which are explicitly and implicitly deleted.

³A conflict arises if a node in the host graph simultaneously matches nodes from N_{del} and N_{ac} .

Definition 14 (Derivation)

Applying a rule r to a host graph pre yields a derivation relation:

$$pre \xrightarrow{r} post \text{ with}$$

$$\begin{aligned} &pre, post \in G_{DMMI} \\ &\exists mt' \text{ a matching with } mt(r_{ac}) = mt'(r_{ac}) \\ &pre \setminus (mt(r_{del}) \cup_E de(mt(r))) = post \setminus mt'(r_{new}) \quad ^4 \end{aligned}$$

A graph derivation leaves the graph untouched apart from the parts to be either explicitly or implicitly deleted. The elements which are to be added to the graph need to form a new matching mt' in relation to the right hand side of the rule. Operationally speaking, one can also say that the application of the rule removes all nodes and edges in the image of r_{del} and all edges in de . Then new elements for the elements of r_{new} are created. For each element in r_{new} a new element in H is constructed which has the same type as its correspondent in R . The labels for new nodes are chosen by $new(\Lambda)$, the labels for edges are identical to the labels of edges in the type graph, as identified by the rule's typing. We express a concrete derivation textually as $G \xrightarrow{mt(r)} H$ where r is the name of the applied rule and mt is the matching function designating the occurrence of the rule in G .

IV.3.5 Negative Application Conditions

Rule application usually requires the presence of certain structures. There are situations, however, where the *absence* of a structure is of interest. Think, for instance, of a rule which moves some element to a buffer, provided no other element occupies the buffer yet. To check for such absent structures, *negative application conditions* (NACs) are defined for Graph Transformation rules in [HHT96]. A NAC is an extension of the left hand side of a rule. The semantics of a NAC is that the structure of the NAC must *not* be present in the context of the rule occurrence for the rule to apply. The effect of a NAC is demonstrated in Fig. IV.2. Rule r_1 requires a Buffer and will match on either b1 or b2 in the given host graph (matchings mt_1 and mt_2). Rule r_2 explicitly forbids the occurrence of an Element in the context of the matched Buffer and can thus be applied to the empty buffer b2 only. Negative application conditions are supported by the majority of GT tools and widely used in theory.

IV.3.6 Application and Consistency Conditions

The application of a rule may be subject to additional conditions which cannot (or at least not easily) be captured by the left-hand side of the rule. Examples include complex conditions on the state of attributes or values outside of the GTR's scope or the existence of unbounded structures (e.g., paths of arbitrary length).

Conditions may either be notated in the rule itself (e.g., PROGRES provides a special *path* construct), but often a textual logic notation external to the

⁴ \cup_E denotes the union for the edge component of a graph

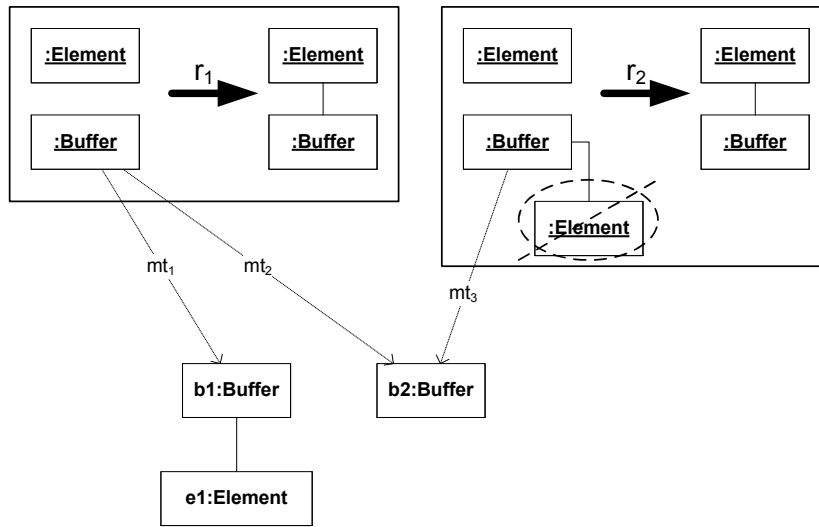


Figure IV.2: Example for the matching of Negative Application Conditions

rule is used (e.g., in [Sch96b]). Negative Application Conditions (as introduced in the previous subsection) are a special and very common case of application conditions.

An important task for application conditions is their deployment to guarantee adherence of the host graph to certain restrictions. If, e.g., the host graph represents an object configuration, the type level usually imposes multiplicity constraints or other restrictions which must not be violated. Using application conditions, one can ensure that the application of Graph Transformations Rules to a graph which is valid with respect to these constraints will always yield a valid graph again. In [HW95] a procedure is provided which allows encoding such application condition in SPO rules directly, using the notion of weakest precondition. Taentzer and Rensink [TR05] extend that work to apply to typed graphs with inheritance.

IV.3.7 Universal Quantification

Elements in the left hand side of a rule usually have an existential quantification, i.e., they need to find exactly one correspondent in the host graph for a successful matching. In many situations, however, a *universal* quantification is not only convenient but necessary. As an example think of a rule which flushes a buffer. If this buffer can contain multiple elements, the rule will have to match and delete all of them to achieve its aim (cf. Fig. IV.3). Universally Quantified Structures (UQS) in the left hand side of a rule will thus match to *all* elements in the host graph which fulfill the given constraints. In Fig. IV.3, the UQS `:Element` thus matches `e1` to `e4` in the hostgraph, as all of them are contained in the buffer. As displayed by the figure, universal quantification poses the problem that the functional character of the matching morphism cannot be retained.

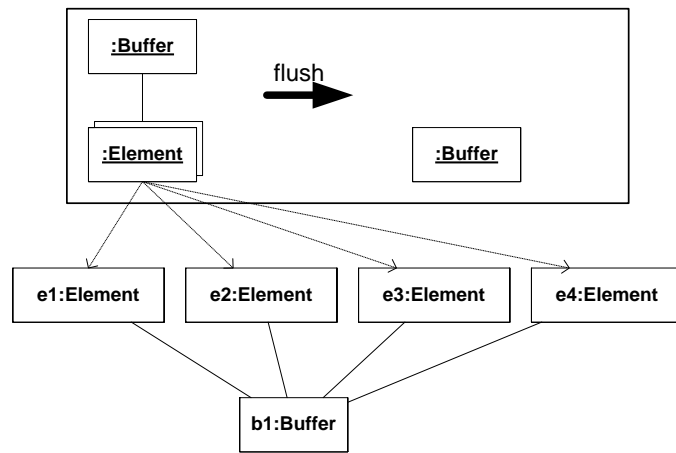


Figure IV.3: Example for the matching of universally quantified elements

Parallel Graph Transformation [Tae96, dLETE04] is a formal approach which can be used to express universally quantified structures in Graph Transformations. Instead of rules, PGT assumes the specification of *rule schemas*. A rule schema consists of the base rule (also called elementary rule), subrules, and interaction schemes which specify the possible connection between base and subrule. The process of *amalgamation* is used to produce a plain GT rule by gluing multiple instances of the base rule(s) with the subrule, thus providing a universal interpretation of the base rule's elements. The usual matching notion can then be applied to these amalgamated rules. Thus, parallel graph transformation circumvents the problems posed by universal quantification by using a pre-processing step⁵ which yields simple rules. Rule schemas are realized by the tools AGG [LB93] and ATOM3 [dLETE04]. Obvious drawbacks of the approach are a significant rise in the complexity of the rule notation and a conceptually unlimited number of amalgamated rules for a single rule schema. In practice, the amalgamation is either done on the fly when applying a rule or by specifying a maximal number of subrule occurrences. The concept of Graph Transformations with variables [Hof05] also allows for the formulation of UQS under the name of *clone variables*. Variables are also pre-processed to yield "plain" GT rules.

A second way to support universal quantification is the adoption of a different matching notion. Schürr studies this kind of matching in [Sch91, Sch96b] for the PROGRES language. PROGRES allows specifying elements with universal quantification directly in the rule graph. The matching notion then degenerates from a graph morphism to a general relation with special properties (cf. [Sch96b]). This degeneration has serious impacts on the underlying theory as the resulting structure is not an algebraic pushout anymore and rule applications are no longer invertible in the general case. The application of existing theory is hampered.

The use of universal quantification in Graph Transformation rules is usually re-

⁵It is in fact called a *two-level derivation* in [Tae92].

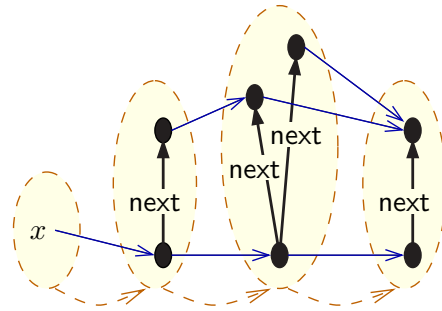


Figure IV.4: Example for a graph predicate (taken from [Ren04b])

stricted. Universally quantified structures may only be attached to an existentially quantified node with the semantics of a complete match in this element's context. In general, however, further characterizations of the matched element set and other constructions are possible which cannot be fully expressed within these restrictions. In contrast, Rensink studies graph predicates [Ren04b] in which a graph is not only extended by one layer of application conditions but by several such layers, each forming a logical negation to the one above it. Since negation of existential quantification yields universal quantification, it is shown that using such a notion of a graph, an expressive power equal to the of first order logic is achieved. While this technique is very powerful, the graph predicates produced by it are rather hard to read. The example in Fig. IV.4 specifies the condition that $\exists y : next(x, y) \vee \forall z : (next(x, z) \Rightarrow z = y)$. Moreover, the application of graph predicates for the use in Graph Transformations is not straightforward as a matching notion taking the multiple layers into account needs to be formulated.

A final issue is the manipulation of universally quantified elements. Deletion of such elements entails the deletion of all matches but, e.g., the creation of an edge to or from an universally quantified node could mean either the creation of a single edge (to one node of the set) or the creation of edges to all members of the set. Even more complicated is the creation of edges between universally quantified nodes: either a single connecting edge, the complete Cartesian product of nodes, or an injective/surjective/bijective and right/left total set of edges may be intended. Up to date no graph transformation approach takes these possibilities into account and provides distinguishing notations.

IV.4 Graph Transformation in DMM

The notion of graph used in DMM has been defined in Subsect. IV.1 and the succeeding notion of basic SPO Graph Transformations has been build upon this notion of graph. The overview in the previous subsection demonstrates that this basic notion can be extended in a number of ways. We now proceed to select the features useful in the scope of DMM.

- ◆ DMM *does* support Negative Application Conditions. NACs are widely

supported by theory and tools and provide required expressiveness to GT specifications.

- ◆ DMM *does not* support other application conditions. Application conditions are usually notated in textual format outside of the rule. Although they can help to fine tune the application of GT rules, they also extend the required formalization substantially, invalidating most theoretic results and diminishing the visual appeal of GT rules. Few tools support application conditions.
- ◆ DMM *does* support universal quantification, but only in a very limited way. We do not see the need to employ the full power of graph predicates. Combinations of rules can be used for the processing of more complex situations.

IV.4.1 Rules in DMM

Rules are visually represented in a special way in DMM. As laid out in Section II.2, the meta modeling approach of UML is about re-using the modeling notations of the UML in its definition. We already use UML Class and Instance Diagram notations for the presentation of graphs (cf. Tab. IV.1) and use UML Communication Diagram notations to represent Graph Transformation rules. The basic idea for this visual representation can also be found in the Story Diagrams notation used in Fujaba [FNTZ00, HZ01].

Table IV.2 provides an example for the correspondence of DMM rule concepts and their visual representation. In the first row we see the representation of basic rule constructs. Elements in r_{acc} (i.e., node a) are notated without special markup, elements in r_{del} (node b and edge type) and r_{new} (node c and edge edge container) carry the constraints {destroyed} and {new} next to their labels respectively. The signature of the rule (its name plus additional information) is located on the top left hand corner in a pentagon shaped compartment.

IV.4.2 Negative Application Conditions in DMM

We use Negative Application Conditions in rules as defined by Habel, Heckel, and Taentzer [HHT96]. Formally, an application condition (positive or negative) is an extension of the left hand side graph of a rule. We only use NACs in DMM, thus the following definition suffices for our purposes:

Definition 15 (DMM Rule with NAC)

$NAC(r) \subset G_{DMMR}$ a finite set with

$$\forall \hat{L} \in NAC(r) : L^r \subset \hat{L}$$

Each NAC is thus an extension of the left hand side of the underlying rule.

A matching $mt = match(r, host)$ satisfies its conditions, if for no $\hat{L} \in NAC(r)$ an extended matching $\hat{m}t : \hat{L} \rightarrow host$ can be found which is identical to mt for elements of the underlying rule's left hand side.

⁶Typing information is omitted in all examples.

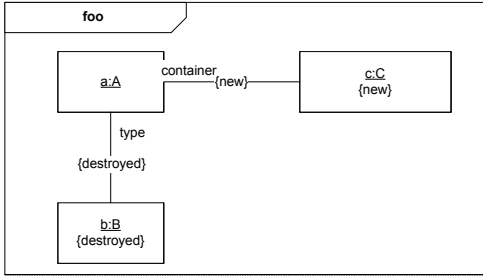
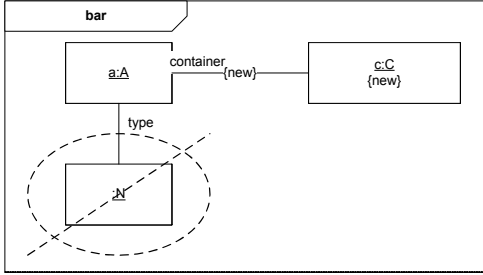
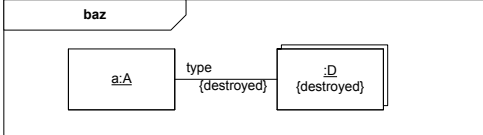
Example (conceptual) ⁶	Example (graphical)
<p>Rule <i>foo</i>=</p> $N^L = \{\alpha, \beta\},$ $l_N^L = \{\langle \alpha, \text{"a"} \rangle, \langle \beta, \text{"b"} \rangle\},$ $E^L = \{\langle \alpha, \text{"type"}, \bullet, \bullet, \beta \rangle\},$ $N^R = \{\alpha, \gamma\},$ $l_N^R = \{\langle \alpha, \text{"a"} \rangle, \langle \gamma, \text{"c"} \rangle\},$ $E^R = \{\langle \alpha, \text{"container"}, \bullet, \bullet, \gamma \rangle\}$	
<p>Rule <i>bar</i>=</p> $N^L = \{\alpha\},$ $l_N^L = \{\langle \alpha, \text{"a"} \rangle\}$ <p>(right hand side omitted)</p> <p>NAC:</p> $N^{\hat{L}} = \{\alpha, \pi\},$ $l_N^{\hat{L}} = \{\langle \alpha, \text{"a"} \rangle, \langle \pi, \bullet \rangle\},$ $E^{\hat{L}} = \{\alpha, \text{"type"}, \bullet, \bullet, \pi\}$	
<p>Rule scheme <i>baz</i>=</p> $N^L = \{\alpha\},$ $l_N^L = \{\langle \alpha, \text{"a"} \rangle\}$ $E^L = \{\}$ $R = L$ <p>UQS:</p> $N^{\hat{L}} = \{\alpha, \delta\},$ $l_N^{\hat{L}} = \{\langle \alpha, \text{"a"} \rangle, \langle \delta, \bullet \rangle\},$ $E^{\hat{L}} = \{\langle \alpha, \text{"type"}, \bullet, \bullet, \delta \rangle\},$ $N^{\hat{R}} = \{\}, E^{\hat{R}} = \{\}$	

Table IV.2: Correspondence of rule concepts and their graphical rendering in DMM

The notation for NACs (also adopted from [HHT96]) is a dashed ellipse enclosing the NAC's elements with a dashed line crossing out the NAC. Note that edges crossed by the dashed borderline are always part of the NAC. The second row in Tab. IV.2 provides an example, in which rule $bar()$ can only be applied to an A if no N can be matched in its context, connected via a *type* edge. Multiple NACs per rule are possible; by the definition given above each of them must hold for a successful match.

IV.4.3 Universal Quantification in DMM

Universally quantified structures (UQS) are supported in DMM rules in a very limited way. A UQS in DMM is based on specially marked nodes. A UQS is formed by such a node and all of its adjacent edges. If marked nodes are directly connected, they form a single UQS. Multiple UQS per rule are allowed. The marking of UQS nodes is done in DMM rules by the notational element *multinode* of UML Collaboration Diagrams. In row three of Tab. IV.2 we find an example in which the node :D is notated as a multinode and thus the UQS is formed by node :D and the adjacent edge.

Conceptually, a UQS matches the maximal set of elements which fulfill its conditions. Technically, we adopt the idea of rule schemes (cf. [Tae96]) which can be 'unfolded' to a number of simple rules. Unfolding is a process which takes the core rule and the different UQS as its input and produces simple GT rules as its output which are formed by combining the core rule with multiple copies of the UQS. This unfolding process is illustrated in Fig. IV.5 which shows a rule schema $r3$ and two unfolded rules $r3'$ and $r3''$. The universally quantified node in $r3$ is supposed to match to *all* nodes of type U in the context of a . The unfolding works by combining a number of duplicates of the universally quantified node together with a NAC to ensure the completeness of the match. Note that this construction works due to the restriction to injective matching. Such an unfolding will potentially produce an unlimited number of rules and thus needs to be restricted to "expected" numbers of duplicates of U . In the example in Fig. IV.5, a maximum of two U s is being handled by the unfolded rules. Multiple universally quantified elements increase the number of unfolded rules even more as different nodes are unfolded independently. Note that universal quantification implies the existence of at least one match.

As laid out in Subsect. IV.3.7, manipulation of universally quantified elements is not trivial. To avoid these problems, we disallow the use of universally quantified structures in r_{new} , i.e., universally quantified elements are either preserved or deleted (resulting in a deletion of all matched elements in the host graph).

The formalization of UQS is given in Subsection IV.6.7.

IV.5 Controlling Graph Transformations

The application of a single Graph Transformation rule usually happens in the context of a Graph Transformation System.

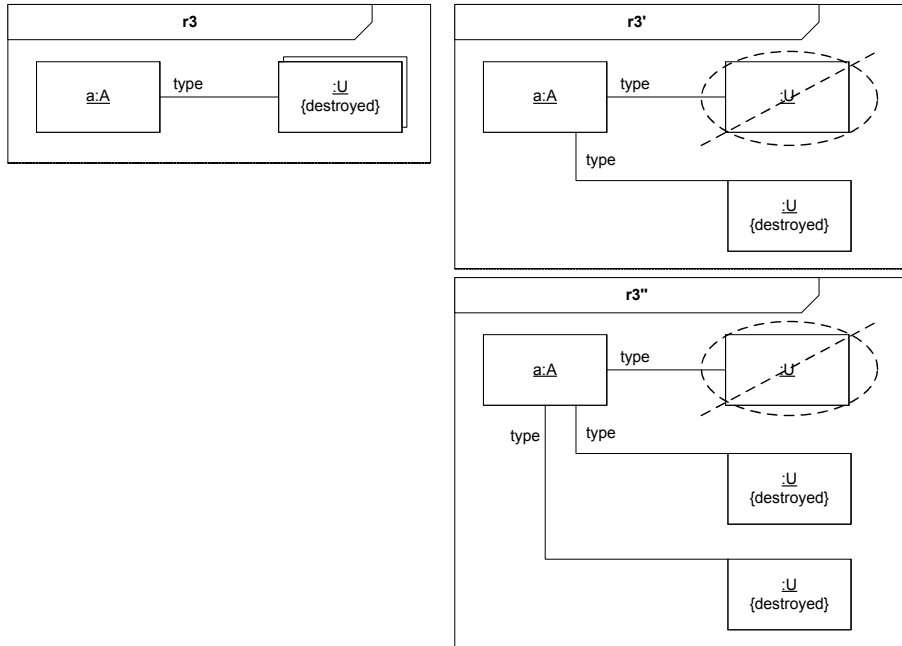


Figure IV.5: Examples for unfoldings of a rule schema

Definition 16 (Graph Transformation System)

A Graph Transformation System is formed by a set of Graph Transformation rules *GTRs* and a start graph G_0 .

The idea behind a Graph Transformation System is to apply rules from the given rule set repeatedly to the underlying host graph which is manipulated by the rule applications. The extension of a Graph Transformation System is thus the set of derivation sequences obtained by applying its rules on G_0 .

Definition 17 (Derivation Sequence)

A derivation sequence $G \xRightarrow{*} H$ is a sequence of derivations $G \xRightarrow{ra_1} G_1 \xRightarrow{ra_2} G_2 \dots G_{n-1} \xRightarrow{ra_n} H$ with ra_i being rule applications, i.e., matchings of rules.

A Graph Transformation System thus defines a set of such derivations with all applied rules stemming from *GTRs*

From an operational point of view, this general notion of Graph Transformation Systems entails two kinds of non-determinism: The rule to be applied next in the derivation sequence is chosen non-deterministically as is its matching in the host graph (if multiple such matchings are possible). While this construction is very simple and powerful, it has some rather undesired properties from a practical point of view. Since a single rule application is the only available unit of synchronization, complex graph manipulations need to be expressed as a single rule. This situation leads to complex rules and large rule sets as each variation of a possible scenario requires a separate rule.

From a practical point of view it is beneficial to have the ability to split complex manipulations into smaller parts (i.e., separate rules) and to control the application of these parts to some degree. This ability simultaneously reduces the size of the rules (by splitting large rules) and the number of rules in a set (as variations can be expressed by recombining existing parts). Thus, several such controlling mechanisms have been proposed in the literature. We provide an overview of these different approaches here as (especially for practical GT approaches) these control constructs form their main distinguishing element.

IV.5.1 Priorities and Layers

By assigning *priorities* to rules in a graph transformation system, one can control their application, as rules with higher priority will be checked for applicability first. Priorities have been studied by Litovski and Metevier [LM93, LMS95] and have been implemented, e.g., in the GROOVE tool set [Ren04a]. Similar to priorities is the concept of *layered graph grammars* [RS97]. Layered Graph Grammars are supported by the AGG tool [TB94]. Both layers and priorities retain theoretical properties of rule sets, thus allowing for analysis, but are rather unwieldy to handle in practice. Unexpected changes may cause a re-assignment of priorities throughout the rule set. Additional measures must be taken to avoid conflicts between rules of identical priority.

IV.5.2 Triggers and Invocations

Triggers were introduced in the GOOD system which applies Graph Transformations to databases [GPTdB93]. A trigger is a special kind of node created in the host graph which determines the next rule to be applied. Triggers can thus be regarded as an invocation of another rule. The same mechanism is employed in the GRRR system the primary concern of which is graph drawing [Rod98, Rod00, RV00]. GRRR extends the original trigger node idea by allowing multiple triggers to be present at the same time in the graph and applying a LIFO strategy to their processing. Thus, an invocation stack is simulated. Triggers keep control local to a rule and allow for the formulation of sequences (by direct invocation) and loops (by recursive invocations). A disadvantage can be seen in the fact that in the GRRR approach the organization of the invocation stack has to be encoded in the rules themselves and that multiple NACs have to be employed just to ensure a consistent handling of this stack by all rules. Going beyond the GRRR concepts is PROGRES which allows not only for invoking other rules but also caters for passing information to this rule application in the form of parameters.

IV.5.3 Transformation Units

Transformation Units have—in different specifications and under different names—been proposed by a number of authors. Common to them is the introduction of an additional control level in the GTS which constrains its possible derivation sequences. Usual control conditions include sequentialisation of rule

sets, fixpoint iteration of a single rule, or graph coverage by rule matchings. The general case of such control conditions and their effects on different graph transformation approaches is studied by Kuske [Kus00, AEH⁺96]. Küster uses a concrete instance of transformation units in [Küs04] to encode a language translation by Graph Transformations. His choice of control constructs preserves confluence properties of the rule set [HKT02]. Varró [Var02] encodes control conditions in control flow graphs which are expressed as a Statechart-like formalism. Habel and Plump show that with additional control conditions in the form of sequential composition and iteration, graph transformation systems become a computational complete language [HP01]. A practical implementation of this result is Steinert and Plump's work towards a Graph Programming Language [Ste03, PS04].

Additionally, transaction properties may be applied to transformation units (or other groupings of rules). These are especially interesting if additional consistency constraints are used in the approach. Such constraints may be violated during a transaction but have to apply again before committing the transaction's result. PROGRES supports transactions and in [HMTW95] the transaction concept is formalized within the SPO context for the AGG system.

IV.5.4 Programmed Graph Transformations

Programmed Graph Transformations take the concept of explicit application control even further as the application of Graph Transformations is embedded in completely programmed control flow, including conditional, branching, and looping constructs usually found in programming languages. The most elaborated example here is PROGRES [Sch90, Zün95, Sch95, SWZ95, SWZ99, MSW00, BR04] which provides very rich facilities to control the application sequence of Graph Transformations. In fact the facilities are so rich that PROGRES can be rather seen as a (textual) programming language with Graph Transformations in it.

The FUJABA system [fuj] also provides rich facilities for programming the graph transformation application but specifies this control in a graphical notation called Story Diagrams [FNTZ00]. Providing similar facilities to classical programming languages, these approaches are very powerful and can specify even industrial systems in a convenient and rather compact way but they lose the analyzability of the base formalism. The claim of intuitive understanding is also weakened by the introduction of the additional control structures.

IV.6 Control in DMM—The Mechanism of Rule Invocation

Our choice between these different proposed control mechanisms for Graph Transformations is guided by the criteria set out at the beginning of this chapter. Especially important is understandability. We feel that the notation for application control should not introduce another level of specification. This greatly decreases understandability, as the reader has to combine both specifications

mentally to grasp the overall meaning. The basic appeal of Graph Transformations is lost in such an approach. This concerns Transformation Units and (especially) programmed Graph Transformations. Priorities and triggers can be notated in the rules themselves and are thus better in this regard.

Analyzability also decreases with the richness of control features. Especially for Programmed Graph Transformations, the application of theoretical results is very hard. Transformation units and the simpler mechanisms of priorities and triggers are more amendable to analysis.

Tool support can be seen from different angles: For programmed Graph Transformations, rich environments exist, which allow the application of these approaches in a practical context. This close connection between tool and formalism also makes the approaches very proprietary. No other tools exist to process such formalisms.

Adequacy finally rules out priorities as they do not scale to realistically sized problems and cannot deal with the changes inevitable in realistic problems.

Combining these judgments we find the approach of triggers to be a (potentially good) compromise between sparseness of features (enabling analysis and general tool support), fine grained control and good maintainability.

From the viewpoint of understandability, triggers also present another advantage: We assumed OO knowledge in our target audience. In the OO view, control flows are realized by having one behavior (operation) invoke other behaviors. Employing this idea to control the application of rules seems a very promising approach to us.

The practical elaboration of the basic trigger idea in the GRRR system is nonetheless lacking in several ways. Neither is the idea fully thought through, nor is a proper formalization given. The approach furthermore contains severe applicability problems as a lot of auxiliary nodes clutter the rules and the underlying graph.

In this section we thus introduce a novel approach to control the application of Graph Transformation rules called *Rule Invocation*. Rule invocations as the main structuring mechanism of Dynamic Meta Modeling allow the modeler to specify complex graph manipulations in a convenient manner by distributing the manipulation over multiple rules. Control in this approach is notated in the rules which retain their intuitive character. This localized control also allows for an OO-like structurization of complex behaviors. In contrast to the basic idea of triggers [Rod98], invocations are established as a first class language feature for Graph Transformation rules. Invocations promise a high understandability (especially from an OO mindset) and are simple enough to be processed by standard GT tools (as demonstrated in Chapter VIII).

To get an intuition on the way invocations work let us regard a small example: Suppose we have a `ListManager` which knows a number of `lists` each of which lists `Elements` of a different `ElementType`. Lists can furthermore be organized according to different ordering schemes. Fig. IV.6 provides two DMM rules without invocations describing the main operation of a `ListManager`, namely the integration of a new element. The left hand rule in Fig. IV.6 describes this integration for the case of a first-in, first-out (FIFO) list which contains at least

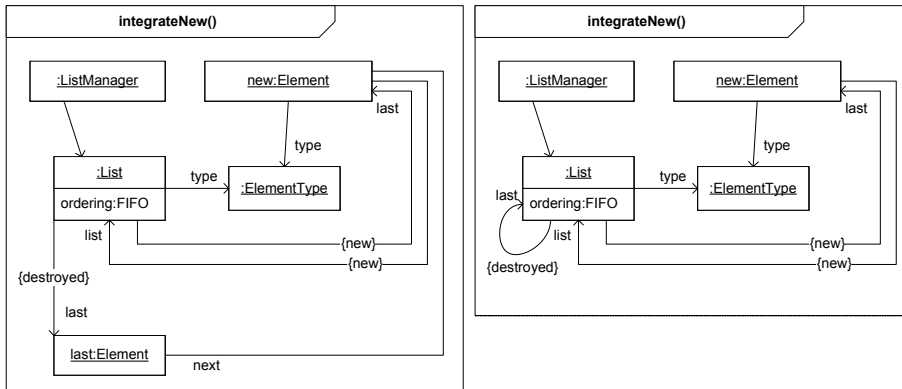


Figure IV.6: Example specification without invocations

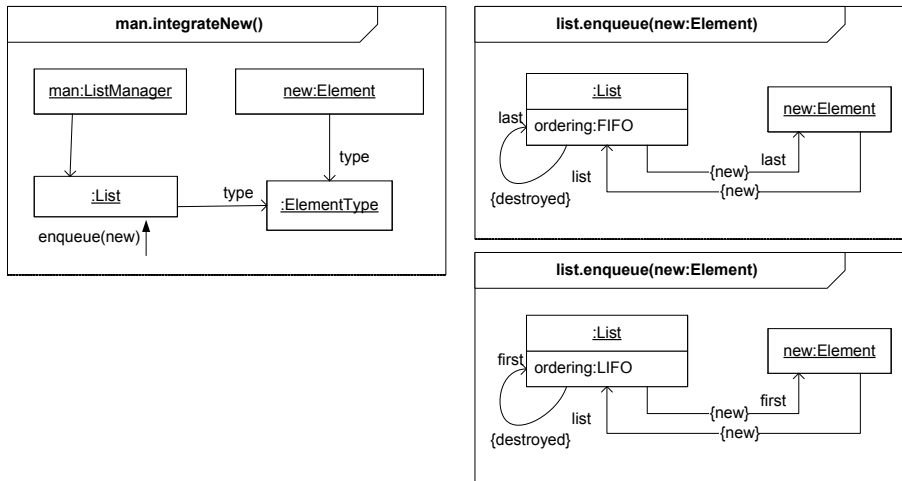


Figure IV.7: Example specification with invocations

one element. We can observe the selection of the correct list (by the matching of the common `ElementType` node) and the correct integration of the new element into the list structure (by assigning the necessary pointers). The right hand side rule displays the same operation for an empty FIFO queue.

We can observe several properties in this example: Each rule handles one specific scenario. Each additional ordering scheme would add (at least) two new rules and the introduction of further crosscutting features (e.g., upper bounds for lists) would multiply the total number of rules. Furthermore, each rule is global in handling all aspects of a scenario. Both rules perform the selection of the correct list as well as the integration of the element according to the ordering scheme of the list. Conversely, the manipulation of a list's ordering structure is embedded in a multitude of rules, impeding the integration of changes to this structure. Finally, the rules are already rather complex and the introduction of additional features would rapidly increase this complexity.

In contrast, Fig. IV.7 presents the same example using rule invocations. Here, we see three rules. The left hand side rule presents the rule for the list manager. Its sole responsibility is the selection of the correct list. It then invokes the rule `enqueue` on the list it selected. How exactly this enqueueing works is of no concern for the listmanager rule. The two rules on the right hand side of the figure specify different scenarios for the execution of the `enqueue` invocation. The upper rule specifies the enqueueing in an empty FIFO list, the lower the enqueueing in an empty LIFO list. Thus, different scenarios can be handled by different rules with the invoker being unaffected by the multitude of detailed differences.

We can observe several properties of DMM rules in this small example:

- ◆ The rules become individually simpler to understand as each rule only has one responsibility and carries out the necessary manipulations for its aspect. The rules thus employ less elements, making them easier to grasp.
- ◆ The invocation mechanism is easily understandable from an OO point of view as it closely resembles the way an OO program would structure its behavior.
- ◆ The notation of the control structure smoothly integrates in the rule presentation.
- ◆ Each rule is concerned only with a single aspect. Internals of the pointer structure of the lists need not be known in the listmanager rule. The rule set becomes more maintainable by this separation of concerns.
- ◆ Adding new list ordering schemes will not increase the number of rules exponentially but will only require the addition of few rules. No other rules must be changed for this introduction.

Technically, the example raises some questions which we will answer in the following subsections:

- ▷ *How do invocations embed into a rule? Can there be multiple invocations per rule? Can information be passed as parameters?*

All of these questions concern the invocation notation. Subsection IV.6.1 provides the details of embedding invocations in DMM rules.

- ▷ *How do invocations apply? Which rule exactly is applied to answer an invocation?* The application mechanism does not directly call a determined rule but issues a request for the application of a rule described by the elements of the invocation. How this request is issued is the topic of Subsection IV.6.2, how rules specify which requests they can answer is described in their signature (Subsection IV.6.3), and how a invocation request and a signature are matched is described in Subsection IV.6.4.

- ▷ *How do invocations interact with the usual (non-deterministic) rule application?*

This question goes right to the heart of the invocation mechanism as invocations allow for a combination of the loose control of rule-based systems with the tight control of invocations as known from OO. The (short) answer to this question is that all DMM rules get partitioned into two groups: *Small-step* rules which can *only* apply when they are invoked

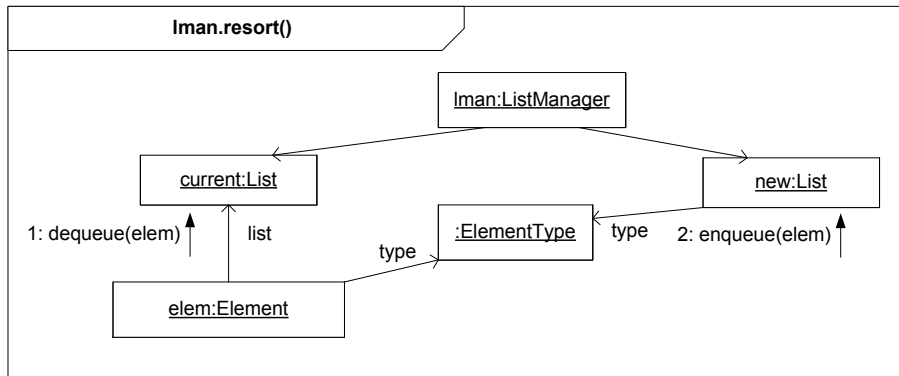


Figure IV.8: Example for the use of sequence numbers

by another rule and *big-step rules* which can apply non-deterministically. Subsection IV.6.5 provides details on these rule types.

- ▷ *The example shows how graph manipulations can be distributed into other rules, can matching conditions also be distributed in this way?*

Yes, they can. There is a third kind of rule called the *precondition rule*, which allows for influencing an invoking rule's match. Subsection IV.6.6 provides details on precondition rules.

- ▷ *But how does it all work together?*

In Subsection IV.6.7 we provide the complete formalization of the invocation notion for Graph Transformations and its impact on the applicability of rules.

IV.6.1 Rule Invocation

Invocations of rules are the main control mechanism of DMM. The basic idea is that a rule can specify one or more invocations. If a rule specifies such invocations it influences the set of succeeding rules in a derivation sequence, i.e., the invoked rules must apply (directly) after the invoking rule has been applied.

The visual representation of an invocation is an arrow pointing to a node (its so called target node). The label at the arrow contains the name of the rule to be invoked and parameters notated in parenthesis. If multiple invocations are used in a single rule, a sequence between these invocations can be fixed using sequence numbers. This visualization resembles the notation used in UML Communication diagrams for messages (cf. [Obj04], p.561)⁷.

Figure IV.8 presents an example for the use of rule invocations with sequence numbers. Here, a `Listmanager` re-sorts an `Element` after it changed its type. The rule invokes two other rules in a determined sequence to perform its task. First, the `Element` is dequeued from its current `List`, then it is enqueued in the new

⁷A difference is that in Communication Diagrams messages always need to run along an association while rule invocations may freely be attached to their target node.

List. The listmanager rule only coordinates this behavior by using invocations with sequence numbers.

Formally, invocations extend the right hand side of a rule

Definition 18 (Rule with Invocation)

A rule with Invocations r_{Invoc} consists of a rule $r : L \rightarrow R$ and an invocation relation $inv = \{\langle seq, name, args \rangle \text{ with } seq \in \mathbb{N}, name \in \Lambda \setminus \{\perp\}, args \in 2^{N^R} \setminus \{\}\}$.

The three elements of an invocation represent sequence number, rule name, and parameters, respectively:

Sequence Number A *sequence number* indicates the (partial) order of invocations if multiple invocations are used in a rule. A value of 0 indicates a not explicitly specified sequence number. Sequence numbers define the way the DMM system must process rule invocations. Invocations with a sequence number of 0 can in fact be processed in arbitrary order. A modeler should make sure that no interdependencies exist between rules invoked without explicit order. An executing system may place these rules anywhere in the execution order and is not bound to try all possibilities (i.e., ordering of these rules is arbitrary but not non-deterministic).

Rule Name The rule to be invoked is identified by its signature, the main part of which is its name.

Parameters The remaining part of the invocation is formed by a set of nodes of R . These nodes represent the information to be passed as parameters to the invoked rule. The first parameter always represents the target node of the invocation. There is always a target node for an invocation. While the sequence number restricts the order of invoked rules, the parameters and the target node restrict the possible matchings of the invoked rule in the host graph by fixing its matching for some elements.

IV.6.2 Applying Invocations

A special case worth pointing out is that invocations can be used to construct recursive loops. This allows processing unbounded sets of elements in a convenient way. An example is provided in Fig. IV.9. These rules specify the reversal of a list ordering. It starts with the rule for `list.reverse` which triggers the process and takes care of changing the `first` to the `last` element. To actually reverse the pointers between all elements, it calls the rule `element.reverse`. The right hand side rules in the figure specify this operation as a recursive loop. The upper rule specifies the processing of an element which is not yet the last in the list. It re-attaches the necessary pointers and calls itself recursively. If a situation is encountered in which there is no more next element (i.e., the old last element is reached) the lower rule applies and forms the recursion end as it invokes no other rules.

Several technical details need to be clarified to understand the mechanism of invocations correctly:

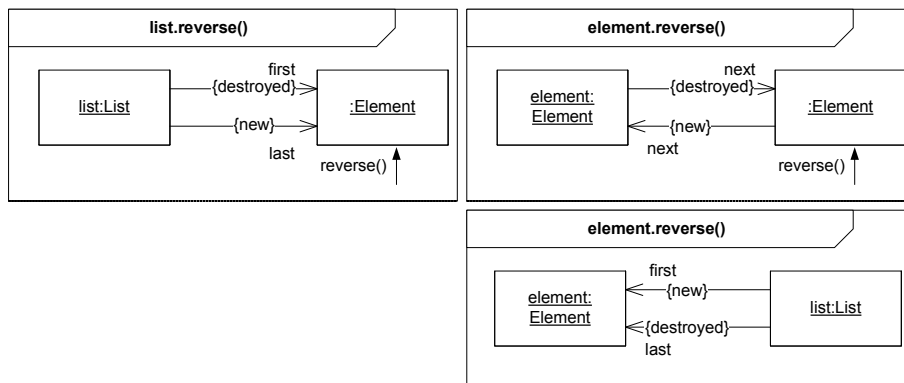


Figure IV.9: Example for rule invocations

- ◆ Invocations do apply *after* the manipulations specified in the rule's body itself. Thus, invoked rules extend the effects of the invoking rule.
- ◆ "Deep" invocations have precedence, i.e., if a rule invokes two other rules in sequence and the first of these rules also specifies an invocation, this invocation is processed first before continuing the sequence of the originally invoking rule.
- ◆ For every invocation there always *must* be a fulfilling rule application. If an invocation cannot be fulfilled by any rule of the system, the invocation is said to be *failed*. This results in the invoking rule being considered as failed, too. Transitively, a whole invocation hierarchy can be considered to be failed. Thus, even though a rule itself might have been applied successfully to the host graph, it might still fail due to the failure of its invocations.

IV.6.3 The Signature of a Rule

An invocation in a rule is a request for some other rule to apply. Symmetrically, rules need to be extended with information about the invocations they will answer to. This information is called a rule's *signature*. The signature of a rule consists of four parts:

Context Node The context node is the first element of a rule's signature. Technically, the context node is only one parameter for possible invocations of a rule. From an object-oriented point of view, one can also say that the context node (or rather its type) *owns* the behavior expressed in the rule⁸. The context node thus has a special relevance. Technically, the context node is referred to by name only (cf. Fig. IV.9), however, keeping the object-oriented interpretation in mind, we recommend choosing the name of the context node in a way as to easily allow for determining its type (as demonstrated in Fig. IV.9).

⁸A point of view which will be emphasized if these rules are employed in combination with a semantic domain meta model, see Chapter V.

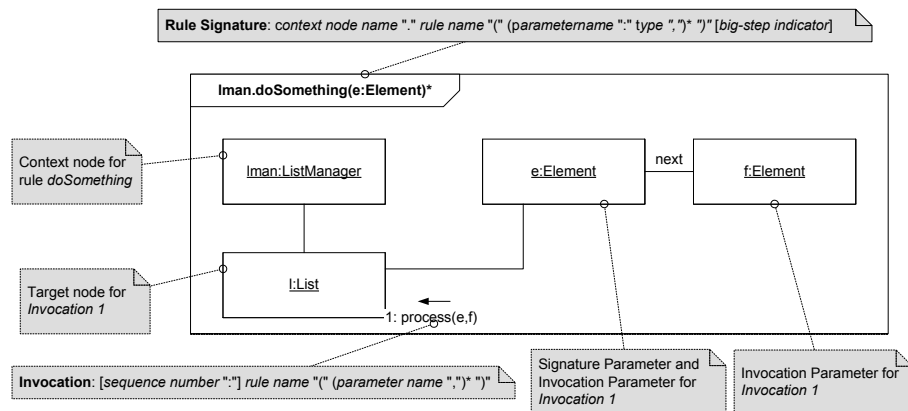


Figure IV.10: Illustration of the differences between signature and invocation in a rule

Rule Name The rule's name is the second part of the rule's signature and the only technical constraint posed to it is that in general it must not begin with the sequence "P_" (as this sequence identifies premise rules).

Parameters The list of Parameters is the third part of the rule's interface. Each Parameter is specified as name, colon, type and needs to appear as a node in the left hand side of the rule. If the rule is invoked, the nodes passed as parameters already have fixed matches in the host graph. Parameters in the rule's signature are separated by commas, the whole set is enclosed in parentheses. Empty parenthesis "()" signify the empty list of parameters.

Big-Step indicator The last part of the rule's signature may be formed by a "*" which signifies that the rule belongs to the group of big-step rules (see below).

It is especially important to note that there is no constraint that a rule's signature must be unique in the rule set. In fact multiple rules with identical signatures are used to handle different variants of a general behavior. In Fig. IV.9, e.g., the general behavior is the reversing of an element's pointers and the two rules `element.reverse` (with identical signature) handle different cases of this general behavior.

While signatures look rather similar in their structure to invocations, they form a strictly different concept. Fig. IV.10 illustrates both concepts in comparison for an example rule. The signature of the rule provides information *about the rule* itself. The invocation specifies information *about another rule* which should be applied next. Parameters of the signature are (in programming language terms) *formal parameters* while the parameters passed with an invocation are *actual parameters*. As both kinds of parameters use names to reference the relevant nodes, such nodes must not be anonymous.

IV.6.4 Invocation Fulfillment

We have now introduced both the syntactic structure of invocations (requesting a rule application) and signatures (offering information about a rule). We now define when a signature *matches* a given invocation. Three conditions must be satisfied for a rule to fulfill a given invocation:

- ◆ The name of the rule and the name given in the invocation must be identical.
- ◆ All parameters (including the target/context node) must match. A rule's parameter matches an invocation's parameter if the former's type is equal to or a subtype of the latter's type. For instance, an invocation `drive` on a node of type `Vehicle` can be fulfilled by a rule `van.drive`, if `Van` is a subtype of `Vehicle`. The order of parameters must be retained between invocation and signature.
- ◆ The rule's left hand side must match taking the passed parameters into account.

The application of a rule in the above enumerated circumstances *fulfills* an invocation. A rule application can fulfill at most one invocation.

IV.6.5 Small-Step and Big-Step Rules

The notion of invocation fulfillment raises the question, whether a rule must fulfill an invocation (i.e., can only be applied if it is invoked) or whether it retains the original rule-notion of applying whenever its left hand side matches. In DMM this distinction is made for every rule. A rule can either be a *small-step rule* or a *big-step rule*. The distinction between both rule kinds is made visible by big-step rules having a "*" at the end of their signature.

Small-step rules are rules which can only be applied to the host graph if another rule invokes them. Thus, small step rules are much like procedure definitions which supply functionality to a main program. Small-step rules may themselves invoke other rules. This kind of rule usually forms the bulk of a DMM rule set.

Big-step rules are rules which can be applied to the host graph without fulfilling an invocation. Thus, big-step rules behave like usual graph transformation rules. The application of a big step rule may trigger a whole hierarchy of rule applications interconnected by invocations. Since each of these rule applications may fail the original big step rule, the application of a big step rule is not considered to be successful until all invocations have been fulfilled. Big-step rules thus begin a kind of transaction process. To avoid unwanted interferences, only one such transaction may be active at any time in a DMM system. A big-step rule can thus not match until the previously applied big-step rule (and all of its invocations) have finished executing.

Effectively, the execution of a Graph Transformation System with Invocations thus comprises alternating phases: Either no invocations are open, then the set of big-step rules can freely match in the graph, or an invocation has been issued and is not yet fulfilled, then the next (small-step) rule application must fulfill

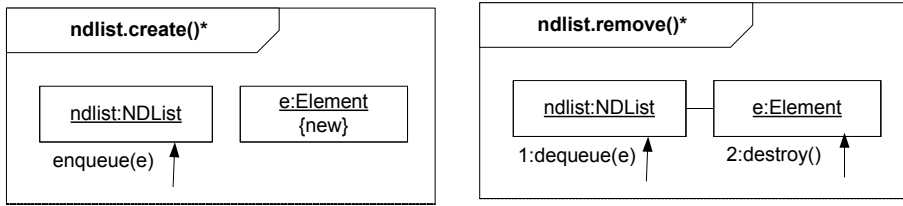


Figure IV.11: DMM big-step rules for the NDList example

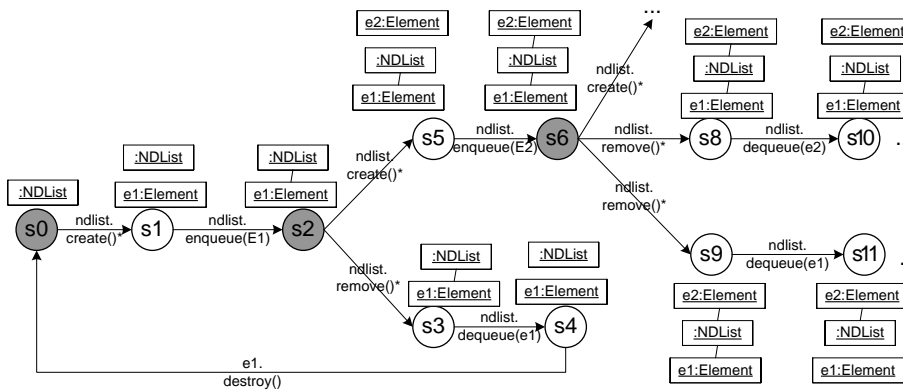


Figure IV.12: Illustration of derivation sequences of the NDList example

this invocation. Note, however, that this fulfillment might also include non-deterministic choices as different rules may be able to fulfill the invocation and the occurrence of the rule in the host graph may not be completely determined by the passed parameters.

Figures IV.11 and IV.12 illustrate this effect. Fig. IV.11 provides two big-step rules for a non-deterministic list (NDList) which creates and destroys its elements randomly. Both of these behaviors can occur spontaneously, thus these are big-step rules. The small-step rules enqueue, dequeue, and destroy are invoked by the big-step rules. These rules are omitted here as their content is trivial. Fig. IV.12 illustrates possible derivation sequences of this specification. Starting with an empty ndlist (State s0), only the rule ndlist.create()* can apply. States without open invocations are shaded grey. In state s1 only the application of ndlist.enqueue() is possible since this rule has been invoked. In state s2 there is now the choice, whether the buffer will destroy the created element e1 or create yet another element. In state s6 this choice is extended further as ndlist.destroy()* may match on either element (e1 or e2). Note how big-step rules only apply to grey states (no open invocations) and small-step rules only to white states (open invocation).

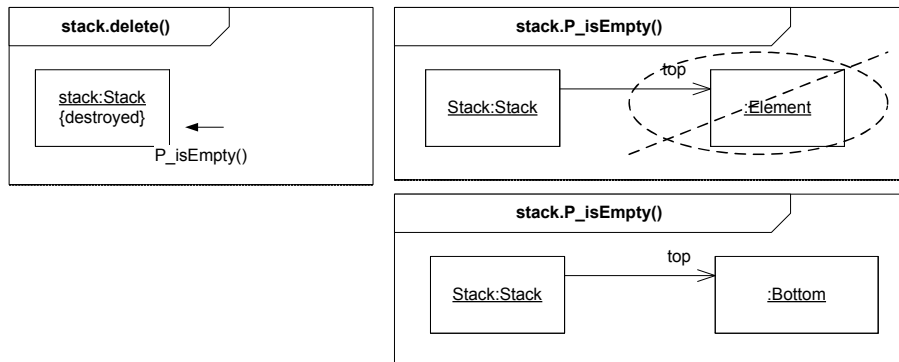


Figure IV.13: Example for the use of premise rules

IV.6.6 Premise Rules

Invocations are usually carried out after the application of the invoking rule, thus extending the invoking rule's effects on the host graph. To allow for a similar decomposition of a rule's application conditions, the concept or *premise rules* is introduced. Invoking a premise rule extends the left hand side of the invoking rule. To distinguish premise rules from usual rules, their name always starts with the prefix "P_". We assume these labels to form the set $\Lambda_P \subseteq \Lambda$.

The advantages of using premise rules are similar to the advantages of small-step rules: On the one hand they allow for less complex rules (by splitting complex left hand sides) and less complex rule sets (variations of a scenario can elegantly be covered by having different premise rules cover the different cases). On the other hand, the use of premises allows for re-use and easy maintenance in the OO sense. If a number of rules depends on a certain fact (e.g., different elements need to check for the availability of a resource) this fact can be encoded in a premise rule which all of the rules invoke. If one needs to change the formulation of this condition later on (e.g., there are additional constraints to check) one can integrate this change by adapting only the premise rule instead of adapting all rules relying on this fact.

As an example for the use of premise rules regard Fig. IV.13. Here, the rule `stack.destroy` depends on the premise rule `P_isEmpty`. Two variants of this rule exist: Either the stack contains no element at all or a dummy element is used to indicate an empty stack. Two rules for `stack.P_isEmpty` handle these different situations. Thus a stack may be destroyed in both of these situations.

Technically, premise rules are treated rather differently from other rule invocations. Since they influence the matching of the invoking rule, they have to be matched together with it. To achieve this effect, premise rules apply as separate rules (i.e., they do not produce their own steps in a derivation sequence) but they are integrated in the matching of the invoking rule. This entails some restrictions on premise rules: Premise rules are always identity rules, i.e., $L = R$ and they may not invoke small-step rules. They may, however, invoke other premise rules. NACS may be used in premise rules, but no UQS. In the invoking rule, however, premise rule invocations may be targeted at UQS nodes.

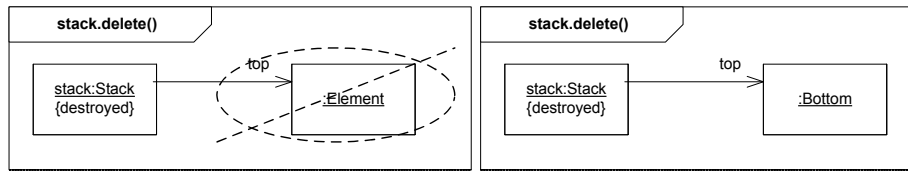


Figure IV.14: Amalgamated rules from the example in Fig. IV.13

To achieve the desired effect, we use the technique of rule amalgamation (cf. [Tae96]) to combine a premise rule with its invoking rule. As with the unfolding of UQS, the amalgamation process takes a complex specification (a rule with the invocation of premise rules and several premise rules to fulfill this invocation) as its input and produces a set of plain GT rules in which the contents of the rules have been integrated. The results of performing the amalgamation for the `stack.destroy` example are presented in Fig. IV.14. We can see that the premise invocation has been resolved by creating two different versions of `stack.destroy` which integrate the information of the different premise rules respectively.

IV.6.7 Formalization of DMM Systems

To formally capture the mechanisms of rule invocation a number of previous definitions must be extended. The most important basic distinction is that between *rule* and *rule schema*. We have already indicated that both the concept of UQS and the concept of *premise rules* go beyond the capabilities of a rule and must be handled by a pre-processing step. Thus, what we colloquially call a DMM rule is technically a rule schema which visually combines a number of overlapping but technically distinct specification parts of a rule schema.

Fig. IV.15 illustrates this idea. On top of the figure we see a DMM rule schema in the previously introduced visual notation. This example rule schema combines all of DMM's features. The five rule graphs below this combined representations show how the visually distinct elements of `a.foo()` indicated their membership to different parts of the specification. The most important part of the rule schema is its core rule (given separately for the left and right hand side). We can explicitly see the deletion of node `e:E` and the creation of node `:F` (both with their respective edges). We can also see that the invocations of the rule schema are an information which belongs to the right hand side of the core rule. The Universally Quantified Structures (node `:B` and its adjacent edges) are defined as a separate rule which contains and extends the core rule. As UQSs may be deleted, UQS extensions also have both a left and a right hand side. Note, that the premise rule invocation is formally an information belonging to the left hand side of a rule (in this example the left hand side of the UQS extension of the core rule). Finally, the NAC is an extension of the left hand side of the core rule. No right hand side is required here as NAC elements may never be matched, let alone manipulated.

From the definitions in Sect. IV.3 we know how the core rule and NACs are matched. As of yet unformalized is the effect that UQS, premise, and small-step invocations have on the matching of a rule and the ensuing derivation

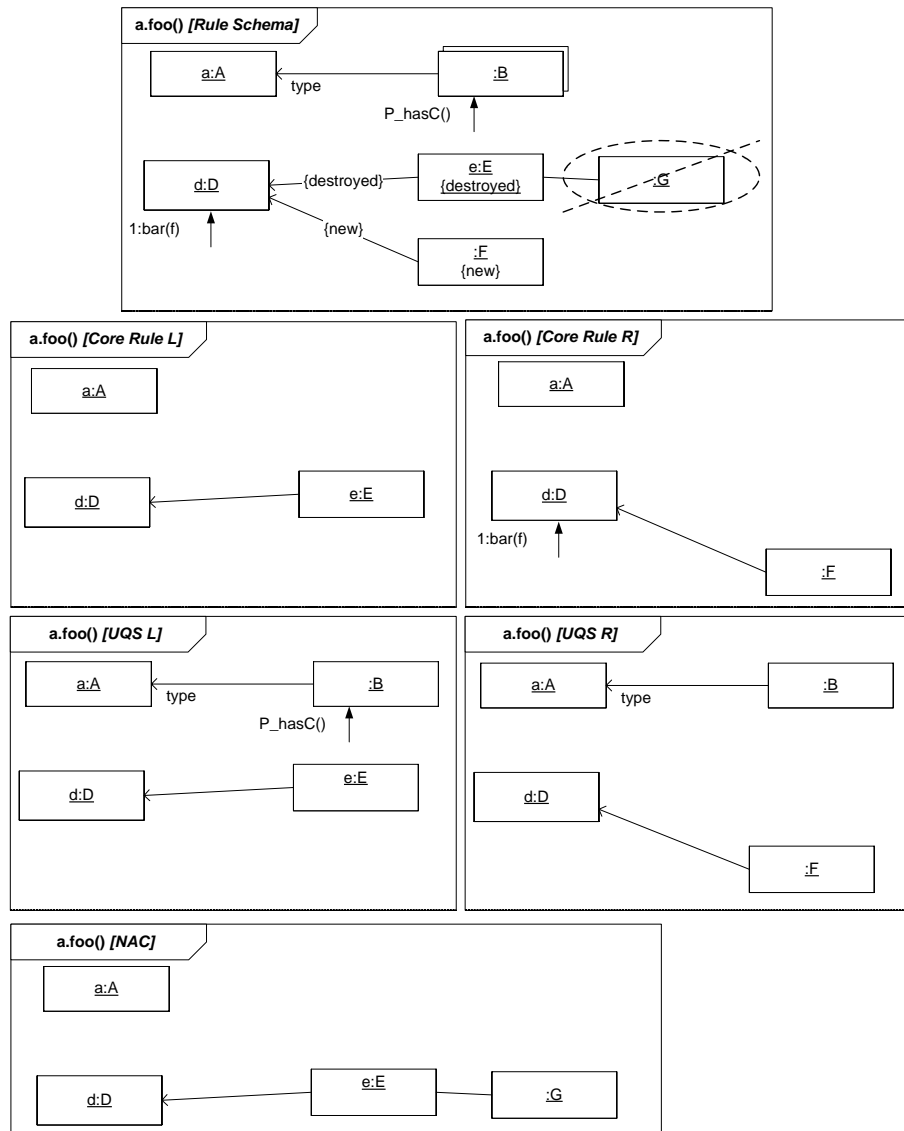


Figure IV.15: Illustration of the different parts of a rule schema

sequences. In the following definitions we will provide this information in 3 distinct steps:

- ◆ *UQS* and *premise rules* are unfolded, i.e. a rule schema is processed to yield a set of final DMM rules (Definitions 19 ff.).
- ◆ The application of a single final rule has to take the *core rule*, *NACs*, and invocations into account (see Definitions 27 ff.).
- ◆ The formalization of a DMM system finally ensures that the control encoded in invocations is properly respected in a derivation sequence (Definitions 31 ff.).

The notion of a rule is extended with structures to capture additional information about the rule itself (its name and parameters), its NACs, and the invocations which are specified in the rule.

Definition 19 (DMM Rule with Invocations (final rule))

A DMM rule with invocation is a six-tuple:

$$r_{DMMInv} = \{ \langle L, R, NAC, inv, Params, name \rangle \text{ with} \\
L, R \in G_{DMMR} \text{ usual rule graphs,} \\
NAC \subseteq G_{DMMR} \text{ a set of negative application conditions,} \\
inv \text{ an invocation relation as introduced in Def.18,} \\
Params \subseteq N^L, \\
\forall p \in (Params \cup inv.args^9) : l_N(p) \neq \bullet, \\
name \in \Lambda \setminus \perp \}$$

The additional constraint on the labels of nodes used as parameters (formal or actual) is necessary to allow for a textual representation of these nodes either in the head of the rule or along the invocation arrow. The name of a rule must not be empty.

Note that this definition does not yet support UQS or premise rules as these are captured in the notion of rule schema only:

Definition 20 (DMM Rule Schema with Premises and UQS)

A DMM rule schema consists of a core DMM rule $core \in r_{DMMInv}$, an invocation relation pre , and a set of universally quantified structures $UQS \subseteq r_{DMMInv}$:

$$RS = \langle core, pre, UQS \rangle \text{ with} \\
core \in r_{DMMInv}, UQS \subseteq r_{DMMInv}, pre \text{ an invocation relation} \\
pre \subseteq \{0\} \times \Lambda_P \times 2^{core.L.N}, \\
\forall u \in UQS: core.L \subseteq_G u.L, core.R \subseteq_G u.R,^{10} \\
u.name = \perp, u.Params = u.NAC = \{ \}$$

Premise rule invocations must have special names (those starting with “P-“) and must not carry a sequence number. A UQS is a complete rule which always

⁹For the following definitions we use the symbol “.” to navigate through hierarchical structures, i.e., $a.b.c$ is the expression for a c which is part of b , which in turn is part of a .

¹⁰ \subseteq_G denoting the subgraph relation

contains the core rule. The "truly" universally quantified structures (UQS_t), i.e., the parts can thus be obtained by $\bigcup_{u \in UQS} u \setminus core$. Note that the elements of UQS_t are not graphs anymore as edges may be left dangling. UQS are only extensions of a core rule, thus their signatures are empty. A UQS rule must not contain NACs, but it may contain invocations (cf. Fig. IV.15 in which the premise rule $P_hasC()$ is part of the UQS rule for node B).

DMM Rule schema are unfolded to final DMM Rules in two steps: The first unfolding resolves the universally quantified structures (UQS) to simple nodes, edges and NACs. The second unfolding glues the premise rules with the core rules.

Applying both of these unfoldings yields final DMM rules (including only NACS and invocations). Fig. IV.16 illustrates this process: The rule schema on top of the figure contains universally quantified elements, an invocation of a premise rule, and an invocation of a small-step rule. The first unfolding yields a number of rules without universally quantified elements, i.e., UQS B is unfolded into two distinct elements here. The second unfolding glues the premise rule, yielding a final DMM rule. If multiple premise rules meet the requirements of the invocations, multiple rules may emerge from this unfolding. Invocations of small-step rules are not affected by the unfoldings.

An auxiliary definition is that of a rule embedding. A rule embedding basically means that all structures of one rule (*sub*) (including NACS and invocations) can be found in another rule (*super*) under consistent renaming of nodes.

Definition 21 (Rule Embedding)

An embedding of rule *sub* in rule *super* is an elp-morphism emb
 $emb : (sub.L \cup sub.R) \rightarrow (super.L \cup super.R)$ with

$$\begin{aligned} sub.L|_{emb} &\subseteq_G super.L, \text{ }^{11} \\ sub.R|_{emb} &\subseteq_G super.R \\ sub.NAC|_{emb} &\subseteq super.NAC \\ sub.inv|_{emb} &\subseteq super.inv \end{aligned}$$

A rule schema may contain multiple universally quantified structures and each of these can be expanded to a number of rules which each contain a different number of copies of the UQS. The concept of "contain a copy of" can now be captured using the notion of rule embedding. In an algebraic definition of Graph Transformations, this construction would amount to a co-limes.

Additionally to the positive (i.e., existential) copies of the UQS, a limiting NAC must be created in the unfolding to prevent partial matches. These two concepts are captured in the notion of positive and negative expansion respectively. Note that the definition requires a non-empty set of positive expansions, i.e., each universally quantified structure implies existential quantification.

Definition 22 (UQS expansion)

The positive expansion of a (universally quantified) rule u n times in the context of a rule r yields a rule $exp_p(u, r, n)$ with

¹¹with $|_{emb}$ renaming all nodes according to emb

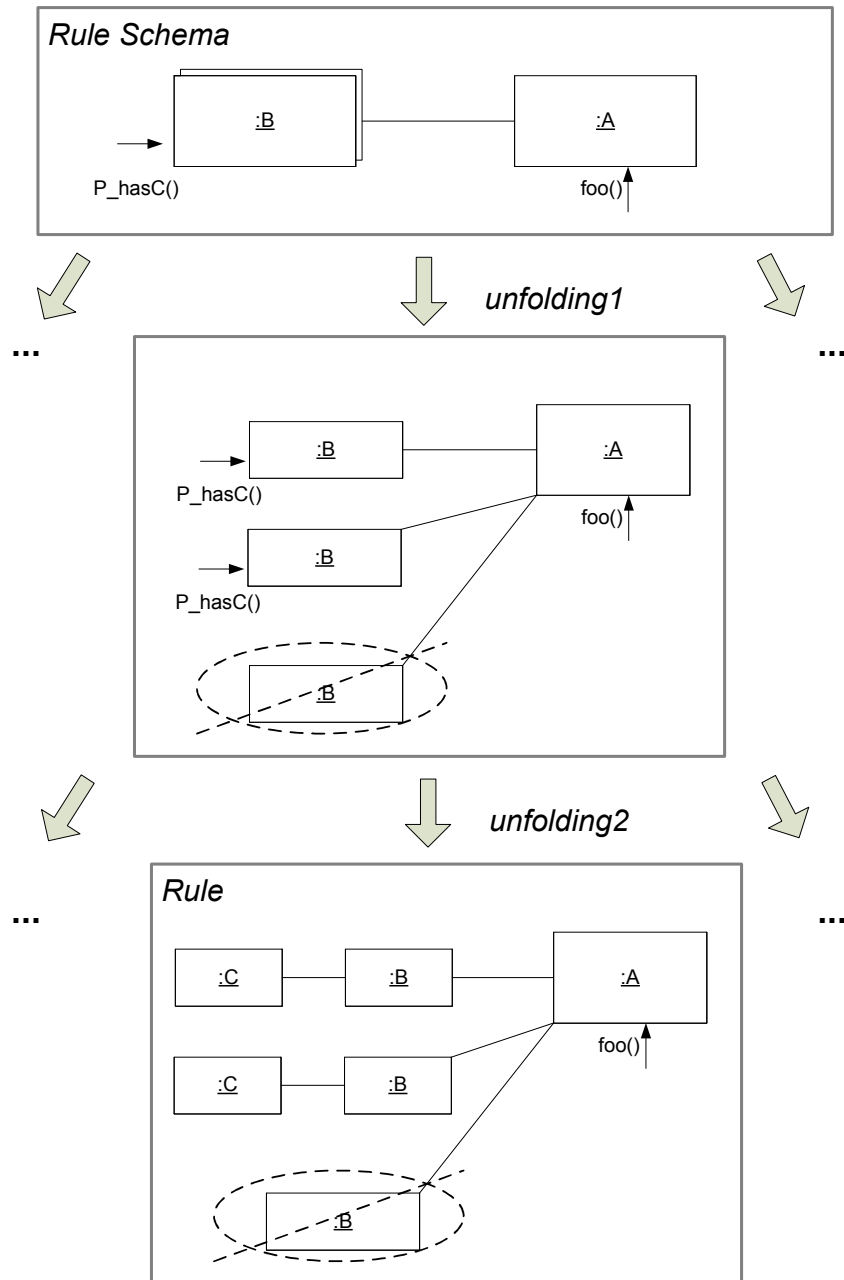


Figure IV.16: Example for the unfolding of a DMM rule schema

\exists a finite set of embeddings *copy* with
 $\forall e \in \text{copy} :$
 e is the identity function for all elements of $r.L \cup r.R$ and
 $\exists f \in \text{copy}$ with $f \neq e$ and
 $\exists n \in \text{UQS}_t.N : e(n) = f(n)$
 $|\text{copy}| = n$, there are exactly n copies of the UQS and
 $\text{exp}_p(u, r, n) = \bigcup_{e \in \text{copy}} e(u)$ no other elements are in exp_p

Note that the different embeddings of the set *copy* form disjoint copies in the expanded rule.

The negative expansion of a (universally quantified) rule u in the context of a rule r is a rule

$\text{exp}_n(u, r)$ with
 $\exists \text{elp-morphism } \text{ncopy} : (u.L \cup u.R) \rightarrow (\text{exp}_n.NAC.L \cup \text{exp}_n.NAC.R)$ with
 $(u, \text{exp}_n.NAC, \text{ncopy})$ is an embedding,
 ncopy is the identity function for all graph elements of $r.L \cup r.R$ and
 $\text{exp}_n(u, r) = r \cup \text{ncopy}(u)$

Both types of extensions define a rule which contains the core rule part of the UQS (identity function) and a number of disjoint copies of the true UQS structures as either parts of the rule itself (positive expansion) or its NACs (negative expansion). Due to the identical core part, both extensions can be combined by a simple union.

Combining a positive and negative expansion yields a single rule with n copies of the UQS and an additional copy of the UQS as a NAC. Such a rule matches if the universally quantified structure appears exactly n times in the host graph.

Definition 23 (Unfolding 1)

An unfolding¹ $uf1 \subseteq S_r \times S_r \times \mathbb{N}^+$ is a relation between rule schemas and for $\langle S, S', i \rangle \in uf1$ the following conditions hold:

$$\begin{aligned} S'.r &= \bigcup_{x \in S.UQS} (\text{exp}_p(x, S.r, i) \cup \text{exp}_n(x, S.r)) \\ S'.UQS &= \{ \} \end{aligned}$$

The unfolding¹ expands all universally quantified structures. The result is a rule schema in which all universally quantified structures have been resolved to existentially quantified structures and NACs (cf. the example in Fig. IV.5). As a rule schema may specify multiple UQS for a rule, the outer union combines all of these separate expansions. Using the parameter i , an unlimited number of unfoldings can be constructed from a single rule schema with UQS.

The unfolding of premise rules works analogously to the unfolding of UQS: We first define the expansion of a *single premise rule* and use this definition to form the unfolding of *all* premise rules of a schema.

Definition 24 (Premise expansion)

The expansion of a rule schema S by a premise rule r_{pre} , invoked by i_{pre} is the rule

$$\begin{aligned}
exp_{pre}(S, i_{pre}, r_{pre}) &= (L', R', NAC', inv', Params', name') \text{ with} \\
&\text{matches}(i_{pre}, r_{pre}) \text{ an invocation matching as defined below} \\
L' &= S.L \cup r_{pre}.L|_{\text{matches}} \\
R' &= S.R \cup r_{pre}.R|_{\text{matches}} \\
NAC' &= S.NAC \cup r_{pre}.NAC|_{\text{matches}} \\
inv' &= (S.inv \setminus i_{pre}) \cup r_{pre}.inv|_{\text{matches}} \\
Params' &= S.params, name' = S.name
\end{aligned}$$

A premise expansion glues the invoked premise rule r_{pre} with the invoking rule r . All structures of the premise rule are merged with the structures of the invoking rule with the passed parameters forming the interface. The union is thus performed under the parameter matching found in $matches$. Since a premise rule may contain invocations of premise rules itself, these will be transferred to the expanded rule. We use exp_{pre}^* to denote the transitive application of exp_{pre} to the point of premise-freeness.

We need to define the notion of matching an invocation to a rule:

Definition 25 (Invocation-Rule matching)

An invocation $invoc$ and a rule r are said to match under the following conditions

$\exists matches : invoc.args \rightarrow rule.params$ a bijective function

$\forall \langle a, p \rangle \in matches :$

$$a.name = p.name$$

$$a.type \in \text{clan}_I(p.type)$$

and $invoc.name = rule.name$

Invocations can thus match to rules which work on subtypes of their defined parameters.

Unlike the unfolding of UQS which are local to the rule schema, an unfolding of premise rules has to happen in relation to a given rule set. This rule set may yield one or more possible rules matching the premise invocation to unfold.

Definition 26 (Unfolding 2)

The unfolding $uf2 \subseteq S_r \times r_{DMMFinal} \times 2^{r_{DMMFinal}}$ is a relation and for $\langle S, fr, Rules \rangle \in uf2$ the following conditions hold:

$$fr = \bigcup_{x \in S.pre} \left(\bigcup_{y \in Rules} (exp_{pre}^*(S, x, y)) \right)$$

The second unfolding produces final DMM rules, i.e., rules which no longer contain any premise invocations. Again, a union over all premise expansions is possible due to the preserving of the invoking rule in each expansion.

Applying `unfolding1` and `unfolding2` to a rule schema thus yields an (possibly unlimited) set of final DMM rules.

While premise rule invocations and UQS were dealt with in this way, the invocation of small-step rules influences the way a rule is applied:

Definition 27 (Rule Application under Invocation)

The notion of rule application has to be extended to handle the invocation mechanism. A rule application is now a quadruple $\langle r, m, caller, level \rangle$ of a final

rule r , a matching m , and two natural numbers $caller$ and $level$. The former indicates the number of the rule application which triggered the invocation fulfilled by this rule application (0 for big-step rules). The latter is used to distinguish between the levels of nested invocations.

Definition 28 (Concrete Invocation)

A concrete invocation $ci = \langle inv, d \rangle$ is a tuple of an invocation and a derivation d with $inv \in d.ra.rule.inv$.

A concrete invocation can be considered as the instance of the invocation specification in a rule. Once a rule is applied, its invocations are concrete and need to be fulfilled by other rule applications. For the interval between the creation of a concrete invocation and its fulfillment we call a concrete invocation an *open invocation*. An invocation is said to be fulfilled if a later rule application in the derivation sequence applies a small-step rule which conforms to the invocations, i.e., which has the correct name and utilizes the elements passed as parameters at invocation time.

Definition 29 (Fulfillment of Invocations)

Fulfillment is a relation, $fulfills : \langle ci, d \rangle$ between a concrete invocation ci and a derivation d with the following restrictions:

$d.ra.caller = ci.ra$ the caller of the rule application is the issuer of the concrete invocation

$matches(ci.inv, d.ra.r)$ the invocation and the rule do match

$ci.ra.level = d.ra.level - 1$ the fulfillment is always a nesting level below the invocation.

Definition 30 (Open invocations)

For any particular derivation state G_i in a derivation sequence there is a set of open invocations OI defined as $OI(G_i) = \{\text{concrete invocation } x \mid x.ra = ra_j, 1 \leq j < i : \nexists ra_k, j < k < i : fulfills(ra_k, x)\}$

Moving from a single rule application to a complete systems, we introduce the concept of big-step rules (*BRules*). This subset of the rules has to obey special constraints in that it can only be applied if no invocations are currently open:

Definition 31 (DMM System)

A DMM System is defined by two sets of final rules *BRules* and *SRules* and a start graph G_0 :

$S_{DMM} = \langle BRules, SRules, G_0 \rangle$

It describes derivation sequences $\{G_0 \xrightarrow{ra_1} G_1 \xrightarrow{ra_2} G_2, \dots, G_{n-1} \xrightarrow{ra_n} G_n\}$ with

- ◆ $\forall 1 \leq i \leq n : ra_i.rule \in BRules \cup SRules$
Big-step and small-step rules may be applied in the system.
- ◆ $\forall G_i \xrightarrow{x} G_{i+1}$ with $x.rule \in BRules : OI(G_i) = \{\}$
Big-step rules may only be applied if there are no open invocations.
- ◆ $\forall G_i \xrightarrow{y} G_{i+1}$ with $y.rule \in SRules :$
 $\exists z \in OI(G_i) : fulfills(z, y)$
and $\forall u$ with $G_j \xrightarrow{z} G_k \xrightarrow{*} G_l \xrightarrow{u} G_m \xrightarrow{*} G_i :$

$$(u.ra.level < z.ra.level) \text{ or} \\ (u.ra.level = z.ra.level \text{ and } u.ra.caller = z.ra.caller)$$

The application of small-step rules is only possible if they have been previously invoked. All derivations between an invocation and its fulfillment must either process other rules invoked by the same derivation or lower level rules. This constraint ensures the stack-like behavior of the open invocations.

- ◆ $\forall G_i \xRightarrow{v} G_i + 1$ with $fulfills(ci, v)$ and $ci.inv.seq > 0 : \nexists w \in OI(G_i) : (w.ra = ci.ra) \text{ and } (0 < w.inv.seq < ci.inv.seq)$
Rules may not be applied unless invocations stemming from the same rule application with a lower (positive) sequence number have been processed first.
- ◆ $OI(G_n) = \{\}$
The last state in a derivation must be free of open invocations.

Note that the last constraint forbids the existence of derivation sequences which are stuck due to an unresolved invocation. A DMM system only contains valid sequences in which no invocations are open anymore. This does not imply termination as the application of big-step rules may still be possible in G_n .

IV.7 Discussion

This chapter provides an overview of existing graph and graph transformation notions. Our usage of these notions for DMM was guided by criteria derived from the general requirements of the DMM approach. We obtained a formalism which provides convenient means to manipulate graphs representing object structures. The rules of this approach can be represented using symbols from UML Communication Diagrams. The main control concept of DMM rules is the innovative mechanism of rule invocations. It combines in a unique way the loose control of rules (ideal for expressing non-determinism and for easy extensibility) and the tight control of invocations (ideal for distributing complex manipulations over different rules and for reuse of existing specifications). The extensive application example in Chapter VI proves that these features work in the way we intend them to and that it is possible to formulate concise rules and rule sets with DMM rules.

A remaining open point is the analyzability of the approach as one motivation to base the operational part of DMM on an established formalism was the re-use of existing knowledge. There are a number of analysis techniques to determine properties of a set of Graph Transformations. Direct application of these techniques is difficult, though, as the invocation mechanisms are not taken into account by these techniques. Termination analysis [Plu98, HKT02, EEdL⁺05] is one such technique which can be applied to prove that a given GT system

yields only finite derivation sequences. Termination is not a generally required property for DMM specifications, though. As models may legally express non-terminating behavior, the semantics must be able to reflect this. One can, however, regard each big-step rule and its transitively invoked rules as a special kind of forward closure (see [Mül96]) and look for monotonicity criteria to argue for the termination of each big-step. Analysis of *conflicting pairs* as carried out by the AGG tool set [TB94, LB93] is also rather meaningless for DMM rules as invocations impose dependencies between the rules which are not taken into account by AGG. Confluence [HKT02] is not an issue for semantic specifications as non-determinism is a feature here and not a bug.

It turns out that in fact all usually gained analysis results for graph transformation specification have little or no relevance in the DMM context. We do believe, however, that practice will show which properties of a DMM specification are desirable. Analysis methodologies can then be devised to answer such needs. This is a topic of future work along the general technique of DMM.

Chapter V

The Architecture of Dynamic Meta Modeling

The outline of Dynamic Meta Modeling as introduced in Sect. II.4 is displayed in Fig. V.1. Having introduced the technical means to express the semantic mapping (Meta Relations, Chapter III) and operational rules (DMM rules, Chapter IV) we focus on the architecture of the DMM approach and its detailed construction in this chapter.

Section V.1 targets the static semantics part of DMM and explains the motivations for and benefits of constructing the semantic domain meta model and its accompanying Meta Relations. Section V.2 turns to the dynamic component of the semantics and details its connections to the static semantics (i.e., the small and as of yet unlabeled connection between the static and dynamic semantics package in Fig. V.1). The effect of combining these techniques is made visible in Sect. V.3 which demonstrates how DMM specifications serve to determine a concrete model's meaning in terms of a Labeled Transition System.

Additionally, a *modularity* concept for DMM specifications is discussed and introduced in Sect. V.4. This modularity concept allows for maintenance and extension of existing specifications.

The concluding Section V.5 reviews the concepts of DMM and evaluates how DMM fulfills the requirements posed in Subsect. II.2.4.

Throughout this chapter we use a small excerpt of the UML Activity Diagrams case study presented in Chapter VI. The example illustrates the way the different sections in this chapter fill the abstract parts in Fig. V.1 with the concrete concepts of DMM. We already know the UML's syntax structure thus Fig. V.2 illustrates the starting point for the semantics definition. We can see that the model (lower left corner) under consideration consists (partially) of an `ActionNode` called 'do something' and an outgoing `ControlFlow`. The fragment of the UML meta model defining these elements is shown in the top left corner. Here, we can see the concepts of `Node`, `ActionNode`, and `Controlflow` being defined together with their connections and a restricting multiplicity.

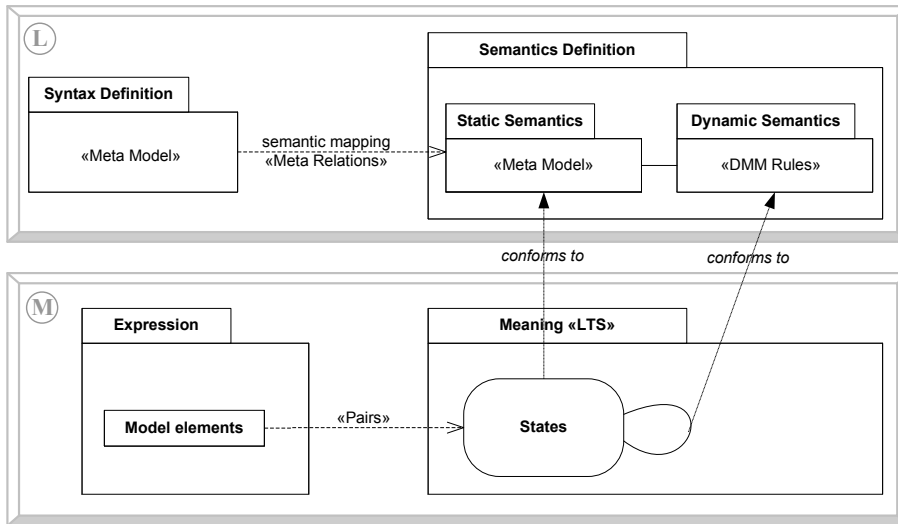


Figure V.1: Overview of the DMM architecture

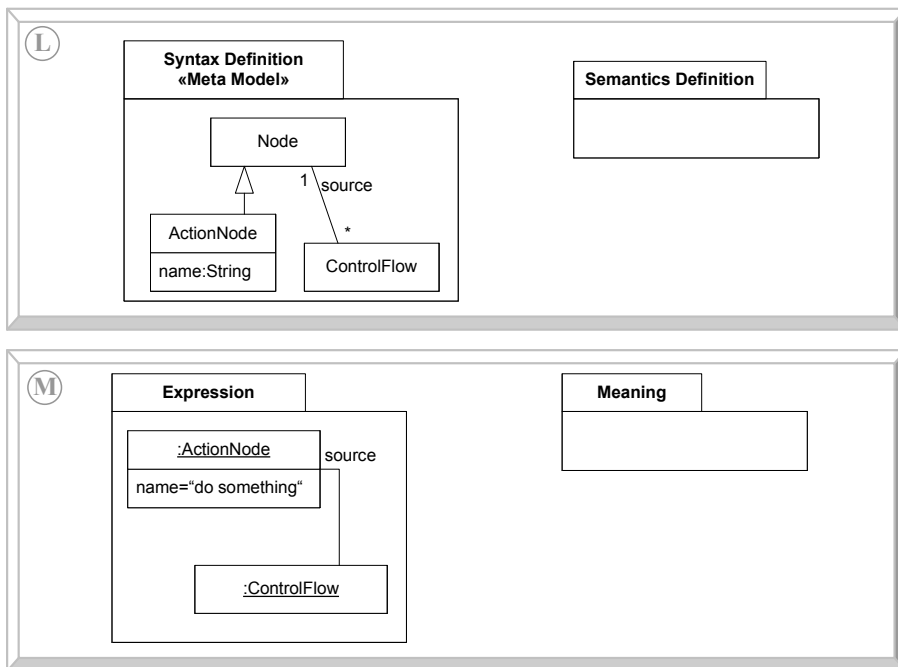


Figure V.2: Example expression in UML

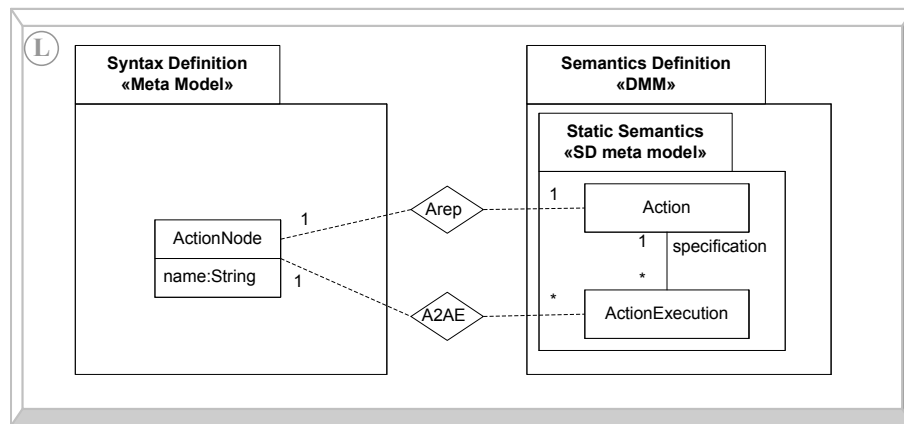


Figure V.3: Example for the expression of static semantics in DMM

V.1 Expressing Static Semantics in DMM

DMM makes use of *denotational meta modeling* (cf. Subsect. II.3.2). For the definition of a Visual Modeling Language's static semantics we thus formulate a *Semantic Domain Meta Model* (SD meta model) and a precise semantic relation between the syntactic and the semantic elements. Geisler, Klar, and Ponds use the terminology of *intensional elements* for elements of the syntax and *extensional elements* for the semantic concepts [GKP98]. We employ this terminology in the following discussions. In the DMM approach we use Meta Relations as introduced in Chapter III to express the mapping between intensional and extensional elements. Employing the denotational meta modeling approach serves several purposes:

Definition of Semantic Concepts Expressing the semantics of a language usually requires different concepts than the ones used to construct expressions in the language. For instance, to understand Class Diagrams one refers to the notion of objects, for Petri Nets and Activity Diagrams the notion of token and offer are central to their interpretation, and Statecharts are usually explained based on the notions of active state and firing. All of these concepts are crucially important, in fact it is not possible to provide a semantics in syntactic terms only. To actually form a reliable basis for a model's interpretation, such concepts and their details need to be formulated precisely. Meta models (i.e., Class Diagrams) provide the means to achieve such a detailed formulation of semantic concepts.

In our running example (illustrated in Fig. V.3) the semantic concept of `ActionExecution` is defined (by being a class in the semantic domain meta model). This class denotes a currently running execution of an `Action`. Additionally, structures (i.e., associations), constraints (e.g., the multiplicities at the association to the specifying `Action`), and generalizations (`Action` is a subclass of the general concept `BehaviorExecution`, cf. Fig. B.63) can be used to specify details of the semantic domain's structure. The explicit formulation of the semantic

domain in this way enables a Language Engineer to clarify and detail his ideas about the semantic concepts underlying the language.

Explicit mapping Not only the extensional elements are clearly expressible, but—using the techniques of Meta Relations—their connection to the intensional elements can also be precisely defined and laid out visually. Regarding Fig. V.3 (and disregarding Relation *Arep* for the moment) we can see that the semantics of an *ActionNode* is given by a set of *ActionExecutions*. This ability of relating intensional and extensional elements is called “a remarkable feature of meta modeling” in [GKP98].

For elements without dynamic semantics this denotational meta modeling semantics is their whole semantics. The (syntactic) concept of a *Class*, e.g., can be fully expressed by relating it to the (semantic) concept of an *Object*. All specifications of the class (attributes, associations, constraints) must be respected by the objects defined by this class. For elements with dynamic semantics, however, additional effort is necessary.

Basis for dynamic semantics Beyond establishing a structure of extensional elements and their interrelations, the semantic domain meta model also forms the basis for the dynamic semantics. The mechanism employed here is again aligned to the object-oriented concepts of the UML. Each semantic concept which expresses dynamic semantics (i.e., it has behavior) captures this behavior in a set of operations. In the example in Fig. V.5 the class *Action* in the SD meta model defines the operation *action.start*()*, which encapsulates the notion of starting an execution of a certain action.

Preparation for operational rules In an ideal world, the denotational part of the semantics would be completely independent from the dynamic part. While DMM allows for such complete independence, it is not easy to achieve in practice. In fact, the ease of formulating operational rules to express the dynamic semantics greatly depends on the way the static semantics are formulated. It is thus advisable to allow for certain modifications to the semantic domain meta model which aid the subsequent formulation of DMM rules. An example for a simple modification is the introduction of an ordering pattern to iterate over sets (cf. Subsection VII.2.3).

A rather far reaching manipulation is the replication of intensional elements in the semantic domain. From a conceptual viewpoint this is inadvisable for reasons of conceptual cleanness and avoidance of redundancy. Intensional (i.e., syntax) elements should be kept in the syntactic domain and related to the semantic domain by means of Relations only. An operational interpretation of the system state will, however, heavily depend on the intensional elements to compute the next system state. This computation would have to take Pairs into account to actually access these intensional elements. Pairs are, however, not covered in the notion of DMM rules as defined in Sect. IV.4 nor do they find support in Graph Transformation tools. With regard to meta operations forming the base for operational behavior it is also often the case that operations are quite naturally located in meta classes representing intensional rather than

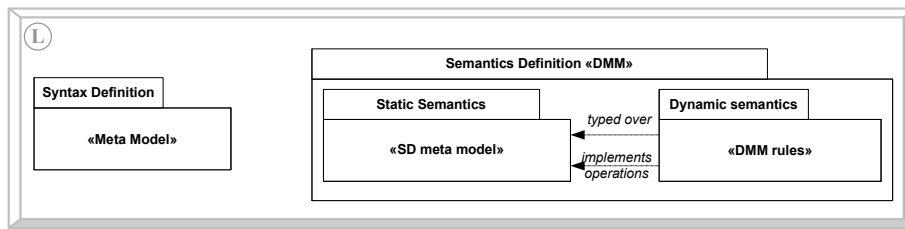


Figure V.4: Detailed view on the structure of semantics definition in DMM

extensional concepts. Regard, e.g., the UML semantics description which uses the terms 'an action executes' or 'a transition fires', both of which refer to syntactic elements.

From a practical point of view it is thus advisable to replicate parts of the specification into the semantic domain meta model. If you regard the example in Fig. V.5 you will notice that the semantic domain fragment provided there contains a replication of the Action class as well as its extensional counterpart, the ActionExecution class.

V.1.1 Summary

Denotational meta modeling semantics form the first part of the DMM technique. They allow for the relation of syntactic and semantic concepts of a language. If we think of the semantics as being an interpreter semantics (cf. Subsect. II.3.3) we can also view the semantic domain meta model as the definition of the runtime states of the interpreter. In these runtime states the specification information (replicated intensional elements) forms the "program" while extensional and auxiliary elements provide the control and data state of the interpretation.

V.2 Expressing Dynamic Semantics in DMM

While denotational semantics descriptions have their strengths in expressing static semantics, elements with behavior semantics are better served by being expressed in an operational style (see the discussion in Sect. II.4). DMM thus adds an operational component to the denotational part. This operational component is expressed by DMM rules as defined in Sect. IV.4. Figure V.4 illustrates the two parts of a semantics specification in the DMM approach.

The semantic domain meta model forms the base for the operational semantics component of DMM in two ways: on the one hand the meta models provides the type graph over which DMM rules are to be formulated. On the other hand the DMM rules 'implement' the operations defined in the classes of the semantic domain.

The DMM rules allow for the formulation of dynamic semantics. For instance, while the semantic domain meta model defines the notion of an Offer for an

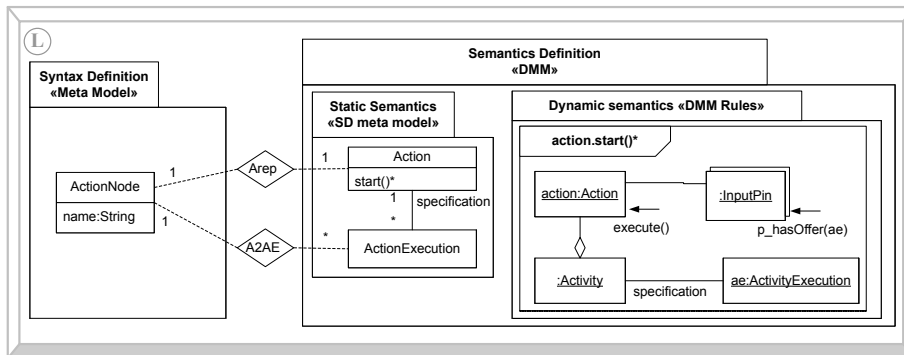


Figure V.5: Example DMM specification including a SD meta model and a DMM rule

Activity Diagram, a rule can express the concept of 'accepting an offer' by expressing the changes to the system state. Using the invocation mechanism introduced in Subsection IV.6.1, complex behavior can be split up over a number of interconnected small-step and premise rules.

The connection between operations and rules (cf. Fig. V.4) is based upon the signatures of the rules which must correspond to that of the (meta) operations. There may be multiple rules per operation. Multiple rules per operation usually express different situations (cases) in which the behavior may execute. Each rule in the DMM rule set *must* have a corresponding operation in the semantic domain meta model.

Adding dynamic semantics to our running example results in the situation in Fig. V.5. The class `Action` has been extended by an operation `action.start()*`¹ describing the start of an action's execution. The rule implementing this operation is provided in the right hand part of the semantics definition. This rule expresses that an action may only start if it carries offers on all of its inputs.

The correspondence of rules and operations takes subtyping into account. If a class in the semantic domain meta model is a subtype of another class, it inherits these class' attributes and operations. It is thus possible to have the (abstract) class `Action` which defines the operation `execute(context:ActivityExecution)` (cf. Fig. V.6), yet the rules for this operation are formulated for concrete subclasses of `Action`, namely `CallBehaviorAction` and `DummyAction`. This mechanism enables the formulation of interface-like abstract classes which leave the implementation of defined operations to the specialized subclasses. It is also possible to extend the behavior of super classes by adding new rules to an existing (and already implemented) operation. Suppressing or overwriting existing rules is not possible.

¹The "*" indicates big-step rules, see Section IV.6 for details.

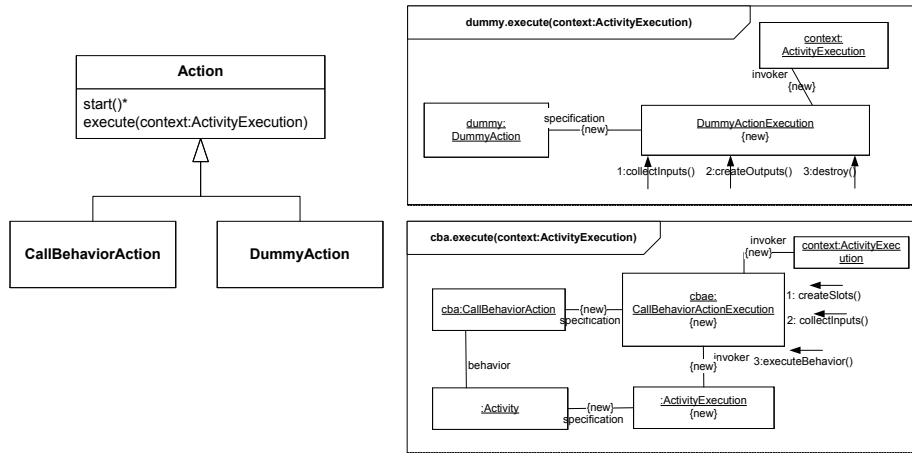


Figure V.6: Example for DMM rules with subtyping of the context node

V.3 Model Semantics in DMM

The semantics of a (concrete) model is a set of Labeled Transition Systems (LTSs).

A single LTS is constructed by states and transitions between these states. The states a model represents are the set of all instances of the SD meta model which can be mapped by Pairs (conforming to the semantic Relations) to the model at hand. In Fig. V.7 two such states are illustrated: One (S2) having a single active execution of the action `do something`, the other (S1) executing this action twice concurrently. In the figure the compliance to the semantic relations is illustrated only for S1 by showing the pairs connecting it to the model.

We call these states *interpretation states* or *system states*. Internally, these states are represented by state graphs, i.e., they form instance graphs to the type graph provided by the SD meta model (cf. Section IV.4). This corresponds to the notion of Extended Transition Systems as defined in [GR01, WC90] or Labeled Graph Transition System as defined in [CHM00]. Usually, we expect the set of all such legal interpretation states for a single model to be unlimited.

The transitions between interpretation states have to conform to the DMM rules defined for the language. Each transition is labeled with the name of the applied rule and details on its application. In Fig. V.7 we can thus observe that the application of `actionExecution.terminate` on the state S1 yields the state S2 (details are omitted in the figure).

The set of all possible states will thus form one or more (in most cases even unlimitedly many) connected Labeled Transition Systems. Each of these systems is characterized by the start state it originates in. Each system must furthermore form a DMM system as defined in Sect. IV.6, i.e., invocations must correctly be fulfilled and may not remain open in final states. Constructively, the LTSs of a model may thus be generated by supplying a set of legal system states of the model and deriving the transition systems from them by repeated

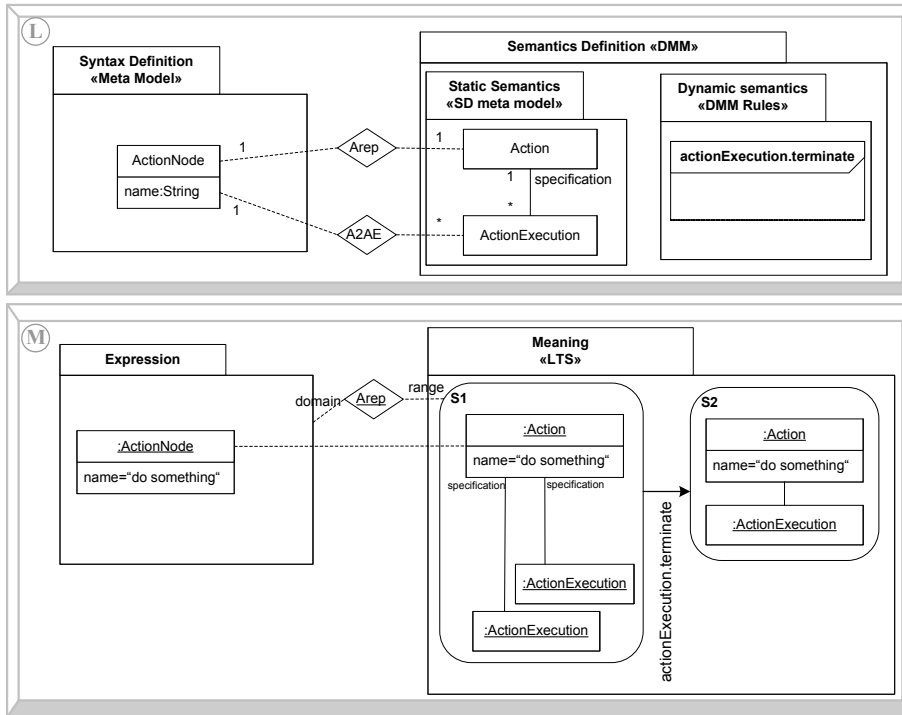


Figure V.7: Example LTS illustrating a model's semantics under DMM

application of the DMM rules. Note that in addition to the number of such DMM systems being unlimited, each of them may also be of indefinite length if the model describes a non-terminating behavior.

V.4 Modularity and Extensibility

Modularity has not been an explicit requirement of a semantics description technique. In Subsect. V.4.1 we elaborate how modularity serves to achieve the desired qualities of adequacy, understandability and universality (the latter by providing support for UML's extension mechanism). Modularity for (semantics) specifications has been regarded before, thus we review relevant literature in Subsect. V.4.2. We then proceed to detail how packages can be used to structure a DMM specification into separate modules and how restrictions placed upon these packages serve to guarantee conservative behavioral extensions (Subsection V.4.3).

V.4.1 Motivation for Modularization

A semantics description for Visual Modeling Languages (especially UML) needs to provide modularity concepts for three reasons:

Understandability While DMM aims to achieve precise and concise semantics specifications, there can be no doubt that a complete formalization of a language like UML will result in a very large specification. Understanding such a specification can only succeed if it is broken up into thematically grouped modules.

Adequacy Up to now we regarded adequacy as a static concept, capturing the ease of use of specifying certain concepts formally. It has, however, also a dynamic aspect in that a semantic specification is as prone to change as its underlying language. Thus maintenance of such a specification needs to be considered, too. Again, clearly structured modules help to localize errors and to determine the impact of changes.

Universality The UML provides explicit means to adapt the language to special application areas. On the syntactic side, the profile mechanism captures user-defined syntax elements. An equivalent construction is necessary on the semantics side to fill these new constructs with meaning. Using DMM such additions should integrate seamlessly with the existing specification. Semantic modules thus help to support this important aspect of the UML.

A first observation is that the concepts of modularity and extensibility can be regarded uniformly. In both cases we require a mechanism to combine different parts of a specification. Whether these parts have been designed for a standard (i.e., are part of the core language) or for specific domains/projects needs to make no difference. It is in fact beneficial for understandability if only a single mechanism is employed for both purposes. We thus provide a single extension mechanism which can facilitate both the combination of pre-defined (i.e., UML standard) modules and user-defined modules.

As DMM is intended for the expression of semantics, it is a prime requirement that extensions must be conservative. The notion of *conservative extensions* is important as it guarantees that a language extension does not change the way in which pre-extension models are to be interpreted. Without this guarantee all investments in the general language (i.e., knowledge, analysis results, tools) have to be re-evaluated upon the introduction of a language extension. An example for the introduction of a language extension in the form of a UML profile and the subsequent conservative extension of the DMM semantics can be found in [HHS01].

V.4.2 Related Work

Modularity for semantics specification techniques is studied by Mosses in [Mos00]. He compares the impact of extending a language specification by data references, exceptions, and concurrency for different operational, denotational, and hybrid approaches. One finding of the investigation is that the modularization approach of Action Semantics allows for a stepwise extension without changes to the existing specification. Applicability of these results to DMM is very limited as data references and concurrency are naturally supported by DMM and exceptions are not usually a first-class language element in modeling languages. We do, however, agree with the author that modularity must be

measured by concrete examples and do thus evaluate our case study accordingly (see Sect. VI.3).

The introduction of a modularization concept for Graph Transformations is being discussed by Ehrig and Engels in [EE93]. Three different types of modularity are being distinguished: *graph partitioning* allowing for local transformations which can be re-synchronized via interfaces to a global state, *system inheritance* which allows a graph transformation system to extend existing specifications, and the *import-export interface* concept which allows reuse of specifications at runtime. The first of these modularity concepts is rather aimed at parallel execution of graph transformations and is not investigated any further in this thesis. The second modularity concept has since been extended to the notion of typed graph transformations with inheritance [BE_dL⁺03] which DMM supports. The third modularity concept finally calls for the (abstract) definition of graph transformation rules which rely on imported rules to execute. Such a notion (albeit without modules yet) is present in DMM by the invocation concept.

Different practical GT approaches have implemented concepts outlined in [EE93]. In DIEGO [TS95], modules export partial views on Graph Transformations which can be imported by other modules. By supplying identically named rules, an importing package can add its own specification to imported rules. The mechanism of rule amalgamation is used to combine the different fragments of a rule specification. Compared to the invocation mechanism of DMM rules, the DIEGO module concept does not allow for the combination of imported rules but only for their extension.

Realization of import/export interfaces in PROGRES is discussed by Winter and Schürr in [WS97]. A key issue for them is to retain the visual appeal of graph transformation rules even if they only form wrappers for imported functionality. Thus, they argue for an interface definition in terms of graph transformation rules which provide a visual representation of the pre- and the post state. The use of visual operation specifications leads to a dilemma: As the rules comprise data types to express their behavior, information about the (usually hidden) data structure can be laid open and are thus accessible to rules from other modules (see also the discussion in [BR04]). For DMM, however, we are less concerned with the graphical representation of interfaces as the invocation mechanism embeds the invocation into a rule and pure textual specifications are avoided.

V.4.3 Modularity Concepts of DMM

Modularity in DMM is based on the concept of classes and packages. Classes are used as explained above to form the semantic domain meta model and to anchor the DMM rules by their operations. Rules are called via the invocation mechanism based on the rule's signature. It is thus possible to modify or interchange rules with identical signatures without causing adaptations in other rules. The rules themselves, however, are free to match and change every part of the underlying graph. DMM does thus not natively support the notion of distributed state space as formulated in [EE93]. While we recognize the increased maintainability of clearly separated state spaces, a consequent separation comes

at a heavy price in terms of understandability.

Class Inheritance

The simplest form of extending a concept is to define a new class in the semantic domain meta model and add a generalization to an existing class. Subclasses may add new operations and may also add rules for operations defined by their superclass (cf. Fig. V.6). Eventually existing rules for the operation are unaffected by this addition.

As instances of the subclass are automatically typed over the superclass(es) and no behavior restrictions may occur, all behavioral sequences permissible for the superclass also apply to the subclass. This kind of behavior inheritance is called *protocol inheritance* by van der Aalst [AB99] and *invocation consistency* by Engels et al. [EHK01, EE95]. Note, however, that the term protocol inheritance does not make much sense here as a DMM system is not reactive as the term protocol implies.

Projection inheritance (or observational consistency) on the other hand is not implied by the restrictions of DMM rules. Subtypes are free to add rules to inherited operations and may thus extend the behavior in observable ways.

Packages

On top of the class level, we use packages as means of structuring a DMM specification. The type of package used in DMM supports only a limited form of modularization. Each package contains a type graph of the classes, associations and inheritance relations it defines. It furthermore contains a number of DMM rules (or technically precise: rule schemas). When packages are combined by package imports (see below), their type graphs are merged to a single type graph.

Definition 32 (DMM Packages)

$$P_{DMM} = \{ \langle TG, Rules \rangle \} \text{ with}$$

$$TG \in G_{DMMT} \text{ a type graph}$$

$$Rules \subseteq RS \text{ a set of rule schemas typed over } TG$$

Package Imports

If a class from one package needs to refer to or extend a class from an existing package, an *import* must be performed. Imports may concern single elements from other packages (`ElementImport`) or the package as a whole (`PackageImport`). UML furthermore differentiates between `<<access>>`, `<<import>>`, and `<<merge>>` imports. These relationships differ in the restrictions they impose on the use of imported classes. The exact nature of these restrictions and in fact the detailed semantics of the merge construct are still under intense debate (cf. [FTF], issue 6279 or [Stö05c], p.115).

In DMM we use the «import» and «merge» relationships. The merge relationship as the more powerful relation allows for the redefinition of model elements by the importing class. The only kind of structural redefinition we allow is the addition of associations to a class. Additionally, imported classes may be extended behaviorally with new operations and additional rules for existing operations.

When using the «import» relationship, imported classes may only be used to derive new subclasses.

To capture these properties formally, we define the imported type graph TG^* which is then extended to the package's own type graph TG .

Definition 33 (Package Import)

A Package import $P_I : P_{DMM} \times P_{DMM}$ is a function between a receiving Package R and a providing Package P with

$$R.TG^* = \bigcup \{x.TG \mid \exists P_I(R, x)\}$$

$$R.TG^* \subseteq R.TG$$

$$\nexists e \in R.TG.E : (s_E(e) \in TG^*.N \wedge t_E(e) \notin TG^*.N) \vee (s_E(e) \notin TG^*.N \wedge t_E(e) \in TG^*.N)$$

$$\nexists (i_1, i_2) \in R.TG.I : i_1 \in TG^*.N \wedge i_2 \notin TG^*.N$$

When using the «import» package import, imported elements may only serve as the basis for generalizations. The last two constraints ensure this condition by interdicting edges between nodes of the imported type graph and nodes defined in the package and by interdicting inheritance relationships from imported to newly defined nodes.

The «merge» import is defined similarly, albeit with less restrictions. The type graph of the merged packages is called TG° .

Definition 34 (Package Merge)

$P_M : P_{DMM} \times P_{DMM}$ and for $(R, P) \in P_M$ holds:

$$R.TG^\circ = \bigcup \{x.TG \mid \exists P_M(R, x)\}$$

$$R.TG^\circ \subseteq R.TG$$

In the Class Diagram of the importing package, all imported classes are marked by the label 'from **PackageName**' near their name. The elements in a package's type graph can thus be partitioned in *imported* ($TG^* \cup TG^\circ$) and *original* $TG \setminus (TG^* \cup TG^\circ)$ elements.

Rule Restrictions

The rules in a package underlie several restrictions which strengthen the modularity of the specification. All rules in a package conform to a single type graph which is the unification of the type graph defined by the original classes of the package (and their interconnections) and all imported packages. Technically, a rule may manipulate any element of the underlying graph. Thus, DMM rules are very powerful and a strict separation of responsibilities for data manipulation is not technically enforced by DMM. Rules may only invoke other rules

which are accessible to it, i.e., whose interface is defined as an operation in one of the original or imported classes in the package.

Conservative Extensions

Extending a specification by introducing a new package may introduce new concepts and thus alter the semantics. But if such alterations are allowed in an unrestricted way, important relations to the base formalism can get lost. In DMM we thus allow a behaviorally conservative kind of extensions only.

Formally, a DMM specifications consisting of a type graph TG (assuming a flattening of the specification in packages) and a set of rules $Rules$ is extended conservatively by a package $P' = \langle TG', Rules' \rangle$, iff for all start graphs typed over TG the derivation sequences are identical in $\langle TG, Rules \rangle$ and $\langle TG \cup TG', Rules \cup Rules' \rangle$. Furthermore, the LTSs gained by interpreting start graphs typed over TG' according to $Rules$ (i.e., new diagrams interpreted according to the old semantic rules) must be a subset (or rather subsystem) of the LTSs gained by applying $Rules'$.

The intention of such conservative extensions is that users as well as tools can rely on the fact that a certain kind of knowledge may never be invalidated by later extensions. In fact it is existential knowledge which will prevail through all extensions. If, e.g., a tool can prove that a basic UML Statechart allows for a certain invocation sequence, it can check this property even on Statecharts extended by a UML profile. Universal and negative properties on the other hand (e.g., proving that a certain condition will never be reached) are not guaranteed to be conserved. From a users perspective, an understanding about a certain kind of diagram can never be invalidated, even if the diagram is extended later on.

Syntactically, conservative extensions are enforced by the restriction that *every rule in a package needs to have at least one original element of the package in its left hand side*. This constraint guarantees conservative extensions: If all rules in $Rules'$ require the presence of an element in $TG' \setminus TG$, they are never applicable to models typed over TG only and thus these extensions are trivially conservative. As rules can never be overwritten in extensions, the interpretation under $Rules$ is always preserved.

V.4.4 Discussion of the Modularization Concepts of DMM

Packages are a practically very important feature of DMM. They allow to structure complex specification into manageable parts. The case study in Chapter VI gives evidence to both the need for such a mechanisms and the suitability of DMM packages to fulfill this need.

Technically, however, the DMM package notion is rather weak. It does neither provide separate namespaces (like UML packages do) nor does it restrict the rules in their ability to manipulate the whole underlying graph.

The former feature, however, is most required in situations where very large specifications are composed from different, possibly independent components.

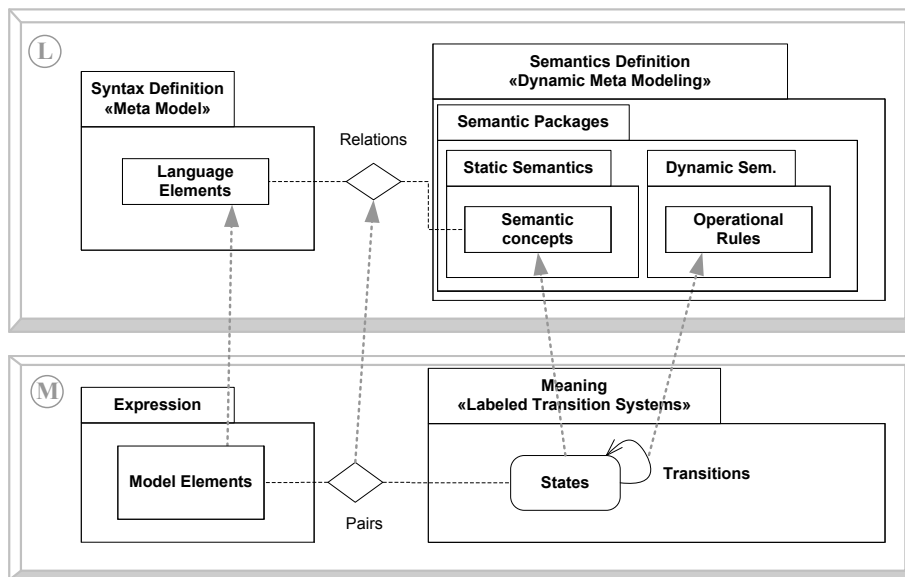


Figure V.8: Architecture of the DMM approach

A rather far reaching trade-off is the abandonment of data encapsulation for packages. The strong visual appeal of Graph Transformations stems from their ability to showcase the manipulation of complex object structures in a intuitively accessible way. Restricting rules to show/manipulate only elements originating in their own package and relying on invocations extensively would have greatly devalued their appeal. We thus accept a decline in maintainability for the sake of higher visual impact of the rules.

V.5 Summary and Discussion

The architecture of DMM can be summarized as follows (cf. Fig. V.8): For a given Visual Modeling Language we provide a semantic domain meta model which contains explicit representations of the semantic concepts. The language's (syntax) elements are related to these semantic concepts by explicit semantic relations. Semantic concepts may contain behavior, which is expressed by one or more DMM rules. These definitions are themselves structured in packages. For a given model (i.e., an expression in the modeling language), a number of states may be constructed (which are valid instances of the semantic domain and have valid instances of the semantic relation to the model). From these start states and the DMM rules, Labeled Transition Systems can be constructed which express the models semantics.

At this pivotal point of the thesis we are at the end of the part which motivates

the problem and introduces our solution concepts. Before we proceed to realizing and applying these concepts, we shall discuss achievements and limitations of the DMM approach. In Chapter I we identified six criteria for an approach to the definition of semantics for Visual Modeling Languages. In the following subsections we discuss DMM under each of these requirements.

V.5.1 Understandability

The prime goal of DMM is to enable the formulation of semantics for Visual Modeling Languages in a way that is easily perceived by our intended target group. All features of the various technique involved have been selected according to this criterion. For the discussion we need to distinguish two scenarios: To understand a language's semantics, a user may either inspect the specification itself, or he may be interested in the concrete semantics of examples, in which case he will inspect the Labeled Transition System(s) for these models. For DMM in general the former case is more relevant. We discuss interpretation of an LTS in Sect. VIII.5.

It also needs to be pointed out that the DMM technique only supplies the provisions to formulate a semantics in a certain way. We will discuss these provisions under the optimistic assumption that they are utilized correctly. The case study in the next chapter reveals the feasibility of this assumption and Chapter VII provides modelers with guidelines for an understandable semantics definition in DMM. Technically, however, it is quite possible to abuse all of DMM's features in a way as to obscure meaning, rather than to reveal it and to confuse the reader. There is thus no guarantee that every concrete DMM specification in fact fulfills all properties discussed here.

Regarding the inspection of rules to understand the constructs of a language we can observe an important property of the DMM approach: The interpretation of DMM rules and their working on the systems state is (almost completely) possible without any knowledge of the underlying formalisms at all. In fact, someone with knowledge in UML only can interpret the communication diagrams in the UML way and reach the same results as someone interpreting the rules on their Graph Transformation background. Notational differences to 'pure' UML communication diagrams are the message notation without an underlying association, the existence of NACs and the constraints {new} and {destroyed}. The latter are well known in the UML community as they were an established part of UML 1.x Collaboration Diagrams (which have been renamed to Communication Diagrams in the move to UML 2). Experienced UML users will thus have no problems in understanding them. Semantic clarifications need to be supplied on the concept of NACs which is not part of UML communication diagrams and the execution order of the different rule types.

Beyond these rather local deviations, the correct interpretation of DMM rules also calls for an understanding of the rule-based character of big-step rules. As the notion of spontaneously occurring (or pro-active) behavior is used in a number of advanced OO concepts (e.g., agent-based systems or user interfaces), understanding about big-step rules should not be problematic. On the other hand the mechanism of big-step rules nicely structures the rule set into different,

independent ‘transactions’ which can be understood separately. A user can thus acquire knowledge about a language a construct at a time. In this way, big-steps add a further structurization means to a semantics definition by DMM.

Expressing semantics of a Visual Modeling Language by our approach really *is* meta modeling as the approach makes use of Class and Communication diagrams only and is able to explain the whole language’s semantics based on this core. Advanced users of UML are thus spared the effort of learning an additional formalism for expressing the language’s semantics.

In designing the techniques for formulating DMM rules and their invocation mechanism we stressed the need for manageable rules and rule sets; the base idea being that a specification should consist of a number of easy to understand basic rules which are combined to form more complex manipulations. The invocation mechanism of DMM rules allows for this kind of structurization. It neither enforces additional textual specifications nor does it rely on the existence of numerous auxiliary elements in the underlying graph. We do thus believe that DMM allows for a clear formulation of both the semantic domain meta model and the operational rules. It should be noted, however, that for easier analysis of the rule sets we have restricted the supported control features (as, e.g., compared to the Fujaba or PROGRES approach). This restriction enforces a certain style of specification which can be regarded as uncomfortable in places.

V.5.2 Precision and Formality

As Dynamic Meta Modeling is (internally) founded upon Graph Transformations, respectively its set-theoretic definition provided in Chapter IV, the notation and application of DMM rules is clear and unambiguous. The definition of UML Relations also harks back to formal logics (in the form of OCL). The precision of the formalism is exemplified by the fact that an automatic interpretation of a DMM specification can occur (as demonstrated in Chapter VIII).

By using Graph Transformations as the technical back-end, we are also free of the meta-circularity problems plaguing Core Semantics or true meta modeling approaches as witnessed by the OMG’s MOF (see the discussion in Subsection II.2.3). Thus, DMM specifications are formal and precise in the semantics mapping, the semantic meta model, and the operational rules.

V.5.3 Analyzability

While DMM is not geared toward specific types of analysis, being a general semantics definition technique, its technical components are based upon established theoretical formalism. Analysis techniques for these formalisms are available, yet it needs to be discussed whether they are profitably employable in the DMM context. Essentially two different parts of DMM can be subject to an analysis: Either the specification itself is analyzed, proving general properties of the language, or a single model’s semantics is analyzed, proving properties of that model.

Labeled Transition Systems are the most common basic format for Model Checking and other analysis techniques. The realization of the case study in GROOVE (cf. Chapter VIII) shows that basic model checking is already possible based on DMM specifications and more sophisticated features are under way as GROOVE is being developed further.

Analysis of the DMM rule set is discussed in Sect. IV.7, albeit with the disappointing result that standard analysis methods are either technically not applicable or that their results are not relevant for DMM. A property of the rule set which would be very interesting is its conformance of a rule set to the type graph and the semantic relations. Both place restriction upon the way a state may be constructed. As the rules modify this state, there need to be mechanisms to ensure that these manipulations produce a legal state. While this problem is hard for Graph Transformations in general (cf. Subsect. IV.3.6), it is even harder for DMM. On the one hand, the constraints are spread over two sets of specifications: one set constraining the state space in general (the semantic domain meta model) and the other one (the semantics relations) imposing restrictions from the concrete model under interpretation. On the other hand, manipulations are usually carried out by a sequence of rules, interconnected by invocations. One might argue that intermediate states in such a distributed state manipulation may temporarily violate the given constraints as long as the resulting stable state is compliant again. A technique which automatically checks a given rule set for guaranteed conformance to the semantic domain meta model and the semantics relations while taking Invocation into account is currently not available.

V.5.4 Adequacy

Adequacy, i.e., the measure of convenience of a semantics definition has been addressed in two ways: First, the flexible construction of the semantic constructs employed to express a language's semantics allows for maximum adequacy. The Language Engineer may choose as many or as few semantic concepts as he deems adequate. All semantic elements can have their own extensional representations or complex mappings can express complex syntactic constructs in terms of few basic semantics entities. DMM provides the means to realize all of these definition styles.

Second, DMM provides modularization concepts which aid maintenance of large specifications. Thus, we hope that even complex languages (the whole UML as the ultimate case) can be expressed in DMM. The missing data encapsulation endangers this kind of maintainability and Language Engineers have to make sure to follow a suitable style of specifications (cf. Chapter VII).

V.5.5 Universality

Whether a specification technique is able to address all of UML's features is captured in the term *universality*. The main characteristics as elicited in Subsection II.2.2 were expression of static as well as behavioral notations (both

supported in DMM), extensibility, and support for underspecification. The latter two terms warrant some additional discussions.

Extensibility Semantics definitions have to be extensible to support modular and/or extensible languages. The extensions mechanism is structurally based on UML packages which is an established mechanism also known from OO programming languages. Being rule-based, the operational part of DMM supports extensions natively. Rules can be added for completely new behavior (i.e., adding new big-step operations) or for extended existing behavior (adding rules for existing operations). The restriction of rules to ensure conservativity of the extensions is reasonable, usually new syntactic elements are used to carry new meanings.

A more severe restriction is the fact that existing behavior can never be suppressed or overridden. A harmful side effect of this restriction is that adding new elements to influence the semantics (e.g., adding a weight attribute to activity edges) can only specify *additional* behavior. It is possible to express that tokens may traverse weighted edges in groups of the specified size. It is not expressible that they may *only* traverse in such groups. In fact the basic behavior of an edge, namely allowing for single tokens to pass can never be suppressed by later extensions. To add edge weights correctly, the core rules of activity edges would have to be re-formulated with a NAC disallowing weight specifications. The core rules would thus have to prepare for changes which are out of their scope (another such anomaly is discussed in Sect. VI.3). Yet, the case study in Chapter VI reveals such cases to be a rare exception.

By providing a clearly defined extension mechanism for DMM we do also provide a clear notion of behavioral extensions for UML model. In the DMM framework, a UML Profile with (syntactic) stereotypes would be complemented by a corresponding package with DMM specifications. The additional semantics such a Profile could express would then be restricted as described above and the extension would be guaranteed to be semantically conservative. This clear notion of extensibility is very beneficial for the UML as especially in the area of Profiles a wide variety of interpretations of the UML standard prevail. DMM provides a clear benchmark to curb such rank growth. If a new semantic concept is expressible within the boundaries of DMM, it is a legal UML profile. If it is not expressible then it does not fall into the category of models describable with the UML. To avoid too strict a cutoff of non-suitable profiles, the UML core semantics specification (in DMM) should prepare for a number of likely extensions by providing their rules with sufficient NACs.

The clear extension mechanisms would, on the other hand, also force the core UML specification to be provided in a way which is purely additive. The benefit for users is that if they understand a meaning of a (basic) construct once, this knowledge will prevail through all further extensions. And while this strict requirement may be uncomfortable in places, it adds necessary rigor to a language specification.

Underspecification Underspecification appears in the UML in two ways: Information can be missing in the language specifications (semantic variation

points) or models can be incomplete in their representation of a system. Both of these situations are addressable in DMM.

Handling semantic variation points (SVP) depends on the type of the variation point. If multiple alternative interpretations are available and it's just a matter of picking the right one for a given domain, then different packages with rules expressing the different workings of the element under consideration can be provided by a Language Engineer. If no semantics are given at all (as, e.g., for the aggregation construct), it is possible to leave the semantics of an element completely open in not providing any rules for its operations. The consequence of this treatment of semantic variation points is that a model's interpretation which encounters such an SVP would stop at that point. Those parts of the model which are independent of the SVP could be interpreted, though. Thus, valuable information about a model can be gained, even though some elements remain unspecified. Special constraints must ensure that the missing semantics have no implicit impacts on the remainder of the model. We expect, however, that such completely unspecified elements are resolved by at least providing a replaceable default meaning in the formalization of a Visual Modeling Language.

For the support of incomplete models we first have to distinguish between *scribbles* and *models*. A scribble is some drawing which uses (amongst others) UML shapes to express certain concepts. Scribbles are not legal UML models and can thus not be the target of a formal semantics definition. An incomplete model does thus at least conform to the UML meta model. This rather limits the amount of incompleteness a model may contain since the meta model enforces a lot of syntactical constraints (it is, e.g., not possible to specify a `CallBehaviorAction` without indicating which behavior is meant to be called by it). There are, however, also a number of elements which explicitly express the fact that detailed information is missing here. For Actions, e.g., the `OpaqueAction` is an action which performs some behavior not specified in UML (if at all). As these elements are well integrated in their surrounding model (e.g., an `OpaqueAction` is an `Action`, can thus have `Pins` and so on), we can easily specify their static semantics. Concerning the rules specifying their behavior such elements can be treated like SVPs, i.e. we can either refuse to formulate semantic rules for them, which is the most accurate way of treating these elements but it results in fragmented LTSs, or we can provide some default interpretation which tries to capture at least the outline of the probably intended behavior. Since the latter approach highly depends on the application domain of UML, we recommend placing such default definitions into extra packages which can be incorporated in the semantics definition if requested.

We can summarize that Dynamic Meta Modeling as a specification technique fulfills all requirements we posed. For some of these requirements, however, the concrete level of fulfillment depends not on the specification technique as such but on the concrete language definition formulated in DMM.

In the next chapter we demonstrate how such a DMM specification is constructed for a subset of the UML.

Chapter VI

Case Study: Formalizing UML Activity Diagrams

In this chapter we will demonstrate how the technique of Dynamic Meta Modeling can be applied to define the semantics of behavioral UML diagrams.

In UML 2 the three main behavioral diagrams are the Statechart, the Sequence/Communication diagram, and the Activity Diagram. Statecharts (in their UML 1.x incarnation) have been formalized using DMM in [Hau01, EHHS00]. Sequence Diagrams have been the running example in a number of papers on DMM, including extensions of sequence diagrams incorporating timing information [HHS01, HHS02a, HHS04]. For the case study here we focus on the formalization of UML Activity Diagrams using the techniques presented in the previous chapters. For readers unfamiliar with Activity Diagrams a short introduction to the notation is provided in Appendix A.

The case study serves both as a proof of concepts and as a further examination and proof of claims made in preceding chapter of this thesis. Its presentation proceeds in three sections:

- ◆ Section VI.3 *discusses* the current state of the semantic description for Activity Diagrams as found in the UML Specification. Here we demonstrate the shortcomings of the current style of UML's semantics definition by pinpointing contradictions, inconsistencies and omissions. Especially the intended token flow semantics of Activity Diagrams and its relation to Petri-Nets are discussed in depth. The section concludes with an outline of our interpretation of Activity Diagrams.
- ◆ Section VI.2 provides excerpts from the formalization of this interpretation in DMM. The complete DMM specification of UML's Activity Diagrams is given in Appendix B.
- ◆ Section VI.3 reviews the DMM system for Activity Diagrams with respect to its qualities. While Sect. V.5 discusses such qualities for DMM in general, we can focus on one concrete specification here and gain additional results.

VI.1 Eliciting the Semantics of UML Activity Diagrams

There is a vast difference between defining semantics and formalizing them. Defining semantics means determining the meaning of constructs, formalizing them is just the activity to write these meanings down in a formal way. The semantics of UML have been defined by the UML specification. Thus, to formalize them, we have to examine the text of the specification to elicit the intended meaning of Activity diagrams captured therein. In the process of this close look at the UML semantics description we uncover a number of deficiencies of the specification. The existence of these deficiencies is a strong argument for the introduction of DMM as a formal specification for the UML's semantics.

That the provided semantics description for Activity Diagrams would be precise and unambiguous was improbable from the beginning. Activity Diagrams in their UML 2 incarnation are a completely new formalism. Given their substantial complexity (the description covers roughly 100 pages), the occurrence of omissions and inconsistencies was almost inevitable. Furthermore, the intended integration of ideas from different application areas and at very different levels of abstraction is a very difficult task. Even in the Finalization Task Force (the body concerned with creating the specification and fixing flaws in it) there is deep suspicion that the amalgamation of these concepts does not work in a smooth way. To quote Jim Rumbaugh (e-mail):

I think there may well be issues related to activity semantics (especially the "partial token" options that seem to me to be unsound, but they were pushed because they are familiar to the business modeling community [and should we expect precise semantics from that source?])

Currently, there are over 100 issues recorded in the OMG's issues database related to the specification of Activity Diagrams, many of which also deal with semantic issues¹.

Subsection VI.1.1 lists a number of (rather local) deficiencies of the UML specification for Activity Diagrams. Going vastly beyond these flaws is the confusion around the central semantic concept of UML Activity Diagrams: token flow. We investigate this concept in depth in Subsect. VI.1.2 and evaluate alternatives for its realization. The conclusions from this investigation are summarized in our (informal) interpretation of Activity Diagrams provided in Subsection VI.1.3. Note that we intentionally do not refer to the formalization concepts of DMM in this section as we are not aiming at an interpretation that fits easily into DMM but we rather try to elicit the "real" semantics of UML Activity Diagrams using

¹Note that during the writing of this thesis, the OMG has advanced the UML 2.0 superstructure specification from the status Final Adopted Specification [Obj03d] over an intermediate Convenience Document [Obj04] to a Published Specification [Obj05]. This finalization process entailed changes to the specification to address noted issues [FTF]. Our own research has contributed to the submission and resolution of issues 3391, 6512, and 7221 [FTF]. Some of the problems mentioned in this thesis have thus found resolution in the finalized version. The statements in this chapter are mostly based upon the Final Adopted Specification of the UML 2.0 Superstructure [Obj03d]

technology independent discourse only. Formalizing these concepts in the DMM framework is the task of the next section.

VI.1.1 Deficiencies in the Definition of Activity Diagrams

In the UML specification for Activity Diagrams a number of deficiencies can be identified which hinder a precise interpretation and formalization.

Elements without concrete notation

Several elements that have an impact upon the execution of an Activity Diagram do not have a prescribed syntax. Examples for this are edge transformations or join specifications. In the abstract syntax, these elements are encoded as an **opaque expression**, i.e., an expression given in a language external to UML. Without a defined syntax and without limitations to what can be included in these specifications, it is impossible to integrate these elements in a formal semantics. As a direct consequence, we do not support guards, thus rendering all decision nodes to points of non-deterministic choice.

Also without a concrete notation are the elements of the Structured Activities package. This package provides high-level notations to encode loops, choices etc. in a single notational element. Meant for concise representations of algorithms, this part of Activity Diagrams is not supplied with any concrete notational symbols and is thus rather useless at the moment.

Missing elements

An example for missing elements in the syntax of the UML is the connection between Pins and Parameters. If an activity contains an action node which requires input data to proceed, this is denoted by one or more input pins. If the processing of this data is in turn specified by a behavior (i.e., the action is a call action) these inputs are regarded as parameters of the behavior. One would certainly expect that the information passed as parameters to the called behavior are identical to the objects that arrived at the input tokens of the calling action. But the UML does not allow for defining this connection, neither visually nor in the meta model. Bock proposes some correspondence relying on the order of pins and parameters but also points out that not even this is possible in the current incarnation of the meta model ([Boc03b], footnote 8).

Contradictory semantics

With some elements, the provided rationale for their existence and their semantics description are at odds. The `ForkNode` can serve as an example: The rationale ([Obj04] p.319) states that " *Fork nodes are introduced to support parallelism in activities. [...]UML 2.0 activity forks model unrestricted parallelism.* " This does especially imply that the different flow emerging from a fork node should be able to execute independently. The semantics for fork nodes

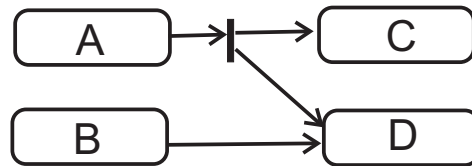


Figure VI.1: Example for an implicit dependency with standard semantics of fork nodes

([Obj03d] p.318) provides the following restrictions: ” When an offered token is accepted on all the outgoing edges, duplicates of the token are made and one copy traverses each edges. “ According to the traverse-to-completion semantics a token can only be accepted by a downstream object or action node. An action node will only accept tokens, if it has offers on all of its inputs. Thus, regarding the example presented in Fig. VI.1, the action D can only accept tokens, if A and B have finished executing. Action C should -according to the rationale- be able to execute, once A terminates. But the semantics of the fork node will only let tokens pass, if *all* of the outgoing flows will accept tokens, thus making C also dependent on the termination of B.

This effect is not only against the stated rationale of the fork node element, it is also quite unintuitive, as dependencies are supposed to be revealed by flows in Activity Diagrams and not hidden implicitly.

As a result of this analysis, we submitted an issue to the Finalization Task Force and cooperated with Conrad Bock in drafting a new semantics for fork nodes. In the proposed² solution a fork node will have the ability to buffer tokens for currently unavailable flows, provided *one* of its outgoing flows accepts the offered token.

Another contradiction can be identified in the rationale/semantics of a DecisionNode with an attached decision input behavior. The rationale given for this construct is [Obj04] p. 288: ”Decision input behaviors are introduced to avoid redundant recalculations in guards.“

One of the illustrating examples offered by the UML specification is reproduced in Fig. VI.2. Here, a complex condition is evaluated (once) in the decision input behavior and the subsequent guards at the outgoing flows can simply test on the result of this evaluation. Yet, the semantics of decision nodes caution ([Obj03d] p. 287):

If a decision input behavior is specified, then each token is passed to the behavior before guards are evaluated on the outgoing edges. The output of the behavior is available to the guard. Because the behavior is used during the process of offering tokens to outgoing edges, it may be run many times on the same token before the token is accepted by those edges.

This warning against multiple evaluations is directly contradictory to the ratio-

²at the time of writing the ballot on this proposal has not yet been held

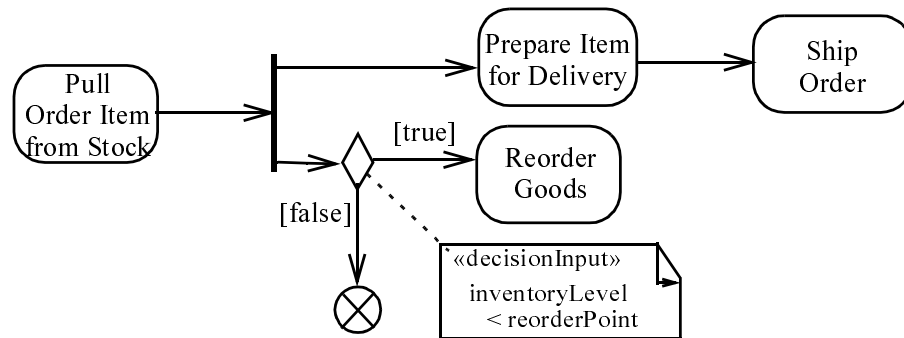


Figure VI.2: Example for a decisions node with decision input behavior (reproduced from [Obj05], p.351)

nale of avoiding redundant recalculations. It remains unclear which role decision input behaviors really play in Activity Diagrams.

VI.1.2 Token Flow in Activities

Token flow is the central semantic concept in UML Activity Diagrams. The UML specification should thus take utmost care to convey its intentions regarding this concept as clear as possible to allow for a precise understanding. Unfortunately, this is not the case.

Discussion the concept of token flow in UML's Activity Diagrams rather provides an excellent example for the problems of the current style of UML's semantics specification: Different statements in the UML specification allow for different interpretations of the token flow concept. There is no systematic way to prove which interpretation takes precedence over the others. There is not even a systematic way to detect all statements relevant to token flow. Users are thus reduced to arguing about the intended semantics with different opinions supported by different statements in the text. In this subsection we illustrate the extent of such an argument by (1) collecting relevant information for token flow from the UML specification, (2) investigating the probable intentions of the authors using external sources, and (3) evaluating different operationalizations of token flow to achieve the intentions elicited in (2).

Tokens are introduced in the semantics section of the metaclass Activity [Obj05], p.308:

A token contains an object, datum, or locus of control, and is present in the activity diagram at a particular node. Each token is distinct from any other, even if it contains the same value as another.

This statement is refined in a succeeding paragraph:

Tokens cannot "rest" at control nodes, such as decisions and merges, waiting to moving 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.

Thus, given the choice of activity nodes defined in the meta model, we can conclude that in a valid state of an activity diagram execution, each token is located either in an object node or in an action node. Yet, in the semantics section of Actions ([Obj03d] p. 280f) we find the following statement concerning the conditions to be met for an action execute:

Except where noted, an action can only begin execution when all incoming control edges have tokens, and all input pins have object tokens. The action begins execution by taking tokens from its incoming control edges and input pins.

Thus, control edges also seem to be able to hold tokens. The following paragraph reads:

An action execution consumes the input control and object tokens and removes them from the sources of control edges and from input pins. The action execution is now enabled and may begin execution.

While the precondition states that all incoming control edges must have tokens, the consumption of these tokens happens by removing them from the source of these edges. Note, that the source of a control edge may also be a control node, a place which may explicitly not hold a token.

Object edges on the other hand receive a different treatment:

An action execution is created when all its object flow and control flow prerequisites have been satisfied (implicit join). Exceptions to this are listed below. The flow prerequisite is satisfied when all of the input pins are offered tokens and accept them all at once, precluding them from being consumed by any other actions.

Note that the statement in the third sentence cannot hold for control flow prerequisites as, by definition, a flow targeting an input pin is an object flow. The quote introduces the notion of offering a token, which seems to target (at least) object nodes. Other clauses in the text imply that it may also target control edges:

When completed, an action execution offers object tokens on all its output pins and control tokens on all its outgoing control edges (implicit fork).

The semantics of Actions ([Obj03d] p.225) extend the notion of offering also to object edges:

Object and data tokens are offered on the outgoing object flow edges as determined by the output pins.

To summarize, on the basis of the UML specification it is neither clear where a token may reside, nor what exactly the process of offering entails. To clarify these questions, one has to go beyond the context of the official specification and take external sources into account. A primary source for information about the intended semantics of activity diagrams is Conrad Bock's article series in the

Journal of Object Technology [Boc03a, Boc03b, Boc03c, Boc04]. Conrad Bock is the member of the Finalization Task Force responsible for Activity Diagrams. Large parts of the specification have been written by him and in the articles in JOT he describes the new semantics independent from the format of the UML specification:

The terminology of the UML 2 specification is that the output pin "offers" the token to the outgoing edges, which in turn offer it to their respective targets. The traversal of the edge cannot take effect until all the elements between source and destination object node accept the offer, including the destination. This article calls the principle traverse-to-completion. [Boc04]

We take up the term *traverse-to-completion* (or TTC-semantics) and use it in this thesis. Following the quotation by Conrad Bock, offering and accepting these offers seems to be a way to construct a path between a given source node (which currently holds a token) and a possible target node. The rationale given for this kind of behavior is:

Preventing control nodes and edges from holding tokens ensures that values do not get "stuck" when alternative paths are open. In any particular direction of flow it may take a long time to select tokens, decide how to route them, for backups to clear, and so on. Traverse-to-completion means that tokens move along the path of least resistance by going to the first available object node. Data and object values are always residing in object nodes or being operated on by actions, moving instantly between them when all the criteria along the path between source and destination are satisfied. The decision of where to route tokens may take time, but no tokens move until the decision process is complete. [Boc04]

To understand the reason for this choice of semantics one has to inspect one of the inspiring sources for the new activity diagram semantics: Petri Nets.

Petri Net Semantics

In 1961 Carl Adam Petri defined a formalism to model the execution of concurrent processes [Pet62]. This formalism was later named *Petri Nets* in his honor. A Petri Net³ is a directed graph with two kinds of nodes, named places and transitions. It is bipartite in that all edges must connect a place and a transition (in either direction). In Fig. VI.3 an example Petri Net is exhibited. Places are notated as circles, transitions as bars. For the purpose of reference, all nodes carry a name label in the figure.

The semantics of Petri Nets is provided by the so called token game [Rei85]: The state of a Petri Net (also called its *marking*) is determined by a configuration of tokens which exist in the places. Each place may hold zero, one or arbitrary many tokens. A transition is said to be *enabled* in a state, if all of its input

³Since Petri Nets have been an object of scientific study up to the current day, a multitude of variants has emerged. For the discussion here we use the very basic formalism of unbounded Place/Transition nets.

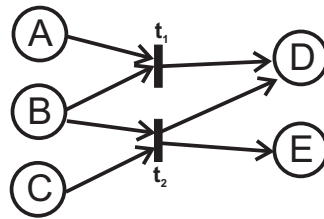


Figure VI.3: Example of a Petri Net

places (i.e., all places which are connected to the transition via an incoming edge) carry a token. From a place's point of view one could also say that a place is *offering* its tokens to the outgoing transitions.

The transition between states is carried out by the *firing* of an enabled transition. This firing consists of removing a token from each of its input places and putting a token on each of its output places (i.e. places which are connected to the transition via an outgoing edge).

Three properties that are important in respect to traverse-to-completion semantics of activity diagrams:

- (PN1) Transitions capture all of their input tokens in one atomic step. A partial consumption of tokens does not occur.
- (PN3) Tokens will only be consumed by an enabled transition, i.e., a transition which can and will fire.
- (PN3) If multiple enabled transitions require the same token (in the example in Fig. VI.3 this might happen if only one token was lying on places A, B, and C respectively) the firing of one transition (e.g., t_1) will consume the token and thus make it unavailable for the other transition (t_2), possibly disabling it. This effect is called *token competition*.

Comparing Petri Nets to Activity Diagrams

While the simplicity of Petri Nets allows for a clean and precise semantics and useful analysis results, they are a fairly low level formalism as compared to other behavior models. Activity Diagrams provide a lot of higher-level features which allow users to encode complex semantics in a single element. Still, Activity Diagrams are supposed to have "a Petri-like (sic!) semantics" ([Obj05] p.292). One possibility of substantiating such a claim would be a mapping from Activity Diagrams to Petri Nets. The UML specification does not provide such a mapping and the opinions on how to establish it vary. In an email message, Jim Rumbaugh suggested:

edges→*places*; a token on a edge represents the static presence of a value, which is how I have always viewed Petri net places. *actions*→*transitions*; that corresponds to a PN transition taking inputs and producing outputs. Activity nodes are a mixed lot; some

(such as object nodes) correspond to places, others (such as control nodes) correspond to transitions. I don't think it was a very elegant modeling job, from that point of view, but it works at a low level.

Another opinion by Conrad Bock (again by email):

PN's have a traverse-to-completion principle between places. UML built on this by generalizing transition to other control nodes, and using the transition semantics for action inputs.

This statement indicates that, rather than mapping elements of Petri Nets to those of Activity Diagrams, the authors intended to build the semantic properties of Petri Nets into Activity Diagrams:

- (AD1) Actions will capture all of their input tokens in one atomic step. A partial consumption of tokens does not occur.
- (AD2) Tokens will only move down a completely open path, i.e. move to an action which can and will execute or an object node which will buffer the token.
- (AD3) If multiple paths originate in an object node, they compete for the token.

Yet, the syntactic structure of Activity Diagrams is quite unlike Petri Nets. Petri Nets are always bipartite graphs with Transitions and Places forming the two groups of nodes. Thus, the firing of a transition affects only its local context, i.e. the places which are directly connected to it. Activity Diagrams are much more flexible as they consist of three basic types of nodes and place no restrictions on the way these nodes are connected. It is possible to construct large Activity Diagram fragments consisting only of control nodes and activity edges. We call these fragments *control structures*. In an activity diagram containing control structures, the enforcement of (AD2) can become extremely complex as the search for an open path might include branching as well as merging of flows. A detailed inspection of these problems is given in the next subsection.

Despite the syntactic differences, Petri Nets are still semantically close enough to Activity Diagrams to attempt a denotational semantics. The most complete approach to this has been published by Störrle in several papers [Stö05a, SH05, Stö04c, Stö04a, Stö04b]. Störrle uses Colored Petri Nets as the semantic domain and maps the elements of activity diagrams into this formalism (the mapping is depicted in Fig.VI.4). In a combination with ideas for Procedural Petri Nets he achieves a formalization which allows for multiple invocations of an activity concurrently without interference of the tokens and without copying the net. However, this formalization does not address the problem of enforcing property (AD2). Since several control nodes as well as most edges are mapped to places or structures involving places, tokens move stepwise, resting at control nodes and possibly moving into "dead ends". This is not a problem of this particular mapping, but rather a general issue for denotations to Petri Nets on the element level. Since control structures enforce a coordination of remote elements in the net, they either have to be translated as a whole into a Petri Net pattern or higher level control structures must be used with the Petri-Net (e.g., zero-safe nets [BM97]).

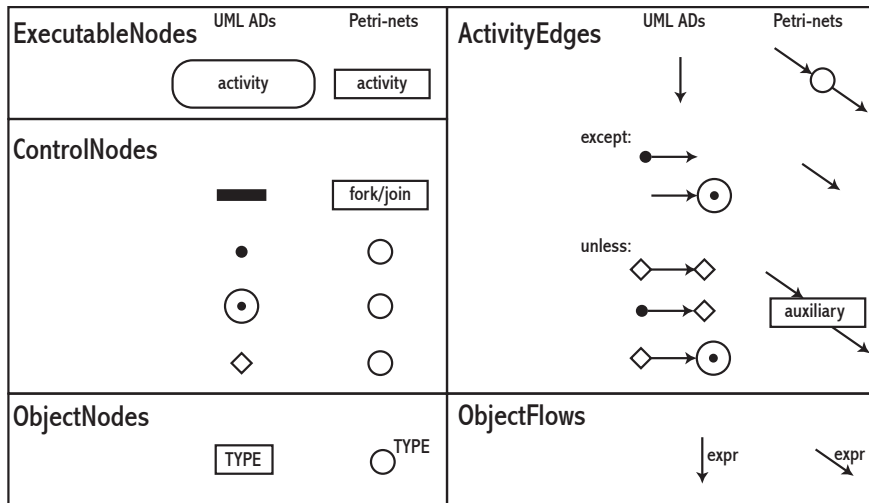


Figure VI.4: Proposal for a mapping of Activity Diagrams to Petri Nets (reproduced from Störrle [Stö05a])

In [SH05] we have investigated the problems of mapping Activity Diagrams to Petri Nets in greater depth. The conclusion of this investigation is that many advanced concepts introduced by Activity Diagrams (exceptions, streaming, traverse-to-completion) do not only require a new Petri Net formalism to express their semantics properly, but that in many cases this extension also requires modifications to the basic intuitive mapping. At the time of writing, no reasonably complete denotational mapping to Petri Nets (especially none taking (AD2) into account) has been published, let alone one that retained a simple and intuitive core.

Operationalization of Traverse-To-Completion Semantics

The properties (AD1)-(AD3) given in the previous subsection are high level semantic properties of Activity Diagrams. For the evaluation of a concrete Activity Diagram, these properties have to be broken down into detailed and precise semantics for the elements comprising an Activity Diagram. The current formulation of the UML specification fails in this task (as discussed before). In this section we discuss several alternatives in providing an operationalization of the intended properties.

To fulfill the intended properties (AD1)-(AD3) for Activity Diagrams, the authors of the UML (particularly Conrad Bock) have envisioned an evaluation mechanism based on tokens and offers. Offers are never defined in the UML specification and there are basically two interpretations for them: On the one hand *offering* can be regarded as a process of evaluation. On the other hand offers can be seen as run-time entities in the activity diagram. To exemplify the differences between both interpretations, we introduce a common example below. First, however, we need to precisely define the terms we are discussing:

Definition 35 (Activity Graph)

An Activity Graph (AG) is a tuple $\langle CNodes, ONodes, ANodes, Edges \rangle$ where
CNodes is the set of Control Nodes
ONodes is the set of Object Nodes
ANodes is the set of Action Nodes and
Edges is a set of tuples $CNodes \cup ONodes \cup ANodes \times CNodes \cup ONodes \cup ANodes$ ⁴

Definition 36 (Control Structure)

A Control Structure is a tuple $\langle CSNodes, CSEdges \rangle$ that is part of an enclosing AG $\langle CNodes, ONodes, ANodes, Edges \rangle$ with

$CSNodes \subseteq CNodes$

$CSEdges \subseteq Edges$ such that

$$\forall \langle A, B \rangle \in Edges: A \in CSNodes \vee B \in CSNodes$$

$$\Rightarrow \langle A, B \rangle \in CSEdges,$$

$$\text{and } \neg \exists \langle C, D \rangle \in CSEdges : C \in (CNodes - CSNodes) \\ \vee D \in (CNodes - CSNodes)$$

This definition means that a control structure is the maximal part of an Activity Graph that is only comprised of control nodes and edges. Note that control structures are not proper subgraphs as they contain dangling edges (the "outer" edges connecting the control nodes to surrounding action or object nodes).

Definition 37 (Input and Output)

The input of a control structure (CS_{input}) is the set of all object and action nodes that may supply tokens to the control structure. Formally:

$CS_{input} = InNodes \subseteq (ONodes \cup ANodes)$ such that

$$\forall \langle M, N \rangle \in CSEdges : M \in CSNodes \vee M \in InNodes$$

The output is defined symmetrically:

$CS_{output} = OutNodes \subseteq (ONodes \cup ANodes)$ such that

$$\forall \langle M, N \rangle \in CSEdges : N \in CSNodes \vee N \in OutNodes$$

The union of input and output is called the *context* of the control structure. The context complements a control structure to form a proper graph.

Definition 38 (Path)

A path is a connection inside of the control structure, recursively defined as

$path(A, B)$ with $A, B \in CSNodes$ holds, iff $\langle A, B \rangle \in CSEdges$

or $\exists C \in CSNodes$ with $\langle A, C \rangle \in CSEdges$

$\wedge path(C, B)$

⁴We abstract from the well-formedness conditions imposed by the UML specification.

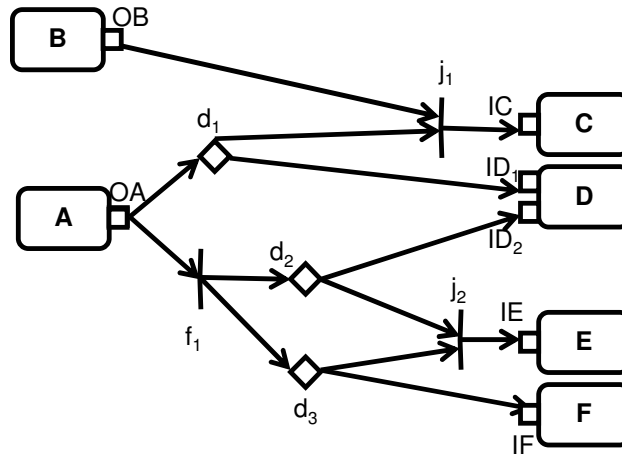


Figure VI.5: Example of a Control Structure

We say a node A is *downstream* from a node B , iff $path(B, A)$. Conversely, a node A is *upstream* from A , iff $path(A, B)$. Note, that Activity Graphs can contain loops and thus a node may simultaneously be up- and downstream from another. While a path is a structural property of an Control Structure, the term *open path* is used if a path can be traversed by a token in the current situation, taking all conditions into account.

In Fig. VI.5 a sample Activity Diagram is provided. For reasons of reference, we also labeled the usually unnamed control nodes and pins. Applying the above given definitions to this example yields the following sets:

- ◆ $ANodes = \{A, B, C, D, E, F\}$
- ◆ $CNodes = \{d_1, d_2, d_3, f_1, j_1, j_2\}$
- ◆ $ONodes = \{OB, OA, IC, ID, IE, IF, IG\}$
- ◆ $Edges = \{\langle OB, j_1 \rangle, \langle j_1, IC \rangle, \langle OA, d_1 \rangle, \langle d_1, j_1 \rangle, \langle d_1, ID_1 \rangle, \langle OA, f_1 \rangle, \langle f_1, d_2 \rangle, \langle f_1, d_3 \rangle, \langle d_2, ID_2 \rangle, \langle d_2, j_2 \rangle, \langle d_3, j_2 \rangle, \langle d_3, IF \rangle, \langle j_2, IE \rangle\}$

and we can identify one control structure consisting of

- ◆ $CSNodes = \{d_1, d_2, d_3, f_1, j_1, j_2\}$
- ◆ $CSEdges = Edges$
- ◆ $InNodes = \{OB, OA\}$
- ◆ $OutNodes = \{IC, ID, IE, IF, IG\}$

It is obvious that these sets satisfy the properties required for a Control Structure and that no other such structure exists in the example.

To actually determine which paths are open through this control structure (assuming tokens to be placed on OA and OB), we need to evaluate it. Intuitively, the following flows should satisfy all properties of activity diagrams:

- ◆ Action C gets both tokens from A and B.
- ◆ Action E gets the token from A.
- ◆ Action F gets the token from A and a copy of that token is placed at the non-successful outgoing flow of f_1 .

We now discuss different alternatives for obtaining this intuitive' result in a systematic way.

Evaluation semantics A first possibility for realizing the intended properties of activity diagrams can be called *evaluation semantics*. Evaluation semantics defines the state of the activity diagram as one where all tokens reside in action or object nodes. An evaluation algorithm is employed to calculate a possible succeeding state. In this semantics, offering is a step in the execution algorithm but not in the state of the diagram execution. The advantages of this interpretation are that activity diagrams are always in a defined state because the computation would be atomic.

Starting from a given token at a node in the input, the evaluation would try to find a path through the succeeding control structure, possibly employing backtracking to support non-deterministic choices. Classic depth-first or breadth-first strategies can be employed to traverse the control structure. If we assume an output token of Action A to be the starting point of the evaluation, the evaluation could visit d_1 and detect an open path to Action D.

When encountering nodes that synchronize several incoming flows (actions and join nodes) the evaluation becomes difficult, however. In these situations, a backward search over the other flows would be needed to find other available tokens to completely satisfy the synchronization criteria. Take, e.g., an evaluation that has detected the partial path A, f_1 and d_2 . To decide whether j_2 is a valid next step, the algorithm needs to make sure that d_3 can also deliver a token. As d_3 is a control node which cannot hold a token, the backward traversal continues toward possible sources of the needed token.

The evaluation of edges against their flow direction comprises several problems. One is that every element needs to have two semantics: One for "forward" and one for "backward" evaluation. Not only does the UML specification only provide the "forward" direction and the "backward" part has to be constructed, but it also has to be constructed in a way as to be consistent with the "forward" interpretation. For a correct interpretation, users would have to keep both semantics in mind. Branching (upon encountering merges) and reversing of search direction into forward evaluation again (upon encountering forks) may occur. Backward evaluations become impossible if transformation specifications are used. Transformation specifications can adorn edges and provide information on modifications to each passing token. These modifications are not necessarily reversible and thus an evaluation could not determine which input tokens would yield output tokens.

To summarize, evaluation semantics promises to realize all intended properties of activity diagrams but synchronizing elements pose great problems for this style of evaluation.

Offer semantics The interpretation provided by Conrad Bock is called *offer semantics* in this thesis. Two statements (taken from an email) outline his opinion on the topic:

*The offers act like "tentative" tokens, laying out a possible path the token can choose from. [...]
Offers "buffer" in the sense that they can remain at a control node until withdrawn.*

In this semantics, offers are entities and thus part of the state an activity diagram can be in. Offers represent the possibility that a token might flow down from the node where it is resting now to the elements where the offer now resides. In this view, offers cache partial results of the evaluation. Since these results can be buffered, the evaluation of synchronizing elements becomes trivial, as they once again only have to check their local context for the right conditions. Thus the problem of backward evaluation is avoided.

Yet, another fundamental problem occurs with this kind of semantics: Since offers embody cached evaluation results and are part of the system state, it has to be ensured that they are kept up to date with a concurrently changing environment (which might influence the evaluation results). An obvious change would be that the base token of an offer is accepted by some element. All other possible ways the token might have moved become instantly invalid. This must be reflected by removing the respective offers. Changes in the environment of the Activity Diagram might also modify data values that have been used to evaluate guard conditions. As these changes happen outside of the scope of the activity diagram and their impact cannot be determined without a complete reevaluation of the net, it is hard to ensure that offers remain valid with respect to these changes. The UML authors, however, chose to disregard this problem, as the following quote by Conrad Bock (personal email) states:

Only modelers using token competition will need to know about the token/offer distinction. They will naturally distinguish the decision to move a token (which might take time) versus the actual movement, because the timing of the decision cause race conditions. UML just says they need to understand offers to structure the decision process.

Actually, I think this is an interesting area to explore. Decisions to go down a certain path might be revoked. This is usually addressed by transactions, and workflow has the concept of compensation, which is a way of "undoing" a transaction that is already completed. We decided not to bring this into UML until we understood it better, but it does have some tools for this, such as exception parameters and protected nodes.

Thus, the offer semantics in UML allows for concurrent changes to past decisions without the offers becoming invalid. Accepting an offer then means a "jump" of the base token to the accepting place rather than a flow of the token along the path taken by the successful offer (since this path might not be open any longer).

Since this evaluation style is the one advocated by the UML's authors and the locality of decisions is easy to understand, we proceed to investigate this

interpretation of Activity Diagrams.

The general framework of token/offer semantics can be realized in various ways. Once again, details are neither supplied by the specification nor by supplemental publications.

Offers at places and at edges It is not yet clear on which elements an offer can reside. As offers represent tokens, they obviously have to be able to exist on a certain place (see also Conrad Bock's quoted remark above). Rumbaugh's clear statement that activity edges resemble Petri Net places (see above) can be used to justify the existence of offers also on edges. Since edges can impose quite complex conditions on the tokens moving over them, it might be beneficial to separate the evaluation of these conditions from the evaluation of conditions imposed by the target node. This modularity is one of our goals, thus we allow offers to reside on edges. However, by buffering offers on edges, additional steps are introduced into the evaluation process and one has to take care not to introduce additional and unwanted effects due to the increased possibility of concurrent interactions. To avoid effect like the "overtaking" of offers on an edge, we allow only one offer to reside on an edge.

Static versus moving offers Offers can be seen as static information, i.e., they represent the fact that the element they currently reside upon is connected by an open path to the element that holds the base token of the offer. In this interpretation, all elements along this path will also carry offers. The advantage of this interpretation is a flexible evaluation as all partial evaluations are stored and the algorithm can advance any of them. Even guards changing concurrently to the evaluation could be taken into account (at least in the enabling case). The disadvantage of static tokens is that a multitude of offers exist for one token. Whenever this token is removed, an extensive cleanup operation has to be initiated to delete all these—then invalid—offers.

The other possibility is that offers represent only the recently visited elements of this path and once the impact of this offer has been taken into account, it is removed. In this interpretation, offers "move" along the paths, possibly duplicating at forks. This interpretation of offers is closer to the formulations chosen in the semantics guide but it requires an algorithm that exhaustively evaluates elements which might result in multiple offers, i.e., a breadth-first search. The effort for cleaning up invalid offers is reduced with moving offers (in control structures that do not involve token competition or forks no cleanup is needed). We thus employ the concept of moving offers in our definition of AD's semantics.

Non-determinism Further variations can be identified in the treatment of non-deterministic choices. Non-determinism exists naturally in situations of token competition and may additionally occur because conditions can not properly be evaluated⁵. There are basically three ways of dealing with these situations: Either all choices are evaluated concurrently, a backtracking mechanism is used

⁵Currently, there is no prescribed syntax for the formulation of guard conditions

to exhaustively search all possibilities, or a non-deterministic evaluation mechanism is employed. Each of these methods comprises certain disadvantages:

Backtracking requires storing information on taken decisions and information to roll back the effect of failed decisions. For human consumption, this large overhead of organizational data is not suitable.

Evaluating all possibilities in parallel is certainly the most suitable approach (for Activity Diagrams) since it conforms best to the notion of "competition". Parallel evaluation would also allow the introduction of fairness conditions to the token competition⁶. The disadvantage of parallel evaluation of non-determinism is that a multitude of offers may exist in a state of the evaluation. Since these offers might originate in mutually exclusive decisions, an interaction between them must be prohibited. Regard the action D in Fig. VI.5. Even if both incoming edges carry offers, Action D should not execute under an parallel evaluation scheme since the offer at the upper incoming edge represents the fact that the upper edge leaving action A gets the token, while the offer at the lower edge entering action D depends on action A supplying its token to the lower outgoing edge. Thus, these offers represent two mutually exclusive possibilities and can never be synchronized by a join or an action with multiple inputs. Keeping track of these exclusivity constraints is possible but again introduces a large amount of data overhead.

Relying on external non-determinism is possible in our approach as graph transformation rules are applied non-deterministically on a host graph. We can thus assume that the complete LTS generated by the set of DMM rules will evaluate the 'right' choice anyway (if one exists). While we are aware that using this style of evaluation is not entirely correct with respect to the UML author's intentions (as unsuccessful paths will also be evaluated and will thus become part of the resulting LTS, forming "dead ends"), the benefits in reducing the overall complexity of the specification justify this deviation. We use this option in our definition of AD's semantics.

VI.1.3 Summary of Our Understanding of Activity Diagrams

At various points in discussion in the previous subsection we made decisions on the way we choose to interpret UML's activity diagrams, thereby clarifying open points or resolving inconsistencies. We will summarize these decisions here before we provide the formalization for this understanding.

Tokens Tokens are defined by the UML specification and we only give a brief overview here: Tokens represent either some data or computational object or a focus of control. Control tokens can only rest at action and object nodes. It is important to note that object tokens are only pointers to the underlying data or object, thus manipulations to the token (e.g. destroying it) will leave the data unchanged.

⁶currently such fairness conditions are not imposed in activity diagrams

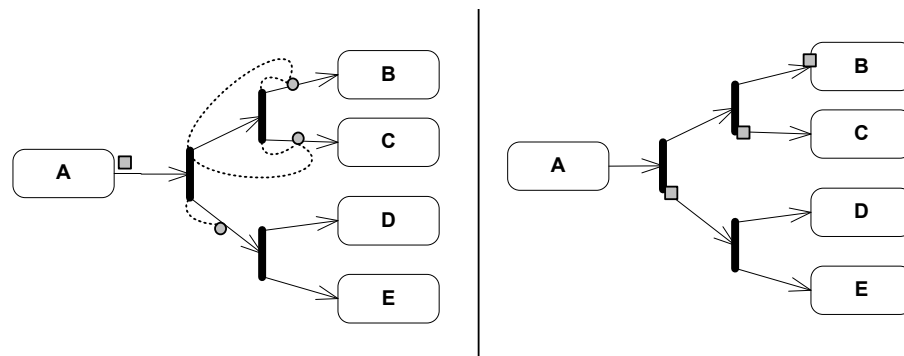


Figure VI.6: Example for the effect of accepting a spawned offer

Offers An offer is an entity that represents a token for evaluation purposes. Each offer represents the fact that the evaluation has found an open path from the current position of the base token(s) to the current position of the offer. This position may be at nodes as well as on edges. An offer "flows" in a step-wise fashion downstream through the activity graph. The movements of the offer underlie the conditions the different elements pose to its underlying token. Nodes can increase or decrease the number of offers in the graph by copying offers or merging them.

Action and Object Nodes An offer can only be accepted by a node whose type allows the base token of the offer to rest, i.e., an action or object node. Once an offer is accepted, the base token(s) are moved from their original position and placed on the accepting node. The number of departing and arriving tokens may differ due to forking and joining of tokens. All other offers referring to the base token(s) are immediately notified of the removal of their base token and cease to exist.

Control Nodes If an offer encounters a fork node, copies (spawns) of the offer are placed on each outgoing edge of the fork node. If one of these offers is being accepted, copies of its token must be enqueued at the edges of the other (non-successful) offers. Fig. VI.6 illustrates this process. The left hand side of this figure shows an activity diagram state with a token at the output of action A and three of its offers with the dotted lines indication their *spawnpoint(s)*. The right hand side of the figure shows the state of the same diagram after action B accepted the uppermost offer. All other offers have been withdrawn and tokens have been enqueued at the intermediate forks to enable the subsequent execution of the remaining actions.

If all incoming edges of a join carry an offer, these offers are removed and a new offer is being created on the outgoing edge. This new offer has the union of all base tokens of the removed offers as its base token and the union of all spawnpoints of the removed offers as its spawnpoint. Note that it is as of now unclear how different data tokens are joined (cf.[FTF], issue 7013), thus offers referring to data tokens will not be joined, unless they all refer to the same base

token (cf. [FTF], issue 6367).

Edges Edges can carry at most one offer at a time. The base token of the offer must conform to the criteria given by the guard of the edge⁷. If an offer has multiple base tokens, only one needs to be evaluated (since multiple base tokens need to be control tokens (cf. semantics of the join) and control tokens are indistinguishable).

VI.2 Excerpts from the DMM Specification of Activity Diagrams

Our intuitive understanding about UML's Activity Diagram has been successfully formalized in the DMM technique. The complete results of this formalization are presented in Appendix B. Here, we only show a few excerpts of this formalization to demonstrate the feasibility of the concepts introduced in Chapters III to V. In particular we show the package structure of the semantic domain meta model for Activity Diagrams (Subsect. VI.2.1), some of its classes (Subsect. VI.2.2), and the rules specifying the token/offer flow mechanism as outlined above (Subsect. VI.2.3).

VI.2.1 Package Structure of the Semantic Domain Meta Model for Activity Diagrams

The semantic domain meta model for Activity Diagrams contains more than 30 classes, despite not supporting all features of activity diagrams. It should therefore be subject to a high-level structurization as described in Sect. V.4. Fig. VI.7 provides an overview of the different packages we defined and their relationships.

We can see that a nice structure emerges in which core packages (which define central concepts like `Instance` or `BehaviorExecution` are imported and supplements by the more specialized packages of Activity Diagrams. All packages conform to the restrictions set out in Sect. V.4.

VI.2.2 Class Structure of the Core Activities Package

Zooming into the Core Activities package we find the class structure depicted in Fig. VI.8.

In this package the core concepts of Activities are captured. Replicated intensional classes are `Activity`, capturing an Activity specification build up from `ActivityElements`, which are `Nodes` and `Edges`. The extensional elements are concentrated on the left hand side of the figure, with `ActivityExecution` being the main class to capture the execution of an Activity. Such an `ActivityExecution`

⁷Since no notation is provided for the formulation of guard conditions and introducing one would go beyond the scope of this thesis, we treat all guards as being constantly true.

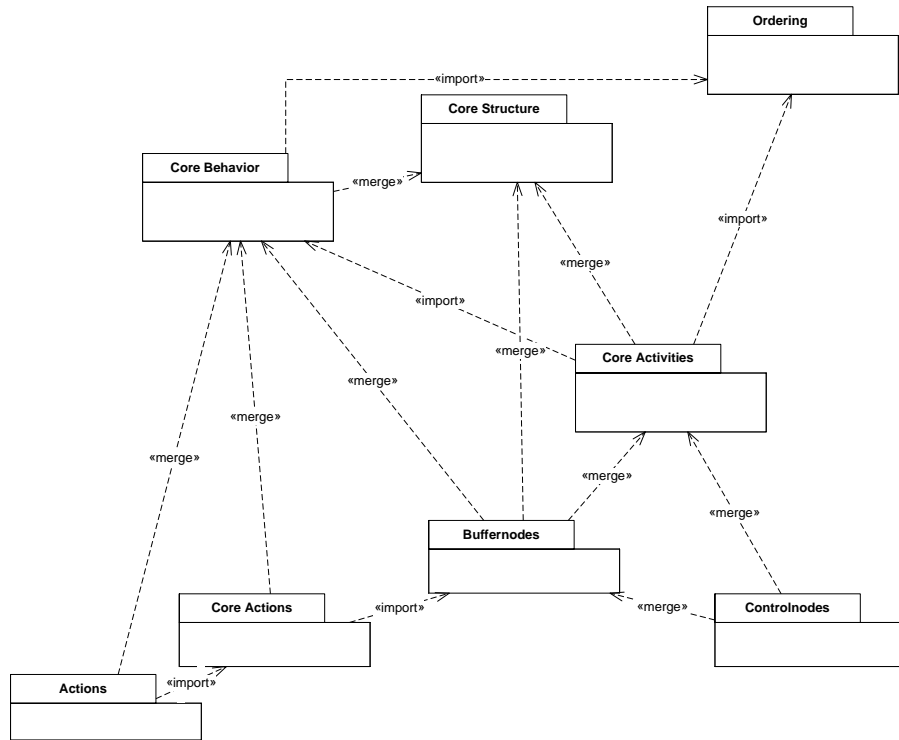


Figure VI.7: The package structure of the semantic domain's meta model for Activity Diagrams

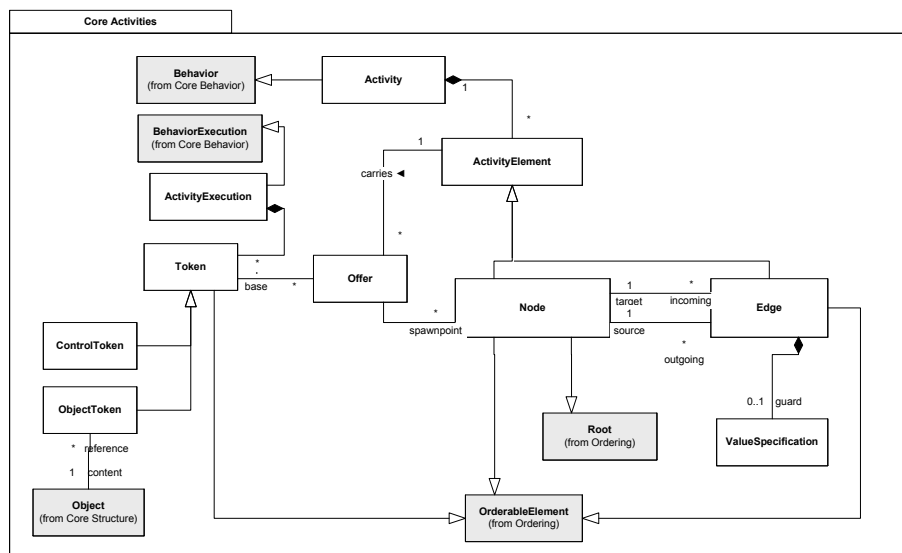


Figure VI.8: The contents of the Core Activities package

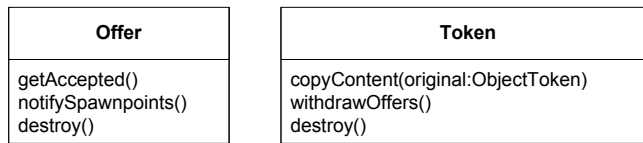


Figure VI.9: Details of the Token and Offer class

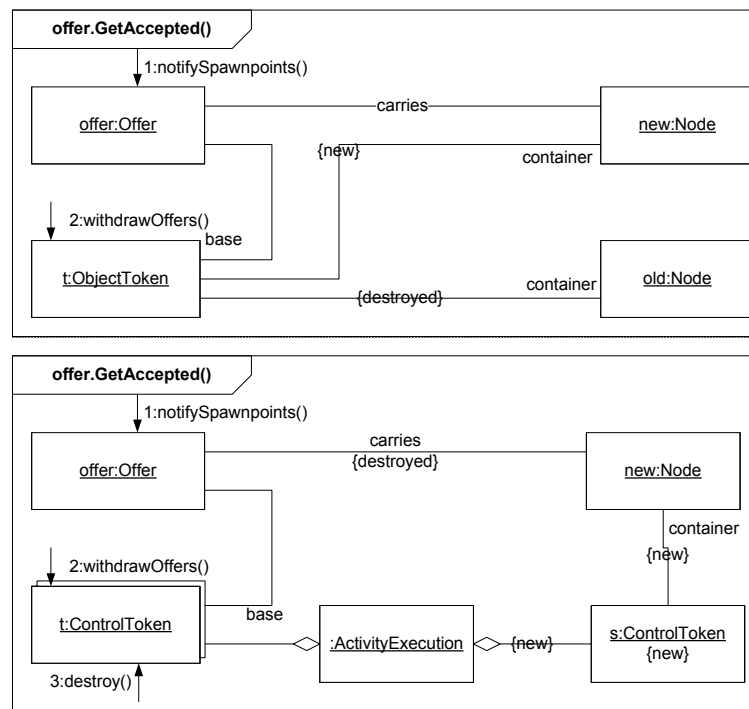
can comprise a number of **Tokens** which come either as a **ControlToken** or as an **ObjectToken**. All **Tokens** can form the base for an **Offer** which is moves over the elements of the activity. Note that the placement of tokens is not yet specified as tokens can be placed on **Buffernodes** only which are defined in a separate package. An **Offer** can designate a certain node as its **spawnpoint**, indicating that it has been spawned at this particular node.

The display of this class structure substantiates several claims we made in the conception of DMM:

- ◆ The semantic concepts become more precise. For an offer, e.g., we can clearly perceive that it is an entity and not an evaluation procedure. We can also see that an offer always has a base token but not necessarily *only* one.
- ◆ The replicated intensional elements are tightly coupled with their extensional counterpart. Thus the replication really makes sense as navigation between the relevant elements is easier than referring to the external semantic Relations.
- ◆ Replicated intensional elements undergo slight changes. If you compare the class **Edge** to its syntactical counterpart you will notice that UML defines two kinds of edges: **Dataflow** and **Controlflow**. These, however, are syntactically important concepts only as they are used for ensuring that complex paths are traversable by either control or object tokens. Semantically, these edges perform the same functions and thus the difference between them is not replicated here.
- ◆ Auxiliary elements are introduced to facilitate the formulation of concise rules. The package **Ordering** defines a mechanism to order sets for an iterative processing. We can see it being used in the **Core Activities** package by making **Token**, **Node**, and **Edge** **OrderableElements**. These elements can be ordered in some context, e.g., the set of outgoing edges from a fork node will be ordered. The class **Node** can also play the role of a **Root** for such an order.

VI.2.3 DMM Rules for Tokens and Offers

As Fig. VI.8 omits internal details of the classes, Fig. VI.9 provides these details for the **Token** and **Offer** class. We proceed to discuss the rules for some of their operations.

Figure VI.10: DMM Rules implementing the operation `offer.getAccepted()`

Operation `offer.getAccepted`

The rules implementing `offer.getAccepted` are shown in Fig. VI.10. We can see that there are two rules specifying this operation. The rules differ in that the upper rule describes the acceptance of offers based on an object token while the lower rule describes the acceptance of an offer based on one (or more) control tokens. In the former case we can observe structural manipulations in the graph which amount to moving the underlying token to its new location (deletion and re-creation of the container link). The rule also invokes the operations `notifySpawnpoints` on the accepted offer and `withdrawOffers` on the moved token.

For control tokens, movement of the token is realized differently (see the lower rule in Fig. VI.10). Since different flows of control may be combined at a join node, an offer may represent several control tokens at once. As control tokens do not carry any information, the simplest way of dealing with this situation is to delete the set of all base tokens of the offer and creating a new control token at the accepting node.

The rules for `offer.getAccepted` demonstrate several features of DMM rules:

Encapsulation of behavior The operation `getAccepted` describes the process of accepting an offer. All other behaviors which entail the acceptance of offers (e.g., the start of an action) simply invoke this operation which performs all necessary manipulations. The operation is, on the other hand, not directly responsible for the consequences the movement of the token

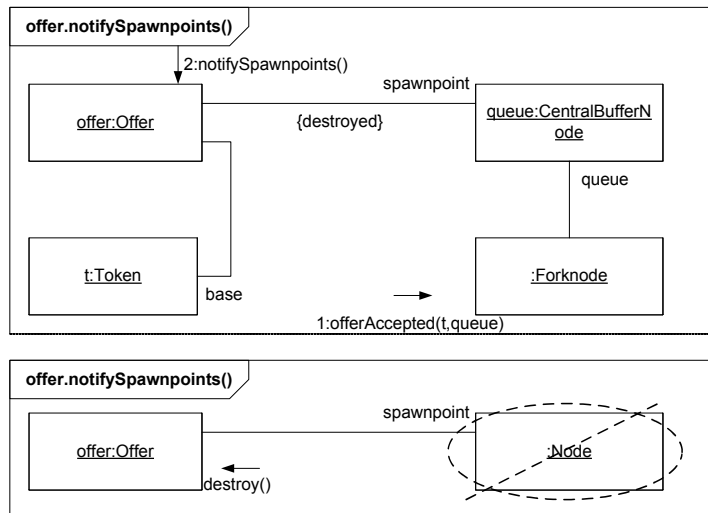


Figure VI.11: DMM Rule implementing the operations `offer.destroy()`

has on other offers. For this it simply invokes other rules dealing with these issues. In this way, responsibilities for complex manipulation can be distributed between different rules, keeping the single rules simple and avoiding redundancy and inconsistencies in the complete rule set.

Treatment of different cases The two rules for the operation `getAccepted` also demonstrate how different cases can be treated by a single operation. Other rules invoking `getAccepted` need not care about the different treatment of object and control tokens. Internally the different rules implementing the operation distinguish between these situations and treat them accordingly.

Execution Order The execution order of the rule's own manipulations and its invocations is also showcased by these rules. As the rule's own manipulations are carried out before the invocations of other rules, the direct deletion of the set of control nodes (i.e., labeling the UQS node `ControlToken` with `{destroyed}`) would not allow for the withdrawing of its offers anymore, resulting in an illegal system state (offers without a base token). Thus, an explicit destructor operation for the token class is used which is specified (by the sequence number) to apply after the invocation of `notifyOffers`.

Operation `offer.notifySpawnpoints`

The operation `notifySpawnpoints` is responsible for initiating the enqueueing of token copies at the relevant fork nodes. This operation is performed by the accepted offer. Two rules implement this operation (see Fig. VI.11). The lower rule in the figure states the fact that there is nothing to do for offers without spawnpoints. The upper rule applies to offers which have a spawnpoint. It navigates over the spawnpoint to its `forknode` and invokes the operation `offerAccept`

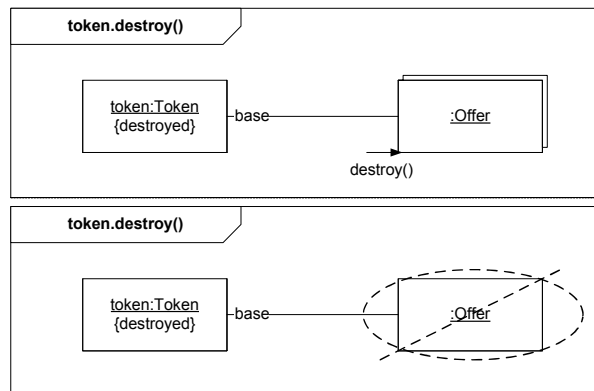


Figure VI.12: DMM Rule implementing the operation `token.destroy()`

on this node. Note that this rule is not part of the package `Basic Activities` but of `Buffernodes`.

Again, we can demonstrate several principles and features of DMM rule in these rules:

Recursive loops As indicated by the example in Fig. VI.6, an offer may have more than one spawnpoint. The upper rule in Fig. VI.11 thus calls itself recursively. By deleting the links to the already processed spawnpoints, it ensures that the number of unprocessed spawnpoints steadily decreases. If no more spawnpoint links remain to be processed, the lower rule applies and the recursion terminates. This is one way of processing sets of elements.

Using NACs for disjoint matchings The rules implementing the operation `notifySpawnpoint` differ in the existence or non-existence of a spawnpoint link. The lower rule uses the construct of a NAC to ensure that it only applies to situations in which there is no such link, thus ensuring that no conflict occurs between the upper and the lower rule.

Parameter passing The invocation of `offerAccepted` carries information in the form of parameters. The base token of the offer is being passed as it forms the original to be copied to the other edges and the spawnpoint of the accepted offer is passed to suppress enqueueing a token there.

Operation `token.destroy()`

The operation `token.destroy()` is a classical destructor. It is responsible for destroying the object it is called upon while ensuring a legal system state. As the token to be destroyed may have emitted offers which depend on its existence, these must be destroyed, too. The upper rule in Fig. VI.12 thus localizes all offers based on the token and triggers their destruction together with the token. Note that a UQS requires at least one match, this rule is not applicable to tokens without offers. Thus, the lower rule in the figure takes care of this situation.

The excerpts presented here give an impression of the way the DMM concepts are applied to formalize our understanding of Activity Diagrams. The complete specification is given in Appendix. B. We now move from concrete details of the specification to a more abstract evaluation of its properties.

VI.3 Discussion of the DMM Specification of Activity Diagrams

When discussing the qualities of the DMM approach and how it meets the requirements set of for semantics descriptions (see Sect. V.5), we point out that due to the flexibility of the approach, some of its qualities depend upon the concrete specification rather than on the general technique. In the following subsections we thus discuss these qualities for the concrete example of the DMM specification for UML's Activity Diagrams. In particular these include quantitative evaluations supporting the claim of understandability (Subsect. VI.3.1), modularity discussions (Subsect. VI.3.2), addressing the inherent concurrency in Activity Diagrams (related to the general quality universality, Subsect. VI.3.3), and finally the adequacy of the specification (Subsect. VI.3.4).

VI.3.1 Understandability

One requirement for the technique of Dynamic Meta Modeling is that it should provide a human-readable specification. While the diagrammatical nature and the UML representation of our rules provides an intuitive appeal, we need to discuss whether this appeal is retained in a full specification. Too large and too complex rules and rule sets will hinder the understandability of a specification. We thus evaluate relevant measurements of the DMM specification for Activity Diagrams in this subsection.

The DMM specification for the basic and intermediate elements of Activity Diagrams contains 73 DMM rules. This can be considered a rather large number at least in terms of comprehensibility by human readers. Aiding the comprehension of the rules is the fact that they do not come in a single set but are structured into operations of 35 classes which are themselves partitioned into 8 packages. No single operation is specified by more than 4 rules (`CallBehaviorAction.createSlot()` uses 4). Especially when rules are directly invoked, it should thus be rather easy for a human reader to locate the candidates for the next rule application.

Another question of size is the complexity of the single rules. The rule with the most elements is `parameternode.accept` with a total of 9 nodes, 12 edges and one invocation. Most complex in terms of using special control features is the rule `fork.spawnOffer` featuring two multinodes and a negative application condition. Finally, in terms of invocations `cba.execute` is the most complex rule by using three invocations. While these rules are certainly not trivial, their content is still easy to grasp.

We can thus state that the invocation mechanism of DMM rules indeed allow for small rules without creating overly bloated rule sets or too complex control constructs. To actually quantify how understandable such rules really are and which additional techniques might be used to aid the reader in comprehending the specification further research is needed. In particular, experimental studies with intended users of the technique have to be conducted.

VI.3.2 Modularity and Extensibility

The structurization of the semantic domain into 8 distinct packages which all obey the restrictions formulated in Sect. V.4 proves that modularity is achievable in a DMM system. In fact the rule set lend itself to this structurization rather easily in that it was originally conceived without the concrete packages in mind and was partitioned afterward with only minimal changes. The rather large number of «merge» relationships between the packages should not be a reason of concern since (as explained in Sect. V.4) these stem mostly from the introduction of new associations. In fact the only real behavioral extension of an element in another package is the addition of an additional rule to the operation `offer.notifySpawnpoints`. It can thus be stated that the necessary modularity can be achieved.

Extensibility on the other hand poses some problems. In Sect. V.4 restrictions are imposed on the way new specifications can extend old ones. In particular the overwriting and modification of existing rules is not possible. Upon reviewing the formalization of Activity Diagrams one can now state that it is possible to obey these restrictions when formulating DMM rules. Yet, the example of the class `Offer` also shows that this is not without problems. As the presentation in Subsect. VI.2.3 shows, the semantics of class `Offer` are (partially) tailored to support the way fork nodes are supposed to work. This now leads to the situation that the `Core Activities` package defines an association `spawnpoint` between `Offer` and `Node` which at this point does not make any sense since fork nodes are defined in the `Buffernodes` package. Unless fork nodes are involved in an activity graph, no instance of this edge will ever be created. Likewise, allowing multiple base tokens for a single offer is a design decision which can only be understood if joining of offers is taken into account. And not only the structural features but also the semantic rules have to take the extension into account: The rule for `offer.notifySpawnpoints` is explicitly designed in a way to allow a later extension. Thus the `Core Activities` package already provides design foundations which are required by other packages. This situation can be regarded as an indication that other concepts which we do not yet support also require the definition of some fundamental structures and thus an adaption of the core packages.

VI.3.3 Degree of Concurrency

One core feature of UML's activity diagrams is the expression of concurrency. The explicit modeling of concurrency (by using forks/joins) as well as the localized control in the form of independent tokens are its most distinguishing features in comparison to the other UML behavioral diagrams. Using DMM for

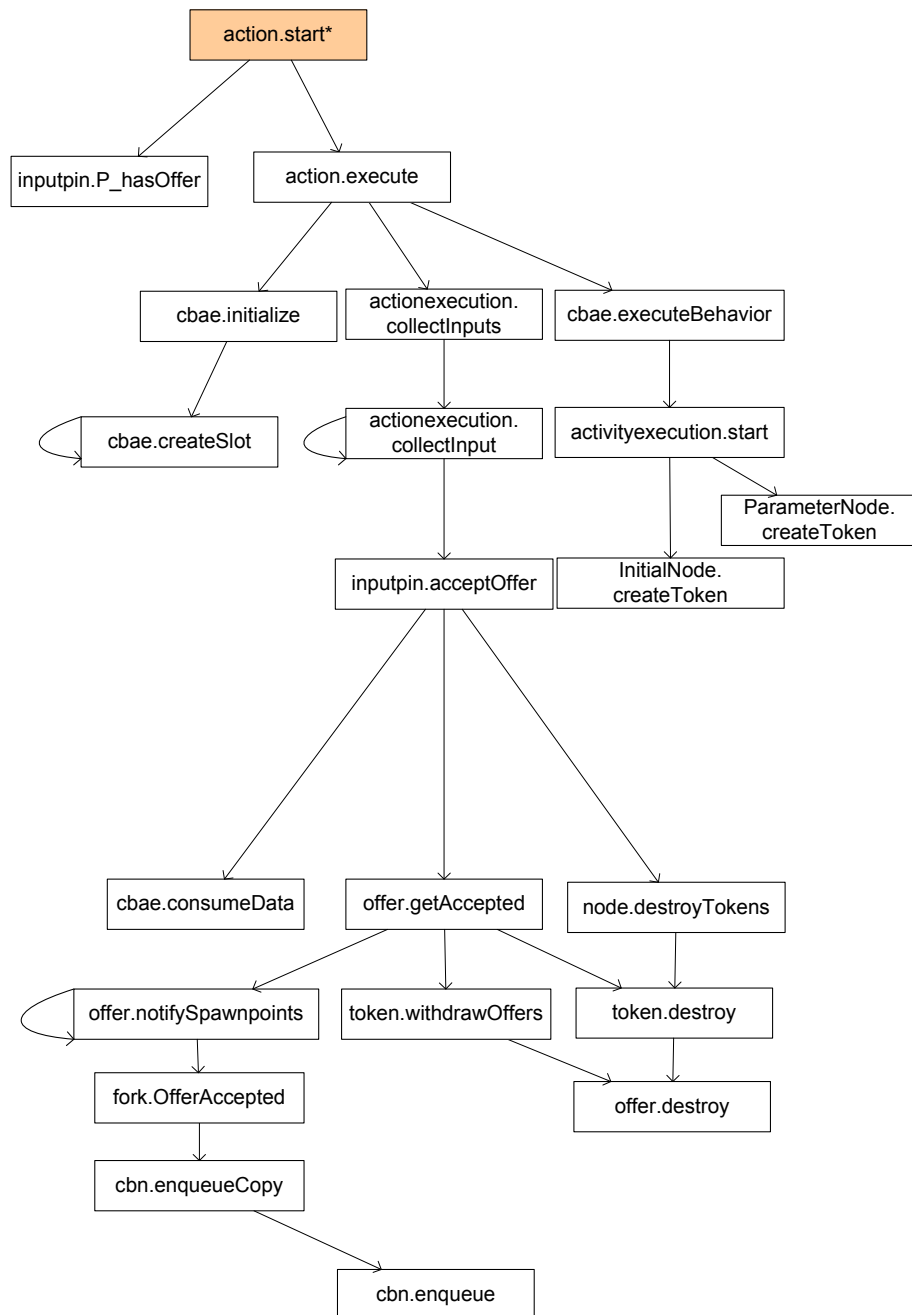


Figure VI.13: Action related part of the call graph of the DMM system for activity diagrams

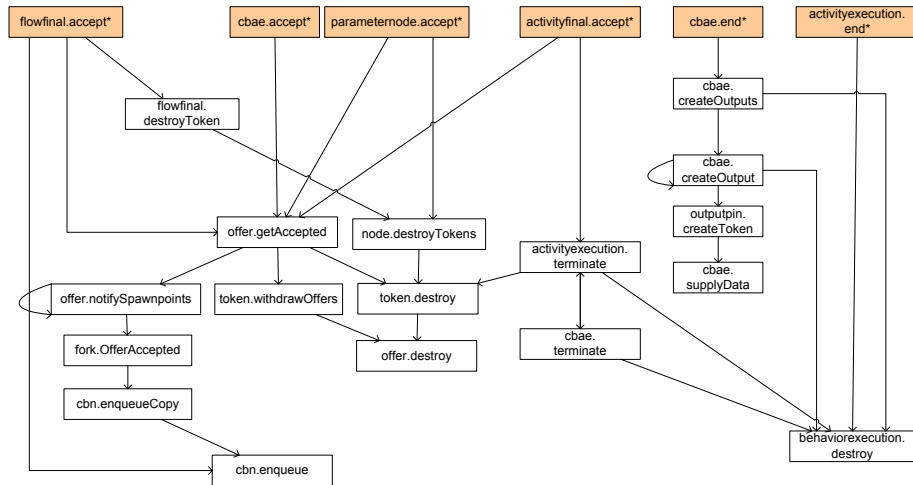


Figure VI.14: Part of the call graph of the DMM system for activity diagrams concerning behavior ending and acceptance rules

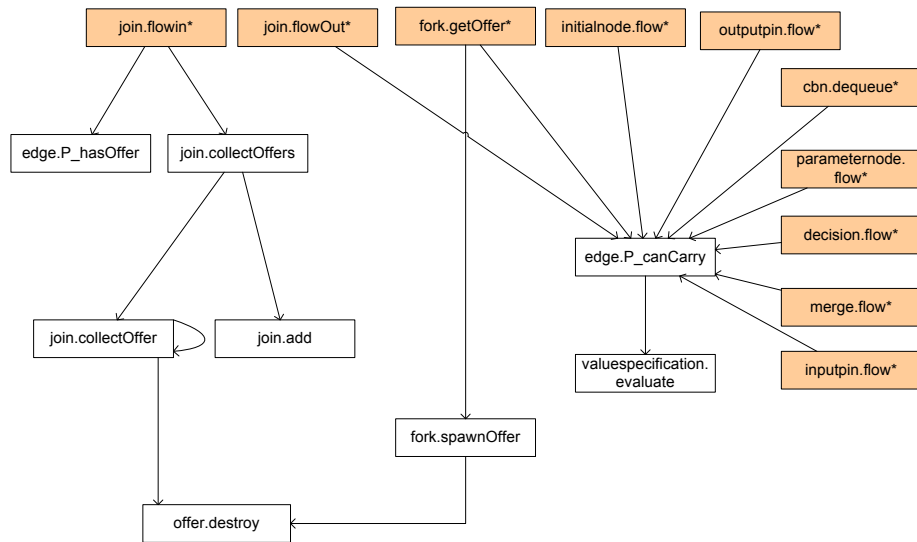


Figure VI.15: Part of the call graph of the DMM system for activity diagrams concerning flow rules

the specification of semantics, we can address concurrency by forming independent big-step rules which apply non-deterministically. The LTS then contains the set of all possible interleavings of the different concurrent behavioral steps. The granularity units for this interleaving are formed by the big-step rules (and their transitive invocation closure). Since the size of big-step rules is a modeling decision it is worthwhile to investigate the size of big-steps in our specification.

We do this by investigating the invocation graph of the activity diagram DMM system. Each node in this graph represents an operation in the semantic domain meta model (i.e., one or possibly multiple rules). Operations with Big-Step rules are shaded in gray and placed toward the top of the figures. A directed edge runs between two nodes A and B if one of the rules representing operation A calls operation B. Reflexive edges are possible and indicate recursive loops. The graph is divided into several figures for reading convenience. Figs. VI.13 to VI.15 show different portions of the graph.

We can observe that the operation `action.execute` has the most complex invocation tree. Much of this complexity is caused by the fact that we modeled the `CallBehaviorAction` which is one of the most complex action types. Altogether a total of 20 rules can be involved in executing the big-step operation `action.execute`. Since some of the rules have reflective edges, the number of actual rule occurrences might even be much higher (depending on e.g., the number of input and output pins and the number of offers emitted by tokens accepted by this action). An interpreter will thus spend a lot of time executing this rule and will not permit concurrent evaluations. However, the semantics of actions demand that all their input tokens must be consumed at once. It is thus impossible to divide this invocation tree since the detection of the necessary preconditions (`inputpin_hasToken`) and the consumption of the tokens need to happen atomically. The execution of the actual action (`cbae.execute`) could be separated from the invocation tree. However, we felt that this would not accomplish much to reduce the complexity. The effect of the invocations (i.e., the execution of the invoked activity) and the execution of the originally invoking activity happen concurrently.

Acceptance of tokens and terminating of executions (cf. Fig. VI.14) create invocation trees of moderate size. Much of the complexity here stems from withdrawing unsuccessful offers. Flows, the probably most commonly applied rules, have a very shallow invocation tree, most flows only rely on the predicate `edge.P_canAccept`. Thus, the evaluation of control structures can indeed pass in a very concurrent manner with an offer passing a single control node as its atomic steps. This is equivalent in size of granularity to Petri Nets where tokens passing a single transition form the atomic steps of the execution.

VI.3.4 Adequacy - Limits and Semantic Shortcuts

After discussing some more technical properties of the created DMM system, we also want to critically review the adequacy, i.e. the semantic "closeness" to the UML semantics description which we achieved with the formalization. In should be clear from the extended discussions in Section VI.1 that we tried our best to understand the intentions of the UML designers and act accordingly.

We are aware, however, that we cut a few minor corners with our formalization. These include the suppression of token duplication at outgoing edges from a fork with failing guards and the restriction of object token joining to tokens with an identical base object. We believe that the official semantics of the UML are not very convincing in these points and that their formalization would have caused an undue amount of additional specifications.

Another issue is the use of non-determinism in our formalization. We decided to solve token competition situations by placing an offer on one outgoing edge only. The same mechanism was employed in the case of decision nodes with multiple enabled outputs. If a complete transition system is generated from the DMM system, all possible decisions are evaluated. Thus, both elements work in the same way. Upon close inspection, this is not precisely the case in UML. Token competition situations are to be resolved by competing offers while a true non-deterministic choice happens at decision nodes. Our formalization does not reflect the intrinsic (if minuscule) difference in this point.

Overall, however, we can confidently state that our formalization does not only reflect the UML author's stated opinions (in the specification and in external publications) but that it also manages to stay rather close to the terminology and structure of the UML specification text. The only term which we explicitly introduced was the spawnpoint of an offer. All other extensional elements (action execution, offers, tokens, etc.) appear in the UML specification text. The semantic closeness is also indicated by the fact that the semantic mapping is rather straightforward in most parts. In fact, we believe that based upon the DMM specification, a clearer re-formulation of the UML specification text would be possible.

The case study of formalizing UML Activity Diagrams provides proof for the claims made in previous chapters. We can observe that in fact the specification of semantics in the DMM framework is not only possible but that it is also modular, extensible, and stays within reasonable limits in terms of size and complexity. Such qualities are, however, only achievable in a specification if the Language Engineer aims for them. In other words, it is perfectly possible to specify a DMM rule set for Activity Diagrams which does not contain these qualities at all. In the next chapter we thus provide a set of pragmatic rules which help future users of this technique to produce DMM rule sets of high quality.

Chapter VII

Pragmatic Guidelines for Formulating DMM Specifications

Languages (be they technical or natural) provide the means to express certain concepts. In this sense we can view Dynamic Meta Modeling as a language which allows the user to express semantics of visual modeling languages. In every sufficiently expressive language the problem arises that a vast number of expressions can be constructed to describe the same concept in different ways. This fact also holds true for DMM. A given set of semantic concepts can be formalized by very different DMM specifications. Potential users thus require guidance on the way an expression in the technique is to be constructed. This chapter provides such guidance for the creation of DMM specifications.

A Language Engineer employing DMM to define a new Visual Modeling Language is basically faced with the task of at first identifying the semantic concepts that should be expressed by this VML. These concepts can then be formalized using the techniques of DMM. While the formalization process yields a formal specification, in itself it is not formalized. Bertrand Morand [Mor99] remarks:

It is more an activity that belongs to experimental sciences even though it uses logic resources.

Does the term "experimental science" then imply that Language Engineers may obtain DMM semantics specifications only by means of trial and error? Fortunately, no! While there is no precise algorithm to describe the creation of DMM specifications, assistance can be provided to a Language Engineer in a number of ways.

In Software Engineering we usually distinguish between the definition of qualities (global properties a specification should have), heuristics (guidelines for local decisions aiming at certain qualities), methodologies (sets of "best" practices usually recommending an order of steps in which to create a specification) and development processes (formal embodiments of methodologies including an

organizational framework).

In this chapter we provide the former three types of assistance for the creation of DMM specifications. We lay out qualities of a DMM specification in Sect. VII.1 and directly derive heuristics to achieve these qualities. As these heuristics are solely focused upon the achievement of a single quality (which usually impedes others), they must be applied in a balanced way. A methodology striving for this balance is presented in Sections VII.2 and VII.3, providing guidelines for formulating the semantic domain meta model and the rule set respectively.

Note, however that the methodology presented here is neither complete nor universal. It is based upon our experience with applying the DMM technique for a specific purpose and on specific examples (formalizing different parts of UML's semantics). The priorities underlying the design choices in Sections VII.2 and VII.3 reflect our intention to create primarily understandable specifications. For language semantics with different intentions (e.g., supporting efficient analysability) these guidelines need not be appropriate. We do furthermore not claim that we discovered all relevant qualities, heuristics, and guidelines for the formulation of UML's semantics, let alone that of other languages. But we firmly believe that the information given in this chapter forms a widely applicable set of such notions and that the assistance provided here can help new users of DMM to employ the formalism quickly in a profitable way.

VII.1 Qualities of DMM Specifications and Heuristics for their Achievement

The first step in guiding a Language Engineer to creating good DMM specifications is to actually define what we mean by the term "good", i.e., we need to define *qualities* of a DMM specification. The qualities we identify and discuss in this section are Correctness, Understandability, Modularity, and Efficiency. These are, in our experience, the most relevant qualities for actually formulating DMM specifications. Other quality notions (e.g., as explained for programs by Ghezzi et al. [GJM91]) may also be applicable to DMM specifications; they will emerge in the future application of DMM.

Two of these qualities (understandability and modularity) are already extensively discussed for DMM in general (in Section V.5). The technique of DMM, however, only *allows* for the achievement of these qualities. It is up to a concrete specification and its use of DMM's features whether it actually *is* understandable and modular. We also discuss correctness and efficiency of a given specification.

Each of the (global) qualities discussed here is complemented by a list of heuristics. These heuristics aim at providing support for local decisions to achieve the global quality. Conflicts occur at two levels: On the one hand the quality goals themselves are conflicting and need to be balanced. On the other hand even different heuristics striving for the same goal may formulate conflicting ways to achieve this goal. The methodology presented in the next sections attempts to procure these conflicts.

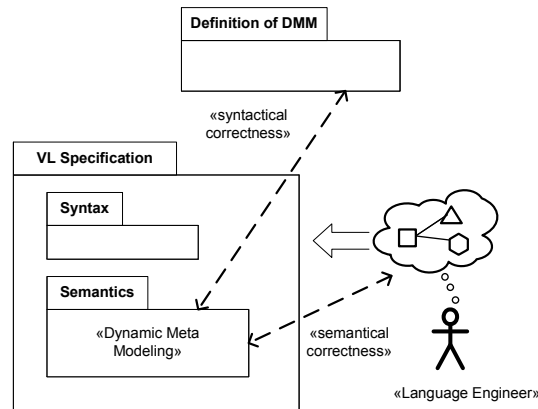


Figure VII.1: Illustration of the difference between syntactical and semantic correctness of a DMM specification

VII.1.1 Correctness of DMM Specifications

There are actually two notions of correctness of a specification, both of which are not subject to concrete heuristics: The first notion is that a specification should be correct in regard to the technique used (*syntactical correctness*), the second notion is that the specification expresses what it is supposed to express (*semantic correctness*)

Syntactic Correctness

Language definitions provide precise guidelines to decide syntactical correctness. For every expression in a language we can decide whether it meets the criteria set out by the definitions and can thus be considered as a *syntactically correct* expression. In Fig. VII.1 this situation is illustrated by the dashed arrow between the (formal) definition of the DMM technique and the specification under consideration.

For DMM these formal definitions are provided in the Chapters III to V. If, e.g., a specification contains rules which do not obey the restrictions necessary for behavioral conservatism (cf. Sect. V.4) then this specification is not a syntactically correct DMM specification. As the definitions provided there are precise and formal, we will not discuss this notion of correctness in the following any further but we will assume all specifications to be at least syntactically correct DMM specifications.

Semantic Correctness

The second notion of correctness is *semantic correctness*. It is comparable to the notion of *functional correctness* for programs:

A program is functionally correct if it behaves according to the specification of the functions it should it should provide (called functional requirements specifications) [GJM91], p.20.

Note that functional correctness is expressed in relation to a given specification. For a semantics definition, no such previous specification exists. In Fig. VII.1 this fact is illustrated by the thought cloud of the modeler which contains his imaginations of semantic concepts. Since this "reference" is intangible to all external verification, the only person able to judge whether a specification is semantically correct is the modeler himself.

For DMM specifications this situation means that a DMM specification can be considered to be semantically correct by the simple fact that the Language Engineer creating it (or the standardizing body publishing it) claims it to be correct.

Semantic correctness is the prime requirement for a specification and is not subject to compromises with other qualities. It simply does not make sense to consider highly efficient, but unfortunately incorrect specifications. For the following discussions we will assume that DMM specifications exist which are equal in their semantic correctness but which differ in regard to other criteria. Among such a choice we can then proceed to optimize with regard to other qualities.

VII.1.2 Understandability of DMM Specifications

Understandability (from a human reader's point of view) is a prime requirement for the whole DMM technique and care has been taken to enable the creation of understandable DMM specifications. It is, however, perfectly possible to provide DMM specifications which are highly unintelligible¹. To provide a high degree of understandability, the following heuristics should be applied:

- (U1) **Use clear terminology.** The actual mechanics of a DMM specification are in large parts based upon the terms defined in the semantic domain meta model. These terms should be as meaningful as possible.
- (U2) **Do not change replicated intensional elements.** Advanced users of a language can be assumed to know its (syntactic) meta model. Replicating these intensional elements in the semantic domain should not alter their definition. Thus users can rely on their existing knowledge in the interpretation of the semantic domain meta model.
- (U3) **Modularize according to meaning.** The conception of what should actually constitute a rule or which elements make up a package should be founded upon the intention and behavior of such elements.
- (U4) **Make information explicit.** A DMM specification is subject to manual inspection as much as to automatic interpretation. One should avoid to rely on implicit information when formulating DMM specifications.

¹Using confusing names is usually a good strategy to sabotage understandability.

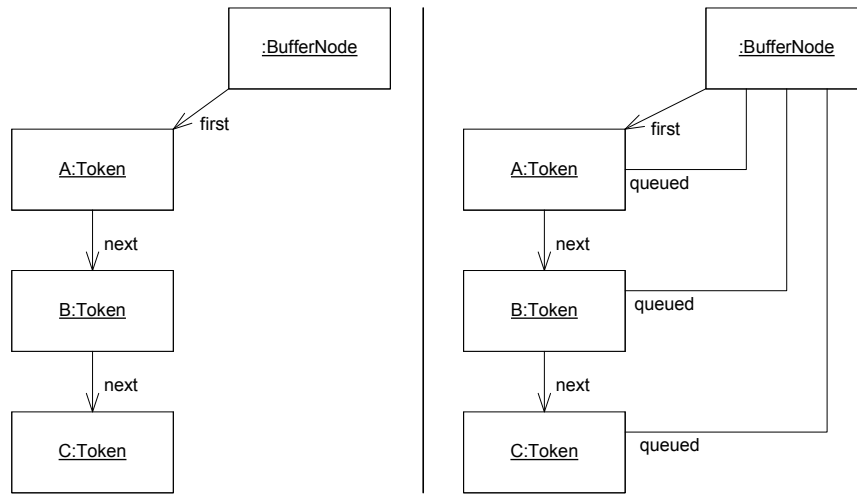


Figure VII.2: Different alternatives in encoding the queued property, implicit (left) and explicit (right)

As an example for such implicit information can be found in the queuing of tokens at buffer nodes. The fact that a token is queued in a node can be implicitly derived from the fact that the token is in an ordered list for which the buffer node forms the root element. For human users, however, the explicit introduction of a `queued` link between all queued tokens and the queuing buffer node is much more intuitive. Fig. VII.2 illustrates this difference with two examples.

- (U5) Avoid auxiliary constructions. Auxiliary constructions which do not serve any semantical but solely a technical purpose should be avoided.
- (U6) Keep rules small. A single rule should not be too complicated. Cognition theory indicates a number of 7 to 9 nodes to be the maximum of what can be easily perceived by a human reader.
- (U7) Keep rule sets small. A rule set should be as small as possible. While complex semantics certainly require large rule sets, excess should be avoided.
- (U8) Limit invocation hierarchies. Language readers can have trouble deriving the meaning of operations if their effects are spread over very deep or broad invocation trees. The graph manipulations implied by an operation then have to be assembled from all involved rules. Combining manipulations in single self-contained rules is preferable.
- (U9) Prefer UQS to iterations. The Universally Quantified Structures feature of DMM is a very concise way to specify sets of elements. From the viewpoint of understandability it is preferable over the mechanism of iterating over elements of a set.

VII.1.3 Modularity of DMM Specifications

At several places in this thesis we emphasize the need for modular and extensible semantics specifications. Again, DMM provides mechanisms which enable modular specifications, but the characteristics of a concrete rule set determine whether it is truly modular, maintainable, and extensible:

- (M1) **Highly modularize operations.** Each graph manipulation and meaningful condition should ideally be separated in its own operation. Examples are, e.g., dedicated destructor and creator operations which ensure that manipulation to the graph structure are always carried out in a consistent way. Changes to the underlying graph structure (by language extensions) can thus be dealt with by changing only the accessor rules of the respective elements. If used consequently, newly added rules only have to specify the additional behavior and can invoke existing operations for everything else.
- (M2) **Pass relevant information as parameters.** Information which is relevant to invoked operations should be passed as parameters. This reduces the reliance of invoked rules upon structures outside of their context and improves modularity.
- (M3) **Keep rules minimal.** Rules should only match the elements necessary for *their own* execution. The matching of additional elements (i.e., early preconditions checking) relies on implicit information about the construction of the invoked rules. Such implicit connections break the modularity and reduce maintainability.

VII.1.4 Efficiency

Efficiency is not an explicit requirement for the construction of the DMM technique. In general, we assume all activities relating to DMM specifications (i.e. validating test cases, proving semantic equivalence etc.) neither to occur very frequently nor to be very time-critical. These assumptions justify the use of graph transformations in our approach. Executing graph transformation rules is—in general—not very efficient.

In fact, matching rules on a graph (i.e., finding a subgraph isomorphism) is an NP-complete problem (problem GT48 in [GJ79]). In practical applications—like the use in DMM—most graph transformation rules expose benign properties: Rules are small in comparison to the underlying host graph, graphs are typed, applicable rules limited by control structures, occurrences are (partially) determined by already matched elements, and the node grades constrained. Under such circumstances, a single graph matching can be computed efficiently [Zün96, Dör95].

A second factor which influences the efficiency of the automatic interpretation of DMM specifications for a concrete model is the size of the ensuing LTS. For each state resulting from the application of a rule, an isomorphism check has to decide whether it is already contained in the known part of the LTS. This check is also computationally complex (while not proven to be NP-complete [GJ79]). The

number of already generated states and their size are the determining factors for this check.

Within the limits posed by the general use of graph transformations, a Language Engineer can strive for efficiency in a rule set by ensuring that a single rule matching can happen efficiently (heuristics (E1) through (E6)) and that the size of the ensuing LTS is kept minimal ((E6) to (E11) with (E6) effecting both goals).

- (E1) **Avoid large rules.** Each rule element must find a correspondent in the host graph. Smaller rule can thus be matched faster. Rules should be small in comparison to the underlying state graph.
- (E2) **Minimize rule sets.** The complexity of rule matching is proportional to the set of rules to check for possible matchings. Thus the number of rules should be reduced to a minimum.
- (E3) **Minimize state graphs.** In formulating the semantic domain meta model, the structures of the state graph are determined. When creating this meta model one should aim for a minimal set of classes. Information which can be encoded in attributes should not be exposed as separate classes. Redundancy should be avoided.
- (E4) **Avoid long paths.** If nodes in a rule are connected via long paths (maybe even arbitrarily long ones via the use of preconditions) the matching of the rule will become extremely slow. If possible, the introduction of direct connections ("shortcuts") abbreviating these paths helps to facilitate an efficient match.
- (E5) **Create large interfaces.** The more elements are passed as parameters in a rule invocation, the easier the matching of the invoked rule becomes since many of its elements can only find a single pre-determined match.
- (E6) **Avoid unconnected rule nodes.** A high connectivity between nodes of a rule allows for a more efficient matching as candidate sets for unmatched rule nodes can be quickly determined. Unconnected nodes can be matched by every node of the correct type causing broad branching in the LTS.
- (E7) **Avoid invocations.** Each rule invocation creates a new intermediate state in the transition system and requires an additional rule matching. A high number of self-contained rules or relatively small invocation trees thus help to make the interpretation more efficient.
- (E8) **Use UQS.** For matching or manipulating sets of elements the mechanism of the UQS should be applied whenever possible as it results in the application of a single rule only, while an iteration usually comprises the application of (at least) three rules, thus enlarging the LTS.
- (E9) **Specialize rule nodes.** Nodes in rules should always have the most special subtype applicable. This reduces the chances of (possibly erroneous) matches with unintended nodes. The worst case in terms of efficiency is an anonymous node in a rule as every host graph node can fill this role. The LTS expands dramatically upon applying rules with anonymous (and possibly unconnected, see above) nodes.

- (E10) **Generalize state graph nodes.** Nodes created in the state graph should be as general as possible. Especially the assignment of names for auxiliary elements should be avoided as anonymous elements can be identified during isomorphism checks, thus reducing the size of the ensuing LTS significantly.
- (E11) **Avoid failing invocations.** Big-step rules should (as far as possible) check that the preconditions for their successful execution (including called rules) are present. This avoids the unrolling of long derivations which are ultimately unsuccessful.

It is obvious that these heuristics are highly contradictory. Following, e.g., the advice of Heuristic (E8) and using UQS for the processing of sets will usually result in larger rule sets, thereby violating (E2) as the UQS is unrolled to plain graph transformations. Similarly, (E1) and (E2) cannot be both fulfilled optimally as a splitting of large and complex rules will increase the size of the rule set. The methodology provided in the next sections will place these heuristics in a pragmatic context.

VII.2 Guidelines for Formulating the Semantic Domain Meta Model and Relations

The purpose of the semantic domain meta model is to explicitly introduce semantic concepts and to form the basis for the formulation of DMM rules (cf. Section V.1). The process of its conception and the creation of the necessary semantic relations can be divided into five separate steps:

- ◆ The conception of extensional entities and their structure
- ◆ The replication of intensional elements and their modification
- ◆ The introduction of auxiliary elements and structures
- ◆ The relation of syntactic and semantic domain
- ◆ The definition of packages

The following subsections provide details on each of these steps. As a running example we provide an extract of the case study of UML Activity Diagrams.

VII.2.1 Conception of Extensional Entities

The first step in creating a suitable semantic domain is the explicit formulation of semantic concepts. It has to be clarified which concepts are considered to be part of an interpretation state and do thus constitute extensional entities. Classes in the semantic domain meta model represent these entities. Associations between the classes express the relations between the different entities. Attributes and multiplicities can be used to further refine the structures of the meta model.

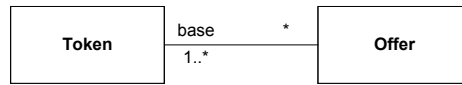


Figure VII.3: Extensional elements in the semantic domain meta model

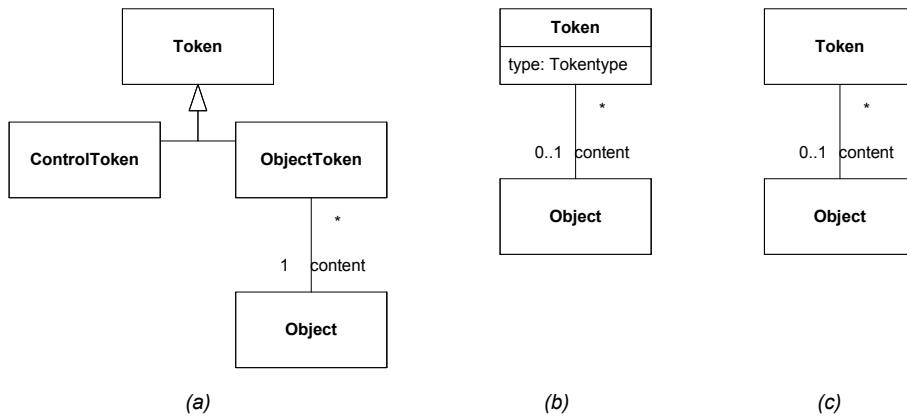


Figure VII.4: Three different alternatives to capturing different token types in a semantic domain meta model

The discussion in Sect. VI.1 demonstrates this process. It is aimed at clarifying the role of the term "offer" for the semantic domain meta model for Activity Diagrams. The conclusion is that the term offer as used in the UML semantics description constitutes an entity and is thus part of the semantic domain meta model. Figure VII.3 illustrates that our semantic domain meta model contains the classes `Token` and `Offer` and the detailed association between them.

Regarding the level of detail present in the definition of the extensional elements, one has to bear in mind that elaborate meta models allow for elaborate state graphs. On the one hand this increases the possibilities for a high number of big-step rules in the operational rule set as much information can be encoded in the state graph. On the other hand, elaborate state graphs impede efficiency of rule matching. The aim is thus to strive for conciseness here, without sacrificing readability (i.e., weighing (U8) against (E4))

Another design decision is whether to formalize necessary information as separate classes, as attributes, or by association. Different requirements from understandability and efficiency need to be respected in this decision.

Fig. VII.4 illustrates such a design problem for the case of tokens. To distinguish between tokens representing an object and tokens representing control one may either (a) create separate classes (and a generalization to the common concept), (b) encode the information in an attribute, or (c) rely on the fact that the existence of a `content` object is sufficient to mark object valued tokens. The latter solution complicates the formulation of rules as they always have to check for the presence or absence of the `content` element. The solution (b) allows for

a local check on the type of the token by introducing an attribute but conceals the token differences rather than revealing them (violating (U4)). Additional constraints have to ensure that the attribute and the attached object always remain consistent. Solution (a) involving separate classes is to be preferred in this case. While it is the most verbose it also serves best to exemplify the differences between the two types of tokens (U1). Rules can either be formulated for the superclass `Token` (for common behavior) or for the special behaviors of the subclasses. As the UML specification also explicitly coins the two terms control and object token (U2), this solution is adequate.

VII.2.2 The Replication of Intensional Elements and their Modification

In Sect. V.1 we argue for the replication of intensional elements in the semantic domain meta model. The following considerations guide this replication process.

Omission of Purely Syntactic Elements

The primary purpose of the syntax meta model of a language is to describe the possible constructions of its syntactic expressions. There may be elements in the (syntactic) meta model which do not directly express semantics or which do not express semantics in a formalizable way.

In the case of UML Activity Diagrams, the element `ActivityGroup` is such a purely syntactic element. An `ActivityGroup` allows for grouping elements together. No specific semantics is denoted by it, it is solely used for presentation purposes. `ActivityGroup` is thus not replicated to the semantic domain meta model for UML Activity Diagrams.

A distinction which is relevant from a syntactic but not from a semantic point of view is that of `ObjectFlow` and `ControlFlow`. As discussed in Subject. VI.2.2, these elements can be unified to the semantic concept `Edge` (cf. the example in Fig. VII.5).

Modifications to Replicated Intensional Elements

While (U2) discourages modifications to the replicated intensional elements, such changes are sometimes useful to allow for more uniform processing or to account for distinctions between syntactic elements and their semantic interpretations. An example here is the extension of the concept of pins to cover control-valued in- and outputs for actions. The problem stems from the fact that UML uses pins as a syntactic element to distinguish between object and control in- and outputs of an action. Semantically, this raises the question at which element offers for control tokens are supposed to buffer until acceptance by the action. Introducing pins for control tokens as well as for object tokens allows for iterating over all in- and outputs of an Action uniformly. Thus we accept a modification of the replicated intensional structures to allow for easier formulation of the operational semantics part.

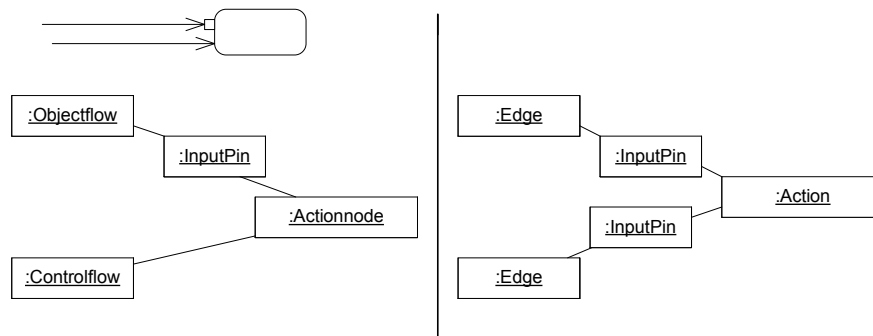


Figure VII.5: Modifying the Actionnode element in the semantic domain (displayed on the right hand side)

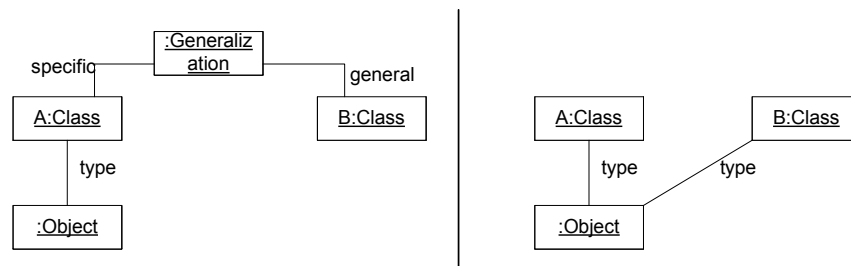


Figure VII.6: Illustration of the type flattening

The element Action also presents a modification in that its syntactic counterpart is called ActionNode. While this name is suitable from a syntactic point of view (Actions form nodes in an activity graph), the removal of the directly connected edges and the semantics of actions, which is different from all other nodes, makes it unsuitable for the semantic domain meta model. Thus Actions are neither in character nor in name nodes anymore. An illustrating example for this modification can be found in Fig. VII.5.

Introductions of Shortcuts

For efficiency (E4) as well as for understandability (U6) reasons the navigation along long paths in rules is undesirable. In formulation of the semantic domain meta model, one is able to introduce shortcuts which enable more direct navigation. An example for the use of this technique is the "flattening out" of type and generalizations relations in the semantic domain meta model for UML Activity Diagrams. Here, each object is connected to each of its types by a direct association rather than by a direct type association and a path of generalizations (as specified by the UML meta model). Fig. VII.6 illustrates this modification by displaying a sample instance of the UML meta model on the left hand side and its counterpart (an instance of the semantic domain meta model) on the right hand side.

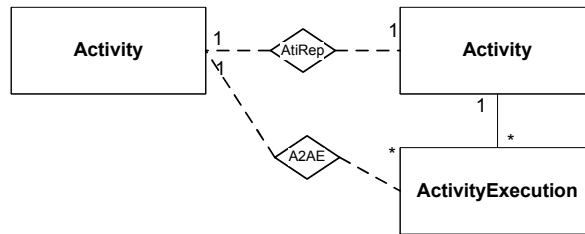


Figure VII.7: Example for the relation of intensional and extensional classes

VII.2.3 Introduction of Auxiliary Elements and Structures

To facilitate an easier formulation of rules, auxiliary elements may be introduced to the semantic domain meta model. These additional elements should be used sparsely, though. Since they do not really comprise semantic information they may confuse readers (U5). They furthermore extend the state space and may thus impair efficiency (see (E3)).

An almost indispensable auxiliary construct is the linked list. Linked lists allow to impose an explicit and graphically specified order on sets of elements. This order then allows for an iteration over the set. An example for the application of the linked list pattern can be found in Fig. VII.2, where tokens are queued in an ordered list at a buffernode.

VII.2.4 Relation of Syntactic and Semantic Domain

The formulation of Relations connecting the elements of the syntactic and semantic domain meta model is closely connected to the introduction of elements as described above.

Relations to Pure Extensional Elements

Extensional elements usually serve to make the semantics of some specification element tangible in the semantic domain. `ActivityExecution`, e.g., is used to denote the execution of activities, thus Meta Relations between the UML construct `Activity` and the extensional element `ActivityExecution` are introduced (compare Fig. VII.7). Such Relations to purely extensional elements usually carry the cardinalities of 1 at the syntactic and *(unbounded) at the semantic end.

Relations to Replicated Intensional Elements

The replication process for intensional elements must be documented and constrained by the introduction of Relations. Usually, a 1:1 Relation connects the original intensional element with its replicated extensional counterpart. Care must be taken if modifications have been made to these elements: additional

constraints to the Relation might be required to ensure that information is replicated in a suitable way. Especially in the case of introducing shortcuts for paths, the features of OCL for handling (recursively defined) sets come in handy.

Relations to Auxiliary Elements

Auxiliary elements are usually not targeted by Relations as they are of a purely technical nature and have no counterparts in the syntactic domain meta model. Quite frequently, however, they influence other mappings, e.g., when introducing orders to sets.

An overall concern when defining Relations is the consistency of the constraints formulated for the syntactic domain meta model (e.g., its multiplicities), the semantic domain meta model, and the Relations. For replicated structures, this consistency is easy to achieve but other elements must be carefully checked whether the given constraints are not contradictory. As a general rule the constraints of the semantic domain meta model should at least be as restrictive as the Relations targeting the semantic domain. In the example the association between the (extensional) classes `Activity` and `ActivityExecution` thus carries the same cardinalities as the Relation `A2AE`.

VII.2.5 The Definition of Packages

Packages (cf. Sect. V.4) are DMM's way to achieve modular specifications. Finding a good package structure allows for separating a complex specification into several manageable parts. A good package structure can be characterized by the properties of *high internal cohesion* and *low external coupling*.

Starting points for a package structure can be formed by considering which elements have similar tasks (e.g., separating buffer and control nodes), which elements interact frequently (behavior definitions and their executions) or which are used in combination.

VII.2.6 Discussion

The methodology described in the previous subsections cannot be seen as a single, one-pass process. It is rather a guideline along which an iterative development of the semantic domain meta model can be structured. Especially the construction of the rule sets (see next section) will probably induce changes in the meta model as rules may call for the introduction of additional auxiliary elements or a restructuring of packages (see Subsection VII.3.4). A major change is that up to now no operations have been added to the classes. These operations (and their detailed interfaces) depend heavily on the detailed structure of the rule set.

VII.3 Guidelines for Formulating DMM Rule Sets

The methodology for formulating DMM rule sets is organized in four steps

- ◆ Partition behavior into big-step rules
- ◆ Distribute behavior on small-step and premise rules
- ◆ Formulate single rules
- ◆ Align rules with package structure

The following subsections provide details on these steps.

VII.3.1 Partitioning of Behavior into Big-Step Rules

The first step in the creation of a DMM rule set is the partitioning of behavior into big-step rules. The following considerations underlie this separation.

Express Independent Behavior

Big-step rules provide units of synchronized behavior. They should thus express a unit of behavior which is ideally self-contained and concerns only a single aspect. While these ideals strive for understandability (U3), sometimes synchronization requirements enforce the combination of rather large blocks of behavior under a single big-step rule (e.g., the starting of an action in UML Activity Diagrams).

Enabling Concurrency by Big-Step rules

In states without open invocations, all big-step rules compete for the next match. If two behavior steps are thus to be carried out concurrently (albeit not synchronous) to each other, they should be specified in separate big-step rules. The interpretation mechanism will then compute all possible application orders, i.e., an interleaving. The complexity of manipulations effected by the big-step rules (and the rules they invoke) determines the granularity of the interleaving.

In the DMM rule set for UML Activity Diagrams, the flowing of offers over the different elements is separated into big-step rules. Thus any of the existing offers can be advanced in a big-step and the LTS will contain all possible interleavings of the different offer movements.

Preparing for Change

Injecting new big-step rules is the easiest way to extend the behavior of DMM specifications. Creating numerous smaller big-step rules (M1) provides more possibilities for later additions. This does, however, come at the price of efficiency losses (E2).

VII.3.2 Distribution of Behavior by Using Small-Step and Premise Rules

Following the initial conception of which parts of a behavior should be synchronized in a big-step rule, the Language Engineer needs to decide how to distribute the manipulation over different rules. The following situations indicate that the invocation of a separate rule is advisable.

Rules become too complex

The prime motivation for employing the invocation mechanism is when the rule at hand becomes too complex. (U6) suggests 7 to 9 elements as the maximum number of nodes. If rules require more elements than that, splitting the behavior into two rules connected by an invocation should urgently be considered.

Distributing Responsibilities

In keeping with the Object-oriented alignment of the DMM approach one motivation for splitting a rule is to distribute the responsibility for behavioral steps between different elements. Instead of a large complex rule with a central control, different nodes can carry out certain local manipulations in their context with the invoking rule only coordinating these behaviors.

Execution order

Invocations allow for the ordering of executions. If a complex manipulation requires a certain order of steps to be taken in its application, a rule can express this sequence by formulating these different steps in separate operations and calling them in a specific order. A good example for such a rule is `activityfinal.accept` which first accepts a token and then proceeds to destroy the activity.

Rule exists

If for a particular manipulation a rule/operation has already be formulated, all other rules should refrain from specifying the manipulation themselves and invoke the responsible rule. Especially in the case of constructors/destructors, only their consequent use ensures that the promised gains in maintainability can be realized.

Rule may be reused

If the modeler suspects that a particular behavior can be used at different places in the rule set, it should be separated into its own operation (following (M1)). The offsets in terms of efficiency (E7) have to be accepted here.

Branching

Whenever a behavioral step varies depending on the existence/absence of a certain condition it should be placed in a separate operation which is then detailed with a number of rules, each implementing one specific case of this operation.

Handling sets of elements

Manipulating sets of elements can be handled in two ways: Either a UQS is used and an invocation is posed to the UQS (which then results in the respective behavior being carried out by all elements matched by the UQS) or an iteration is created by a set of rules. Both mechanisms require the invocation of another rule to actually carry out the manipulations.

Using UQS has advantages in terms of understandability (U9) and efficiency (E8), albeit the latter depend on the native support of the interpreting mechanism for UQS². In GROOVE (cf. Chapter VIII) no such support is provided, thus the advantages are largely offset by the increased size of the rule set (as a result of the unfolding mechanism). Without native support for UQS there are also necessary limitations to the maximum number of elements matched by a UQS. Should a host graph exceed this number, the rule set will not specify the correct behavior.

In the AD rule set, e.g., the rule `activityfinal.accept` (Fig. B.37) makes use of a UQS to terminate all running behaviors of an activity.

The alternative to using UQS is the explicit formulation of a (usually recursive) iteration. This typically results in three rules: A first rule identifies the first element of the set and invokes a processing operation on it. This processing operation itself is implemented by two rules: one for processing an element and invoking the processing of the next element and one to end the recursion upon encountering the last element of the set. Variations of this general pattern can occur. We recommend specifying explicit orders (see Subsect. VII.2.3) to guarantee termination of the recursions.

As a general style we furthermore recommend following a naming pattern: The rule for the identification of the first element should be named similar to the processing operation, with the distinction of being formulated in plural. For instance the rule `actionexecution.collectInputs` triggers the recursive application of the rule `actionexecution.collectInput` on all inputs of an executing activity (cf. Figs. B.53 and B.54).

VII.3.3 Formulating a single rule

The formulation of a single rule gives rise to considerations on the formulation of its interface, its preconditions, and the manipulations actually carried out by the rule's body.

²To our knowledge no GT tools exist which natively support our notion of UQS.

Construction of a Rule's Interface

A rule's signature or interface (and its corresponding operation's interface) determines the context object of the rule/operation and the parameters passed at invocation time.

The context element, i.e., the node on which the rule is invoked, should a) have direct connection to most if not all elements of the rule, b) determine most of the other nodes in the rule by "to one" associations and c) be defined in the same package as all elements which are being created or deleted. If a Language Engineer consequently employs this style of specification, the rule set becomes easily maintainable as all manipulations to elements of a package happen in operations belonging to classes of this package. Efficient rule matching is also supported as this style follows (E4) and (E6).

Note further that it is a general convention that the name of the context node should reflect the class of the context node. This allows for a much easier navigation of the rule set. For complicated class names, abbreviates may be (consistently) used. For instance the context nodes of the operations of `CallBehaviorActionExecution` are called CBAE uniformly in the case study's rule set.

Formulation of a Rule's Preconditions

The formulation of a rule's preconditions determine in which situations it is applicable. Care must be taken to precisely determine the type of situation in which the rule may be safely applied. If multiple rules implement the same operation they must usually not be in conflict. Two rules are said to be in conflict if they may match (partially) on the same elements of the host graph. To avoid such a conflict the rules need to apply either to disjoint scenarios (e.g., the two rules implementing `offer.getAccepted` (cf. Fig. B.16) differ in applying to offers of object and control tokens respectively) or they depend on the presence/absence of certain structures. In the latter case the absence must be formalized by using a NAC (e.g., the rules for `join.collectOffer` (cf. Fig. B.47) differ in the presence or absence of a `next` link to another input pin).

If the rule to be formulated is a small-step or premise rule, several elements (at least the context node) are pre-determined since they are passed as parameters. Note that all of these elements need to appear in the rule's body. All other elements of the rule can easily be matched if they are reachable from already given elements by associations identifying the next element uniquely (E5). In the ideal case such a matching is possible in direct connection to the elements pre-determined by the interface (E6). If such direct and unique connections are not possible, one should think about introducing shortcuts to the meta model to allow for such direct connections (E4). Especially if long paths need to be followed in a rule, one should take this optimization criterion into consideration.

Early Precondition Checking A big issue for the formulation of rule preconditions is the question of *early precondition checking*: Especially big-step rules initiate the execution of complex behavioral steps which most likely incorporate

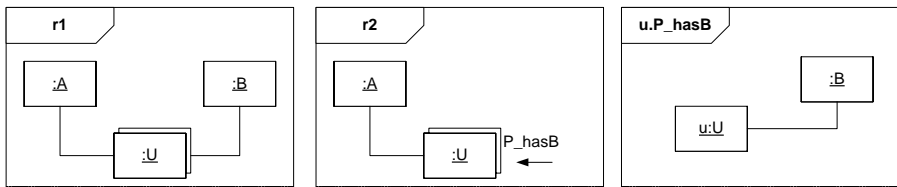


Figure VII.8: Example for the different usages of the UQS construct

the invocation of other rules. The question is now to which extent should an invoking rule ensure that the preconditions for the application of all (transitively) invoked rules apply?

The one extreme is that rules only ensure that the preconditions for their own manipulations apply. The drawbacks here are that human readers are left somewhat guessing on the "real" applicability of the rule as the probability for later failures is rather high (violating (U4)). In the creation of the LTS, numerous attempted big-step applications will ensue, which ultimately fail (strongly violating (E11)). Thus the effects of this alternative are most adverse to understandability and efficiency.

The other extreme is that rules try to ensure that all their invoked rules can apply. Beside the point that this is not technically feasible in the general case (as invoked rules may contain recursive loops etc.), it may also swell rules out of proportion. Modularity of the rule set is in this case non-existent as rules require detailed knowledge about all other rules which may be involved in their invocations (severely violating (M3)).

A reasonable compromise is that big-step rules incorporate all preconditions which either aid understanding or which prevent frequently occurring application failures. As an example from the Activity Diagrams case study take the rules for `action.start` in Fig. B.52. The rule checks whether all of its inputs have acquired offers. This is a necessary precondition to understand when an action executes and it prevents actions trying to start prematurely. Small-step rules should in general not perform early precondition checking.

UQS in preconditions Using universally quantified elements in a rule's precondition gives rise to two additional considerations:

A basic difference in using UQS is whether information appended to the UQS should serve as a *characterization* of the set expressed by the UQS or whether it formulates a *predicate* over this set³. Figure VII.8 illustrates this difference.

In the rule `r1` of the figure, a UQS (`U`) is connected to two existentially quantified nodes (`A` and `B`). This rule will match on the hostgraph if its core rule matches (i.e., if there are nodes of type `A` and `B`). Additionally, it will match all nodes of type `U` which have a connection to both nodes matched by the core rule. The set of matched `Us` is thus *characterized* by the attached elements `A` and `B`. Each

³Technically, characterization is also a predicate, but one *defining* the set.

further element attached to the UQS will strengthen the characterization of the set, possibly reducing the number of actual elements matched by the UQS.

The rule in the middle of Fig. VII.8, r2, expresses something else. Here, the core rule only matches a single node of type A and the elements matched by the UQS are all Us connected to A. There is also a premise rule P_hasB attached to the UQS. This premise rule has to hold for all nodes matched by U. Thus this premise rule is not an additional characterization of the set expressed by the UQS but a *predicate over* this set. Rule r2 now expresses that it will match an A provided all Us attached to this A also have a B attached. Rules r1 and r2 thus have very different meanings.

A second issue arising from the formalization of UQS in Section IV.4 is that UQS imply an existential match. The left hand side rule in Fig. VII.8 will thus only match on an A and a B if *at least* one connecting U exists. The effect of this definition is that all structures visible in a rule will (at least) be present once in the rule's matching. Universal Quantification in the literature does usually include the empty match, though. Language Engineers with a background in Graph Transformations should be aware of this difference. A practical effect is that the complete absence of the UQS (if that is a situation of interest) must be explicitly treated by a separate rule.

Formulation of a Rule's Manipulations

The graph manipulations in a DMM rule are expressed by marking elements with the constraints {new} or {destroyed}. When formulating such manipulations the modeler needs to ascertain that no consistency constraints toward the rest of the specification are violated by them.

A very simple and general rule is that elements of the host graph which represent specification information, i.e., elements whose types are replicated syntax elements in the semantic domain meta model (cf. Subsect. VII.2.2), are *never* subject to manipulations. Creating and deleting such model elements would constitute an evolution of the original specification (cf. [GKP98]) which is not supported by DMM. The maximum amount of modification such elements may experience is the creation and destructions of links leading to nodes representing purely extensional elements.

Another type of consistency is that multiplicity constraints in the semantic domain meta model may forbid or require the existence of certain structures. Looking at Fig. VII.3, we can, e.g., note that an offer may not exist without a base token it represents. Thus no rule may create offers without simultaneously creating this link. Conversely, the destruction of a token "orphans" its offers. Explicit constructor and destructor operations help to consistently build and destroy such structures. In the Activity Diagram rule set, the operation token.destroy (cf. Fig. B.13) initiates the destruction of all emitted tokens when destroying the token itself.

Finally, the manipulations of a rule may violate constraints posed by the mapping between the underlying specification model and the state graph. This kind of consistency violation is as rare as it is hard to detect. Usually the interdiction

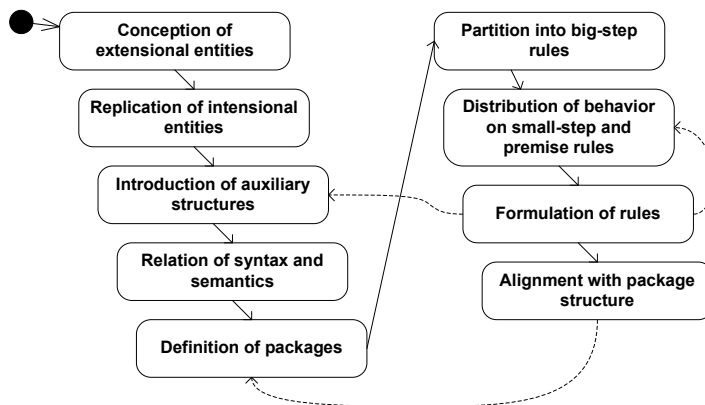


Figure VII.9: Overview of the steps in the methodology for the creation of DMM specifications

of manipulations to replicated intensional elements suffices to prevent problems in regard to the semantic relations.

VII.3.4 Alignment of Rules and Packages

A final step in constructing a DMM rule set is the alignment of package structures defined for the meta model with the rule set. Each rule is assigned to a package; the most likely package is the one which also defines their context element. This assignment implies that all elements used in the rules must a) be accessible in the package the rule is defined in and that b) at least one left hand side element is an original element of the package. The latter condition is problematic only for rules not defined together with their context elements (and will thus mainly apply to a posteriori language extensions). Condition a), however, provides new insights to the formulation of the package structure. If rules violate this condition, either the rules need reformulation or the package structure conceived upon the static information only does not correctly reflect the implicit connections of the semantic domain. The alignment of rules and packages may thus well result in a redesign of the packages.

VII.4 Summary and Discussion

The methodology for creating DMM specifications is summarized in Fig. VII.9. The basic structure of the steps follows the order of presentation in the preceding sections with the dashed arrows indicating possible iterations over previously conceived constructs.

From a purely technical and scientific point of view, this chapter does not add much (if any) information to the technique of DMM. It does, however, lend weight and justification to the term Language Engineer which we use throughout this thesis. The creation of a new language specification is in fact an undertaking

similar to programming. The fundamental concept of Software Engineering is that such creations need to be performed systematically, i.e., engineered.

The contribution of this chapter is to provide the fundamentals of such a systematical approach for the formulation of semantic specifications. Future work can complement the results presented here in several directions. Additional qualities, heuristics and best practices guidelines may be discovered in applying DMM to further languages. Tools may provide further assistance, e.g., in supplying automated checks to guarantee a rule set's consistency to the formulated meta model. Other engineering techniques like systematic testing can also find application in this area. Finally, a formal process can be devised which integrates all of these ideas in an organizational framework.

All of these contributions to the field of what may then be called *Language Engineering*⁴ need to be bred from broad experience. In this chapter we supplied readers with the necessary knowledge to actually start applying DMM and thus to add to the (hopefully growing) experience in using DMM. The next chapter will continue along this line in allowing Language Engineers as well as Language Users to actually experience the effect of a DMM specification.

⁴The term Language Engineering was coined by Reiko Heckel for a Dagstuhl Seminar [BH05] in which (amongst other techniques) DMM was presented.

Chapter VIII

Automatically Applying DMM Specifications

Given a language specification with DMM, every model of the language can—in principle—be precisely and exhaustively interpreted by a human user of this language. The formal foundations in Chapters III to V allow for an unambiguous construction of the LTS originating in the model's start state. In practice, however, this kind of manual interpretation is limited to exemplary situations only. For an exhaustive rule application on even minimal examples automation is essential. We call the tool to carry out this task a *DMM interpreter*. A DMM interpreter is a generic tool which takes a start graph and a DMM rule set as its inputs and produces the complete LTS defined by these inputs¹. Note that the DMM interpreter does not add any information to an existing DMM specification and neither does it define the semantics in the way that reference implementations of programming language compilers do. It simply automates and speeds up a process which is solely based on the previously presented DMM specifications.

Since DMM specifications are internally based on Graph Transformations we can rely on existing technologies for the prototypical realization of a DMM interpreter. Unrolling an LTS from a given rule-based specification is the purpose of model checking tools. We discuss the state of the art in model checking of Graph Transformation systems in Sect. VIII.1. The result of this survey is our decision to employ the GROOVE tool set for the prototypical realization of the DMM interpreter. We introduce the GROOVE tool set and its input format in Section VIII.2. As this format supports less control constructs than DMM, a systematic translation of the provided DMM rules is necessary (see Sect. VIII.3 for details of this translation). Achievements and shortcomings of the translation of our case study to GROOVE are presented in Sect. VIII.4 and consequences for future work toward a dedicated DMM interpreter are discussed in Sect. VIII.5.

¹We assume the LTS to be finite

VIII.1 Model Checking approaches for Graph Transformation Systems

Graph Transformation rules are a very powerful formalism. Their concise and visual descriptions are leveraged by non-determinism in both selecting and applying a rule. Thus, even very small rule sets can create rapidly expanding transition systems. For a long time research was thus focused on investigating properties of the rule sets directly (e.g., the notion of conflicting rules as checked by the AGG tool set [Bey93]). In practical applications of Graph Transformations, the non-determinism is usually heavily restricted to yield deterministic, provably terminating, and confluent transition systems (e.g., for code generation in Fujaba or model transformations in the Consistency Workbench [Küs04]).

For the description of semantics, however, this heavy restriction of non-determinism is not appropriate. Not only do semantics often contain intentional non-determinism, but concurrency can also be addressed by applying rules non-deterministically (i.e., expressing concurrency by interleaving). And while properties of the rule set might help to validate properties of the semantics definition, we are ultimately interested in the semantics of a single model, i.e., the transition systems originating in the start states of this model. Only completely producing these transition systems yields the semantics of a model.

Advances in computing power as well as successful research in the field of *model checking* during the last years opened up new possibilities in actually computing and handling transition systems with a large state space. In two different approaches, researchers tried to apply these techniques to graph transformation rules: The CheckVML approach by Dániel Varró, and the GROOVE (GRaphical Object-Oriented VERification) tool set by Arend Rensink. Both approaches face the same task: starting from a Graph Transformation System (i.e., a set of Graph Transformation rules and a start graph) they produce the set of all possible derivation sequences (i.e., the Graph Transition System) originating in the start graph. They approach this problem in very different ways (cf. Fig. VIII.1).

The *CheckVML* approach [SV03, Var03] is basically a denotational approach in which the whole input transformation system is converted into the input specification of a model checker. This translation already curbs some of the inherent complexity of Graph Transformations as the number of elements which might possibly be created when applying the rule set must be limited a priori. The resulting specification is then processed by a dedicated model checker (SAL) which produces an LTS and checks properties on the LTS. Advantages of the approach (cf. [RSV04]) are that the employed translation and model checking framework is highly optimized for efficiently generating such state spaces (cf. Section 5.3 of [Var04]) and that properties can be expressed using the full power of temporal logic. A drawback is that the basic result of the model checker is a single value indicating success or failure of the checked property (plus counterexamples in the case of failure). The visualization and exploration of the complete LTS is not supported.

The GROOVE tool set in contrast works directly on a set of graph transformations. An editor is supplied which allows for editing graphs as well as graph transformation rules, using a visual format similar to ours or Fujaba's (i.e.,

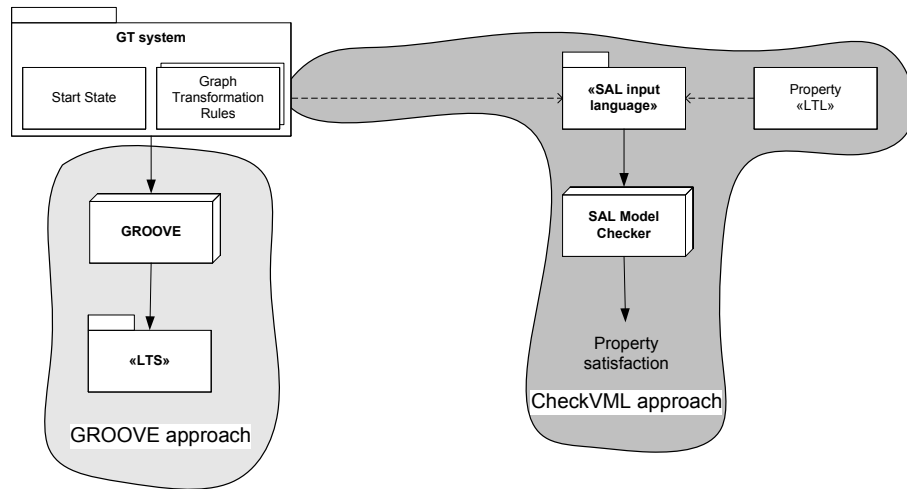


Figure VIII.1: Illustration of the differences between the CheckVML and the GROOVE approach to model-checking Graph Transformation systems.

rule sides are combined into a single graph with element labels providing the necessary distinctions). Based upon a rule set and a start graph, GROOVE can then produce and visualize the complete LTS originating in the start state. GROOVE can also perform basic model checking by allowing the state space generation to be bounded by rules specifying required or unwanted properties (i.e., reachability and safety checks). Complex and temporal properties cannot be formulated in GROOVE. The advantages of GROOVE are that it allows for unlimited element creation and that it visualizes the resulting LTS.

In a comparison of the two tools [RSV04], CheckVML displays clear advantages in terms of generation time for relatively static systems (i.e., systems in which element creation/deletion occurs only rarely) while GROOVE is able to process dynamic systems much better (in fact, CheckVML completely failed to process some of the more dynamic examples). Since we are interested only in a prototype, the lower efficiency of GROOVE is not a real drawback while the visualization of the complete LTS is a very strong advantage. We thus choose GROOVE as the basis for a prototype DMM interpreter.

VIII.2 Introduction to the GROOVE Tool Set

The GROOVE tool set currently comprises four separate tools which are briefly introduced in the next subsections. The whole project is developed by Arend Rensink and Harmen Kastenberg and information about it is available in several papers [Ren04a, Ren04c, Ren03b, Ren03a] and the website <http://groove.sourceforge.net/groove-index.html>. The toolset is implemented in Java, is placed under the GNU Public License (GPL), and is open source.

Of particular interest for us is the concrete Graph Transformation mechanism

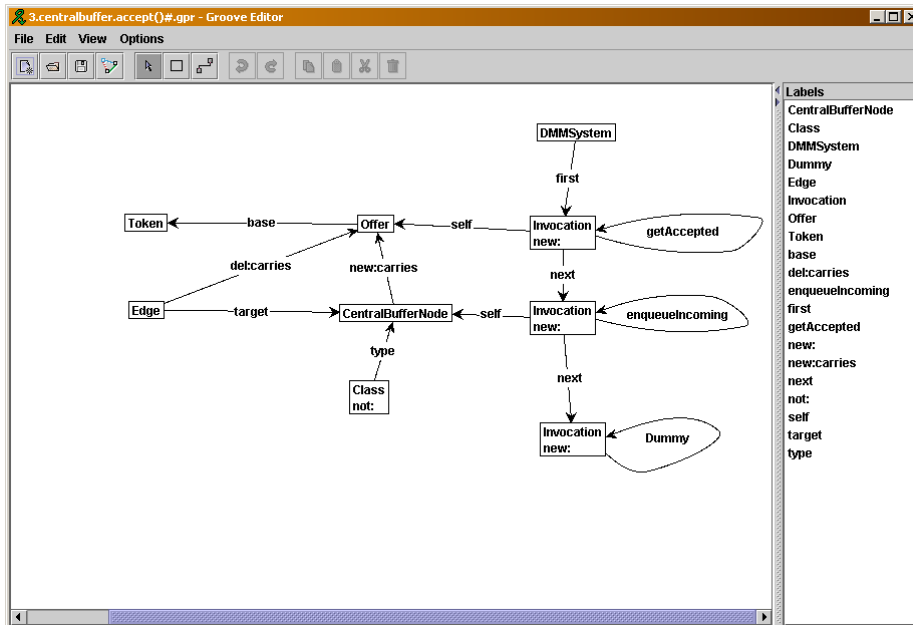


Figure VIII.2: The GROOVE Editor tool

which GROOVE employs. As the discussions in Chapter IV show, a lot of technical details can distinguish two Graph Transformation approaches. We will carefully examine the GROOVE mechanisms to find out which features of DMM rules are natively supported and which require a translation (see Sect. VIII.3).

VIII.2.1 GROOVE Editor

The GROOVE editor (a screenshot of which is displayed in Fig. VIII.2) allows for the editing of graphs and graph rules. For graph rules, special labels `new:`, `del:`, `not:` are used to signify that an element should be created, deleted, or is part of a negative application condition. The editor also supports a rule view, in which the different groups of elements are distinguished by using different colors and line styles. Newly created elements are displayed with green solid lines, elements to be deleted are displayed with blue dashed lines, and elements of NACs are displayed with red dotted lines. We can thus use the groove editor to create either start states (file format `.gst`) or rules (file format `.gpr`).

VIII.2.2 GROOVE Imager

The GROOVE imager is an auxiliary component which allows for the visual display and layouting of graphs and graph rules. It is mainly used in the editor and simulator components although it can also be used as a stand-alone tool. The layouting algorithms currently supported are spring layout (for displaying state graphs) and forest layout (for displaying LTS graphs).

VIII.2.3 GROOVE Generator

The GROOVE generator is the component responsible for generating full or partial transition systems. It is used by the other components and only allows for command line access. The algorithmic solutions for efficiently performing the graph matchings, performing checks on isomorphism of state graphs (using graph certificates), and storing the graphs are described in [Ren04c].

VIII.2.4 GROOVE Simulator

The GROOVE simulator is the main tool of the GROOVE tool set and allows for the unrolling of an LTS from a given graph transformation system. A screenshot of the simulator component is displayed in Fig. VIII.3. One can either explore the LTS interactively in a stepwise fashion or request a full generation. Additionally, the simulator offers the possibility to select a rule from the underlying grammar as the bounding rule. If a branch in the transition system reaches a possible application of this rule, it is not generated any further. Similarly, the non-applicability of the rule can be used to limit the generation of the transition system. Using such bounding rules allows for reachability (is a certain situation reachable in a derivation sequence) and safety analysis (is a certain property never violated in the system).

The simulator component also allows for the export of a created LTS. The export format is GXL, an XML dialect for the description of graphs [Win01]. If further analysis of the LTS is required, this exported file can serve as the basis for additional tools.

The Simulator has three different views. In the *state view* (Fig. VIII.3) a graph state is visible. The elements of this graph are automatically layouted (by the imager tool) but can be moved by the user. Convenient operations to emphasize certain elements and hide others allow for a purposeful exploration of the state graph. A special feature when viewing states of an LTS is that rule matchings can be highlighted by the simulator. A user can thus perceive which elements of the state partake in the application of the next rule(s).

The *rule view* (see Fig. VIII.4) can be used to display a rule graph. Rules are displayed as graphs with color and line style distinguishing the elements (see the editor component). Layouting and highlighting/hiding of elements is possible in this view as well.

The *LTS view* (see Fig. VIII.5) is used to present the LTS as far as it is currently generated. The start state is always marked green and terminal states (in which no more rules are applicable) are marked red. If an LTS is only generated partially, the nodes which allow for further rule applications are shaded grey. By clicking on such a node the user prompts the system to generate and display all possible derivation steps originating in this node. A user can thus explore exactly that part of an LTS which is relevant to him. By using this manual exploration style, even unlimited LTS (originating in non-terminating rule sets) can be (partially) explored. By default, the LTS view employs a forest layout which emphasizes the sequences of rule applications.

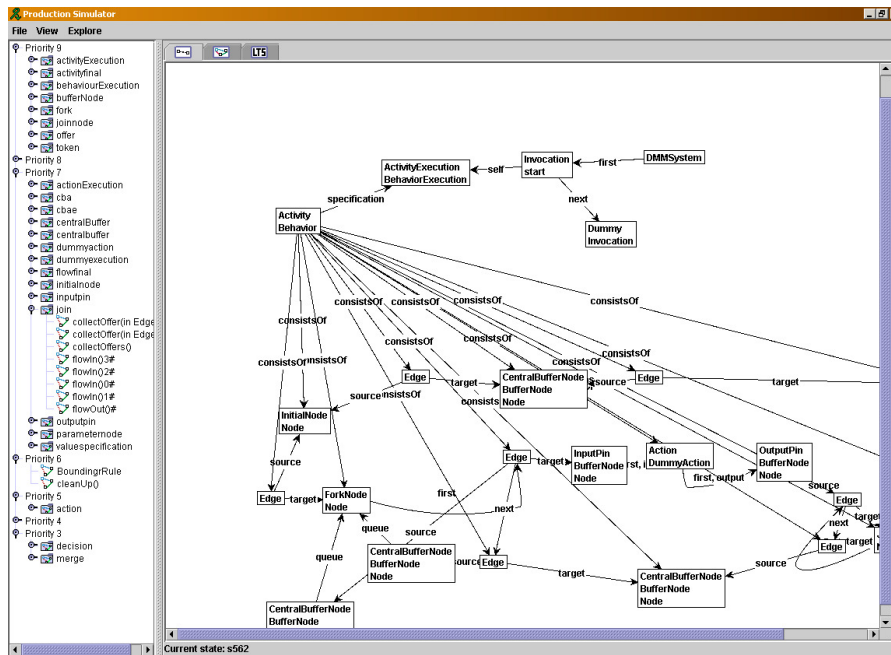


Figure VIII.3: The state view of the GROOVE Simulator

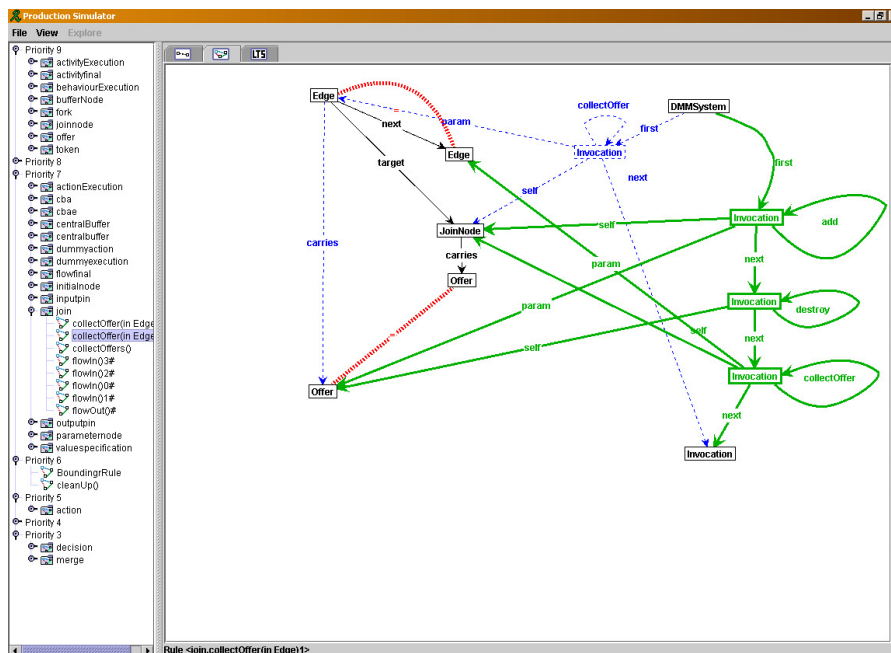


Figure VIII.4: The rule view of the GROOVE Simulator

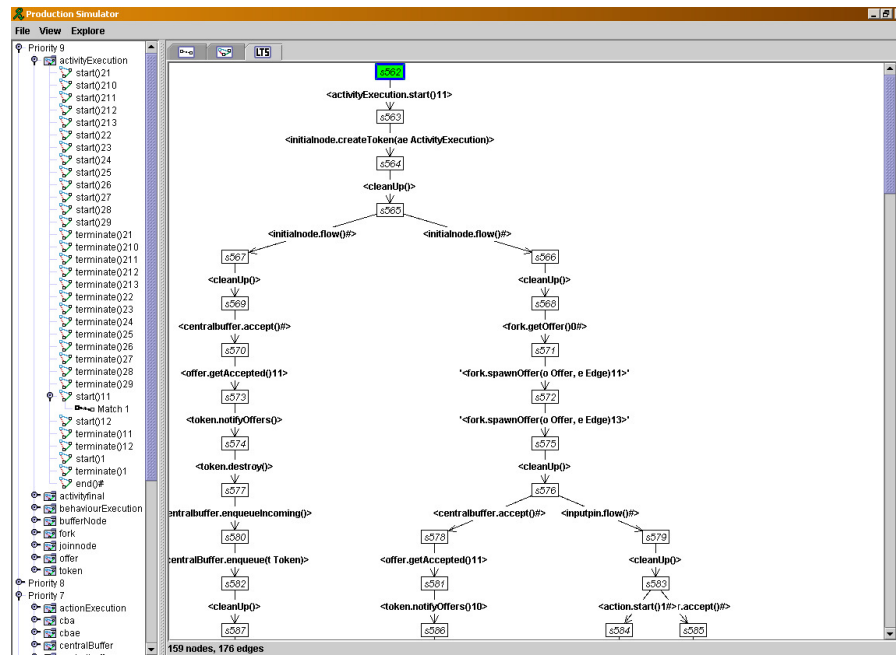


Figure VIII.5: The LTS view of the GROOVE Simulator

VIII.2.5 Graph Transformations in GROOVE

The graph notion of GROOVE is that of a directed graph with labeled edges. The node labels displayed in the GROOVE tools (cf., e.g., Fig. VIII.2) are technically labeled loop edges attached to the nodes. Using this technique, a node may even be multi-labeled (in Fig. VIII.3 several nodes carry multiple labels). Note that we will also display GROOVE graphs as node and edge labeled graphs since the layout of the label carrying loop edges makes for very cluttered and unclear presentations.

GROOVE follows the SPO approach to Graph Transformations and supports non-injective matches. It does, however, allow for a local suppression of non-injective matches by defining a special edge called a *merge embargo* edge. Two nodes which are connected by a merge embargo edge must not be matched to the same element in the host graph. Using merge embargo edges exhaustively results in injective matching.

Negative Application Conditions are supported in GROOVE. Connected elements whose labels are prefixed with *not:* form a NAC. Multiple NACS per rule are possible.

Universally Quantified Structures are not supported by GROOVE.

GROOVE allows control over rule applications in the form of priorities. Priorities are assigned to rules by prefixing a rule's file name with an integer. Higher priority numbers mean higher application priority. No other control constructs are supported by GROOVE.

We call this combination of features *GROOVE rules* in the remainder of this chapter.

VIII.3 Translation of DMM Specifications into GROOVE Specifications

The graph transformation approaches used in DMM (cf. Chap. IV) and GROOVE (see previous section) are rather different in many details. To actually execute DMM rules by the GROOVE tool set, we need a translation which expresses the additional DMM features in terms of GROOVE rules. We start by defining the translation of graphs and then move on to rules and control constructs.

VIII.3.1 Translation of Graphs

Graphs in DMM can either be type graphs, rule graphs or instance graphs. GROOVE does not support typing, thus type graphs are not translated. Only rule and instance graphs are translated, resulting in rules and state graphs respectively.

The graph notions of DMM and GROOVE are very similar, but several details differ. Basically, each node in a DMM graph is translated to a node in a GROOVE graph, each edge in a DMM graph to an edge in the GROOVE graph. Details of this translation are as follows:

Translation of Nodes

The encoding of state graphs is the more general case: Nodes in DMM can have a name(label), a type, and attributes. The name of the DMM node is transferred to a GROOVE node label. To distinguish it from type information, transferred node names are prefixed with the special character ”_“. The typing information of a DMM node is encoded by placing the name of the node’s types in the GROOVE node’s labels². Note that the complete typing hierarchy of a node is encoded in this way (cf. Tab. VIII.1).

Encoding rule graphs is simpler: Since the most specific type of a node suffices to match only the correctly typed nodes (all super types are implied), nodes which are part of the left hand side can be labeled with their direct type only. Node names are not translated in rules as they are only used in connection with the invocation mechanism (see below).

Concerning rule graphs, the differences in the matching notion between DMM and GROOVE must also be taken into account: GROOVE supports the more general notion of non-injective matchings but in DMM we assumed injective matchings only. To ensure that the translated DMM specification keeps its

²This kind of typing by labels was the usual way to handle typed nodes before the formal introduction of type graphs

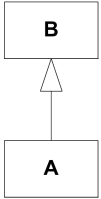

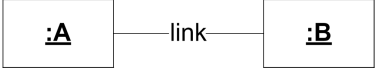

Concept	DMM	GROOVE
Node		
Edge		

Table VIII.1: Correspondence of graph concept in DMM and GROOVE

behavior, we need to explicitly disallow injective matchings in all GROOVE rules: All nodes in a rule graph that have either identical types or where one is a supertype of the other need to be connected by merge embargo edges.

Translation of Edges

Fundamentally, each DMM edge is translated into a corresponding GROOVE edge, i.e., an edge which runs between the translated DMM nodes.

GROOVE does not support the threefold label structure as defined for DMM. Encoding the DMM matching concept correctly would have necessitated a tripling of edges, creating three separate GROOVE edges (if all label components were used) for each DMM edge. We decided to use a simpler approach and fixed a single label for each edge in the DMM type graph.

VIII.3.2 Encoding of Rules

The rule notions of GROOVE and DMM are very close, thus all elements of r_{del} are marked with "del:" in GROOVE, elements from r_{new} with "new:", and elements from $NACs$ with "not:" (cf. Tab. VIII.2).

The rule signature of the DMM rule becomes the file name under which the GROOVE rule is stored. As the asterisk character "*", used in DMM to indicate big-step rules, is not a legal character for filenames under Microsoft Windows, it is replaced by the hash character "#".

Encoding UQS

Universally Quantified Structures pose big problems for the rule translation. Similarly to the unfolding procedure formalized in Subsect. IV.6.7, a DMM rule with universally quantified elements needs to be translated to multiple

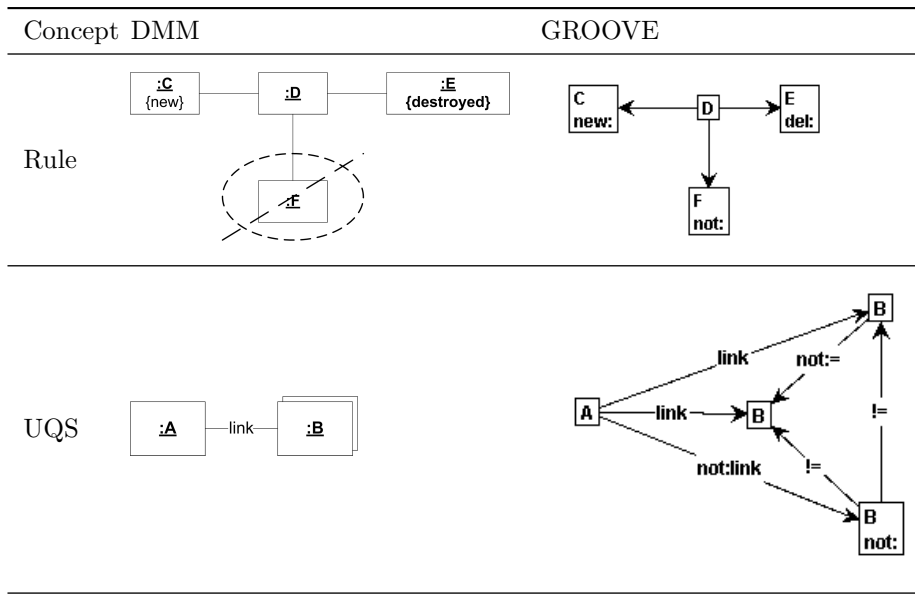


Table VIII.2: Correspondence of graph transformation rule concepts in DMM and GROOVE

GROOVE rules. Each of these rules will handle the case that a concrete number of instances of the UQS exists in the host graph. Several problems are apparent:

The number of potential occurrences of UQS is unlimited, thus the number of unfolded rules is infinite. For a practical translation we will thus have to fix the maximal number of UQS unfoldings the translation will produce. In Tab. VIII.2 a translation with two positive copies of the UQS is provided.

No unfolded rule may match if it does not cover all potential matches for the UQS, i.e., no elements may be forgotten. To achieve this, either the mechanism of priorities can be employed (giving rules with more copies higher priority) or an additional NAC must be employed to ensure that a further copy of the UQS would not find a valid match. We employ the latter approach.

Due to the non-injective matching supported by GROOVE, additional measures must be taken to prevent GROOVE from matching all positive copies of the UQS on the same elements. Merge embargo edges (as illustrated in Tab. VIII.2, row 2) are used to enforce injective matching in this case. A special kind of edge (labeled "!=") also prevents the NAC from matching on elements already matched by the positive copies of the UQS (cf. Tab. VIII.2).

VIII.3.3 Encoding of Application Control

Control of rule application is performed in DMM by the mechanism of rule invocations. GROOVE on the other hand supports no other control mechanisms than priorities. The invocation mechanism thus has to be encoded in a way as to retain its properties, but using only basic Graph Transformation constructs.

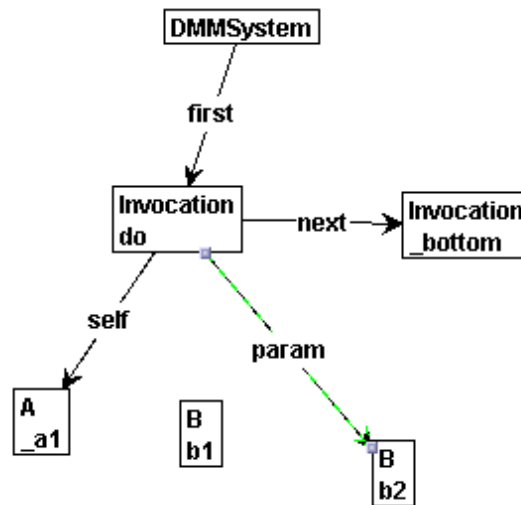


Figure VIII.6: Example state of the invocation stack

The suitable construction here is the explicit modeling of the invocation stack in the state graph. Each state graph contains a special singleton node called *DMMSystem*. Attached to this node are a number of invocation nodes, each of which signifies an open invocation in the system in this state. Each invocation node carries three pieces of information: A label with the name of the operation to be invoked, a self edge pointing to the node the rule was invoked upon, and a (possibly empty) set of *param* edges to nodes which have been passed as parameters at invocation time. Fig. VIII.6 illustrates the situation where the operation *a.do(b:B)* has been called on the node *a1* and the node *b2* has been passed as a parameter.

Invocation nodes form a linked list with the *DMMSystem* keeping a pointer to the first element and each invocation linking to the next open invocation. The *bottom* invocation is a special element which marks the empty stack and allows for uniform handling of the stack (i.e., there is always an invocation to push down).

When translating start graphs, a single *DMMSystem* node needs to be added to the translated graph. Attached to this by a *first* edge is the *_bottom* invocation node. If the start state requires an open invocation to start (i.e., the state itself is not sufficient to initiate the behavior) additional Invocation nodes can be queued in the *DMMSystem*.

Rules manipulate the *DMMSystem* stack by either enqueueing new invocations or fulfilling existing ones:

Translating Invocations

If a DMM rule specifies a rule invocation, the corresponding GROOVE rule must ensure that an invocation node is being created and pushed on the *DMMSystem*

stack. Invocation nodes are always placed on top of the existing stack, thus the existing first edge is redirected to point to the new invocation and the rest of the stack is linked by a `new:next` edge. See Table VIII.3 for an illustration.

If multiple invocations are made in a DMM rule, several such invocation nodes need to be created in the GROOVE rule. The ordering of these rules depends on the sequence number preceding invocation in DMM rules. If no such sequence number is given, an arbitrary order is chosen.

Translating Small-Step Rules

Small-step rule may only match when invoked. Thus in the GROOVE equivalent of a small-step rule, an `Invocation` node with the correct name and with the correct self-object needs to be present. As the invocation is fulfilled by the rule's application, the `Invocation` node can be deleted and the pointer structure of the stack is adapted accordingly. The illustration in Tab. VIII.3 displays very prominently how the binding of self and parameter objects works. The rule application can only match if the objects passed with the invocations can be integrated in the match of the rule. In the example, the rule itself requires the presence of an (arbitrary) `Y` and `Z` node. As the rule is being invoked, however, both nodes are already fixed by the parameters passed with the invocation: the `Z` node by being the context object (`self` edge) and the `Y` node by being passed as an invocation parameter (`param` edge).

Translating Big-Step Rules

While small-step rules depend on a (particular) invocation to apply, big-step rules depend on the absence of open invocations to apply. Thus, all GROOVE rules corresponding to big-step DMM rules are adorned with a construction requiring the `DMMSystem` to have a `bottom` invocation as the first element in the invocation stack (cf. Tab. VIII.3).

Encoding Premise Rules

Premise rules are encoded differently than the other rule types. As they directly influence the invoking rule's ability to match, they are merged to the invoking rule. Thus, the invoking rule is extended by the information provided in the premise rule. Elements which are passed as parameters comprise the interface of this merge.

NACs and UQS present in the premise rule are treated like elements of the invoking rule. Invocations of other premise rules (the only kind of invocations allowed in premise rules) are handled transitively. If the invoked premise operation has multiple rules implementing it then each of them is merged separately to the invoking rule.

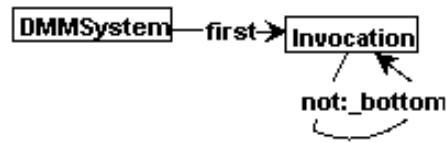


Figure VIII.7: Rule checkInvocation to separate failed and stable final states

Encoding Invocation Application

The execution of a DMM system does not guarantee that all invocations are successful. In final states (i.e., states where no rules are applicable anymore) we thus have to distinguish between failed states (in which an open invocation cannot be fulfilled) and stable states. To allow for this distinction in GROOVE, we insert a special rule `CheckInvocation` (see Fig. VIII.7) in the GROOVE rule set which checks whether the invocation stack still contains invocations (apart from `_bottom`). To restrict the application of this rule to final states only, it is given a lower priority than all translated rules. If none of the translated rules can match anymore, the application of the `CheckInvocation` rule signifies that the reached state is failed. If the rule is not applicable, the state is stable.

VIII.3.4 Combination of Translation Concepts

The separate translation concepts introduced in this section need to be combined to form a complete translation. The combination is straightforward and works along the order of presentation in this section. First, simple nodes and edges are being translated, then UQS are unrolled, and finally the invocation mechanisms are being encoded.

A complete rule translation is displayed in Fig. VIII.8 which shows the rule `cbae.end` from the DMM specification of UML Activity Diagrams in both the DMM representation and the corresponding GROOVE representation. We can observe that especially the encoding of the invocation features bloats the GROOVE rule in comparison with its DMM original. The GROOVE representations of premises and UQS can also create a lot of additional nodes. Understandability is significantly decreased during the translation. The GROOVE rules are thus regarded as an internal format only and are not intended for user presentation.

VIII.4 Interpreting Activity Diagrams with GROOVE

Along the translation concepts introduced in the previous section the DMM specification for UML Activity Diagrams has been translated into a GROOVE specification. Since no automatic facility is currently available for this task,

Concept	DMM	GROOVE
Invoc.		
Small-Step		
Big-Step		
Premises		

Table VIII.3: Correspondence of application control concepts in DMM and GROOVE

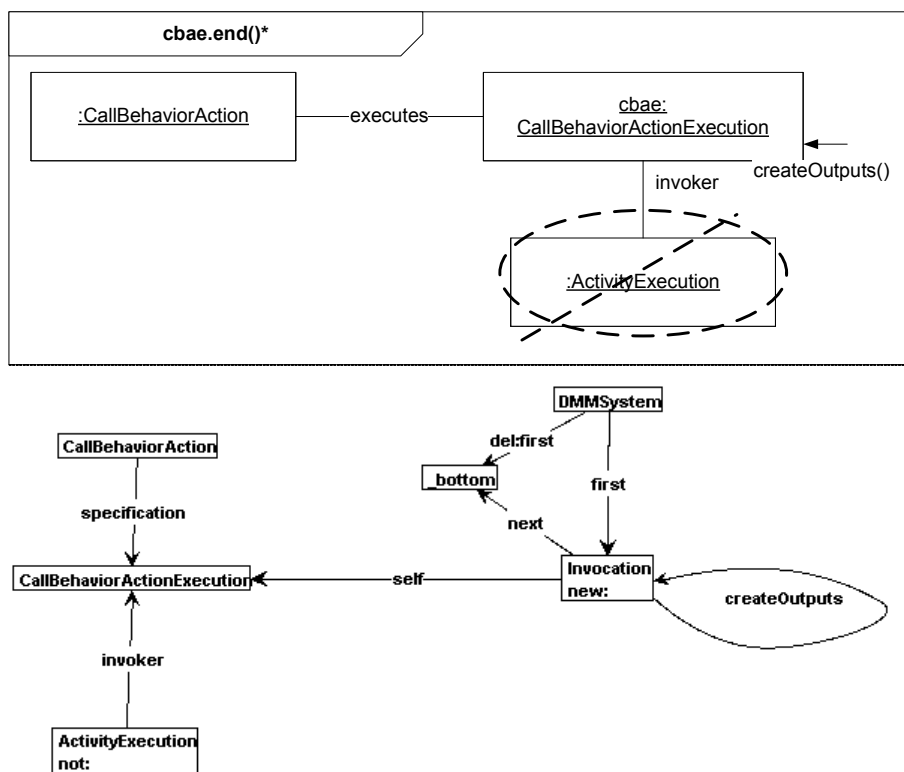


Figure VIII.8: Translation example: DMM rule `cbae.end` (top) and its GROOVE translation (bottom)

the translation has been performed manually³. The unfolding of UQS has been limited to a maximum of three copies. As a result the rule set grew from 75 DMM rules to nearly 180 GROOVE rules.

The automatic application of (translated) DMM rules by the GROOVE tool set also allows for a two-stage validation process. The first stage of this process is interesting to the Language Engineers using DMM. Upon conceiving a DMM specification of a VML, Language Engineers can now validate this specification by using test models. Each test model is a model in the described VML and the output of the automated DMM interpretation of these test models gives the Language Engineers insights whether their specifications correctly reflects their ideas. For these test models, positive as well as negative examples can be used, i.e., it can be tested whether things which should work do so and also whether things which are not supposed to work are correctly detected by the specification. Most interesting in this stage of validation is whether all expected outcomes of certain test models are computed by the DMM interpreter. The specification of the test's expected results is thus the Language Engineer's intuition on what should happen according to a specific test model. If these validations fail, changes to the DMM specification have to be made until the automated interpretation matches the intuitively expected results.

The second stage of validation actually works the other way around: After a DMM specification is defined as the language's standard semantics, users of this language can build an understanding about the language by automatically interpreting sample models. Similar to learning a new programming languages, the user can thus learn about language concepts from the specification or accompanying texts and validate his understanding by submitting sample models to an automated interpretation. If the DMM interpreter does not confirm his intuitive interpretation, the user can follow the interpretation in a step-wise fashion to detect the point at which his interpretation diverges from that produced by the machines. He can then proceed to refine his understanding about the language by investigating this difference and its cause more closely. This process is again well-known to software engineers as it closely resembles the testing and debugging of programs. Once again we see that DMM employs skills present in the target audience. Thus, automated DMM interpretations add to DMM's claim to understandability.

VIII.4.1 Validating the Activity Diagrams Specification

We performed the former kind of validation for our DMM specification of Activity Diagrams. A number of (minor) errors was detected (and subsequently corrected) in the specification. We can now state confidently that the rule set provided in Chapter VI works well and that it reflects our interpretation of Activity Diagrams.

The following test models (amongst others) were use to validate the specification:

³Performed by the very diligent and patient Daniel Beverungen

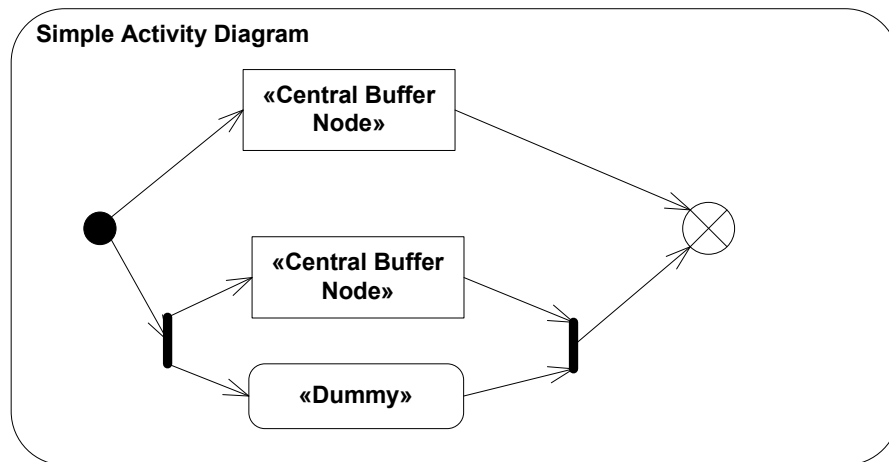


Figure VIII.9: Test model: Simple Activity Diagram

Simple Activity Diagram The simple Activity Diagram (cf. Fig. VIII.9) comprises control flows and some commonly found control nodes. The starting configuration for the test case includes an invocation for `ActivityExecution.start`.

Control Structure The control structure example is introduced in Subsect. VI.1.2 and showcases the difficulties of evaluating complex and interwoven paths between actions. We tested two configurations of this example. In the first configuration we assumed that actions A and B had created their outputs, i.e., both carried a token on their output pin. In the second configuration we placed two tokens on the output pin of A and none on the output pin of B.

Nested Activities The third example describes the process of designing, producing and mailing party invitations in two activities. The example comprises object flows, action invocations and actions with mixed inputs. Note that all actions apart from `Produce invitations` are Dummy actions (i.e., they don't do anything beyond consuming and producing tokens). The start state for this example entails an invocation of `Invite Guests`.

Carpenter Deadlock The Carpenter Deadlock example is taken from Conrad Bock's article on data flow in Activity Diagrams [Boc04]. It is intended to highlight the synchronization of input pins on an action. The starting configuration comprises one object token in each «centralbuffer».

VIII.4.2 Results of the Test runs

The described models were processed by the GROOVE tool with the (translated) specification of Activity Diagrams. Figure VIII.13 gives an impression

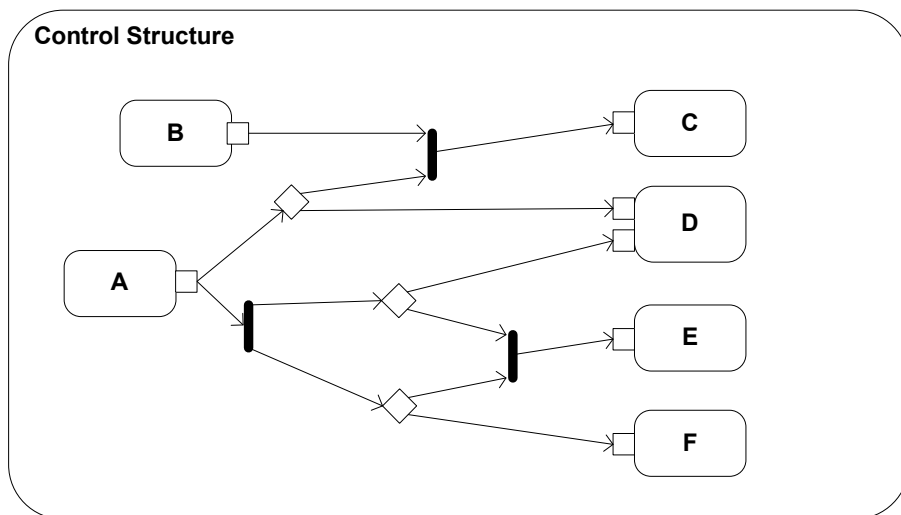


Figure VIII.10: Test model: Control Structure

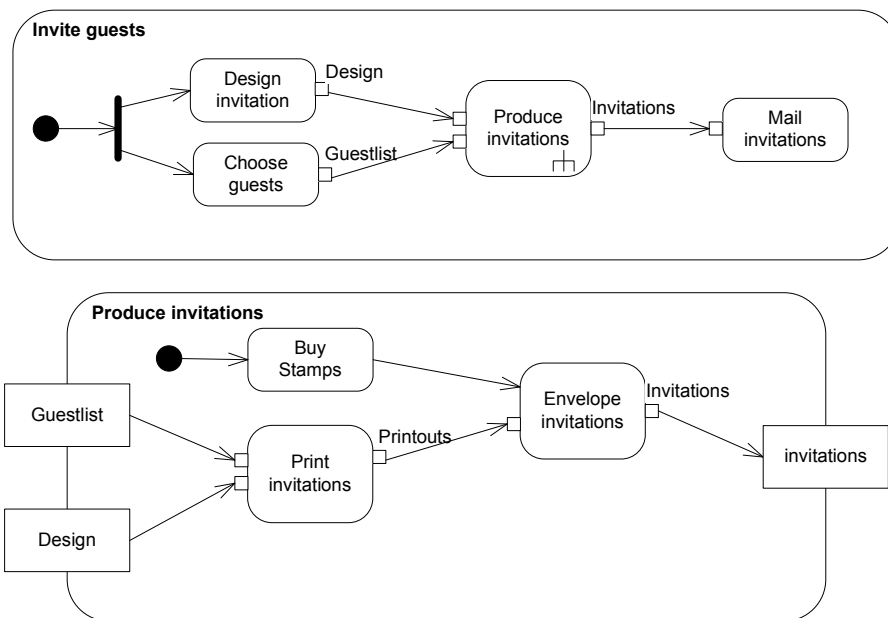


Figure VIII.11: Test model: Invitations

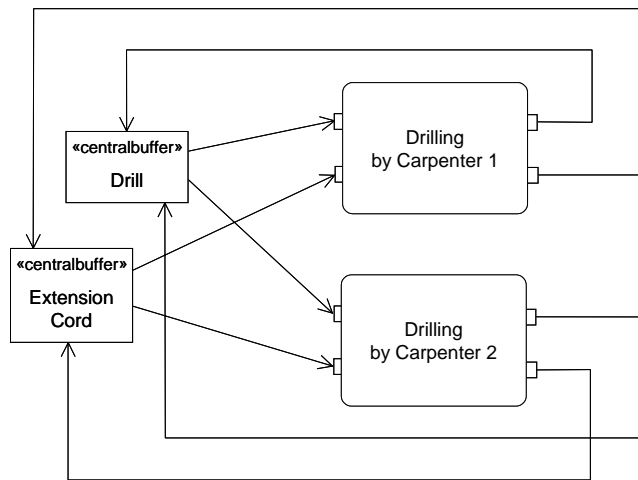


Figure VIII.12: Test model: Carpenter Deadlock

on the size and presentation of the generated LTS. All of these results conform to our intuitive expectations.

Three interpretation results were especially interesting:

- ◆ With the Control Structure test model (configuration 1) our intuitive expectation about the possible action executions was confirmed (cf. Subsect. VI.1.2). However, the interpretation additionally revealed that only one of these executions (execution action D) allowed for the enclosing activity to terminate as all other possible action executions left remaining tokens in the activity graph. This was an effect which we overlooked in our manual inspection and which proves the necessity of having a systematic (i.e., automatic) facility for applying DMM specifications.
- ◆ The difference between the two configuration of the Control Structure example was unexpectedly wide. While the configuration 1 produced 224 states, the LTS of configuration 2 was almost ten times bigger. This rapid growth was caused by the high degree of concurrency by having offers for two tokens competing in their traversal of the control structure.
- ◆ A third interesting observation is that despite the obvious non-termination of the Carpenter Deadlock test model, the resulting LTS is finite. The reason for this effect is that GROOVE checks encountered states for isomorphism and thus produces a circular LTS.

Beyond validating our DMM specification of Activity Diagrams, the test runs also give an impression on the efficiency and limitations of the GROOVE system. Table VIII.4 provides a number of measurements taken when testing different test cases. All measurements were taken on a Personal Computer, 2,4 GHz processor speed, 1024 MB main memory, Microsoft Windows XP operating system with a standard installation of Java 1.5. The GROOVE generator version 1.2 was used as a stand-alone tool, i.e., we measured only generation time for the LTS without any layouting overhead. The generation time given in the table

<i>Test Case</i>	<i>t_{full}</i> (in sec)	<i>#states</i>	<i>#transitions</i>	<i>#big - st.</i>
Simple AD	1,45	118	125	31
Control Structure (Conf. 1)	3,10	224	266	108
Control Structure (Conf. 2)	33,12	2882	3468	1064
Invitations	7,74	581	666	184
Carpenters	2,46	201	258	130

Table VIII.4: Measurements of test runs with the Activity Diagram rule set applied by GROOVE

is the average of three independent runs of the system.

VIII.5 Discussion of the GROOVE Prototype

Two kinds of conclusions can be drawn from our prototypical realization of a DMM interpreter with GROOVE. On the one hand we can focus on the shortcomings of the prototype and derive ideas for dedicated DMM interpreters from it. On the other hand we can focus on its merits and discuss which implication these have for the DMM technique as a whole.

VIII.5.1 From the GROOVE Prototype to a Dedicated DMM Interpreter

Concerning the efficiency of the GROOVE tool set we can state that generation times remained within reasonable limits for all examples. These are, however, only small examples and the underlying rules set only covered a subset of Activity Diagrams which are themselves only a part of UML. An important observation is that especially the amount of concurrency in the system is a determining factor for the size of the generated LTS and thus for the generation time. This is especially apparent in the two configurations of the Control Structure example which yielded vastly different results. Note that the degree of concurrency depends on a) the formulation of the rule set, b) the model under consideration, and c) the given start state. A slight change in the latter (moving one token to another pin) caused the LTS of the control structure to expand by factor 10. This is a specific instance of the state space explosion problem generally encountered in model-checking. We can thus expect that examples with a high degree of concurrency will only be amendable to automatic interpretation in a very limited way. Further experiences with the DMM approach are necessary to provide guidance for evading this problem.

For human comprehension of the generated LTS, however, generation efficiency is not even the main problem. A much more noticeable restriction is the efficiency of the GROOVE imager component which is barely able to handle graphs with several hundred nodes. Basic operations like zooming or panning the view in the 2000+ nodes graph generated by the Control Structure examples took an

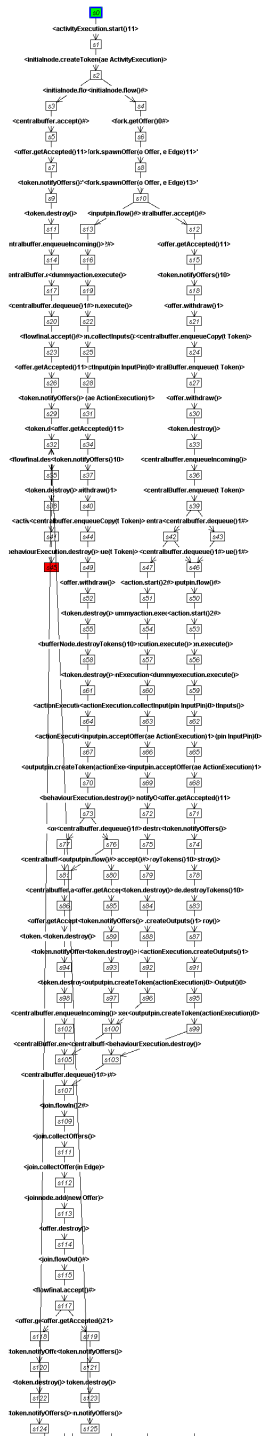


Figure VIII.13: LTS of the Simple AD example

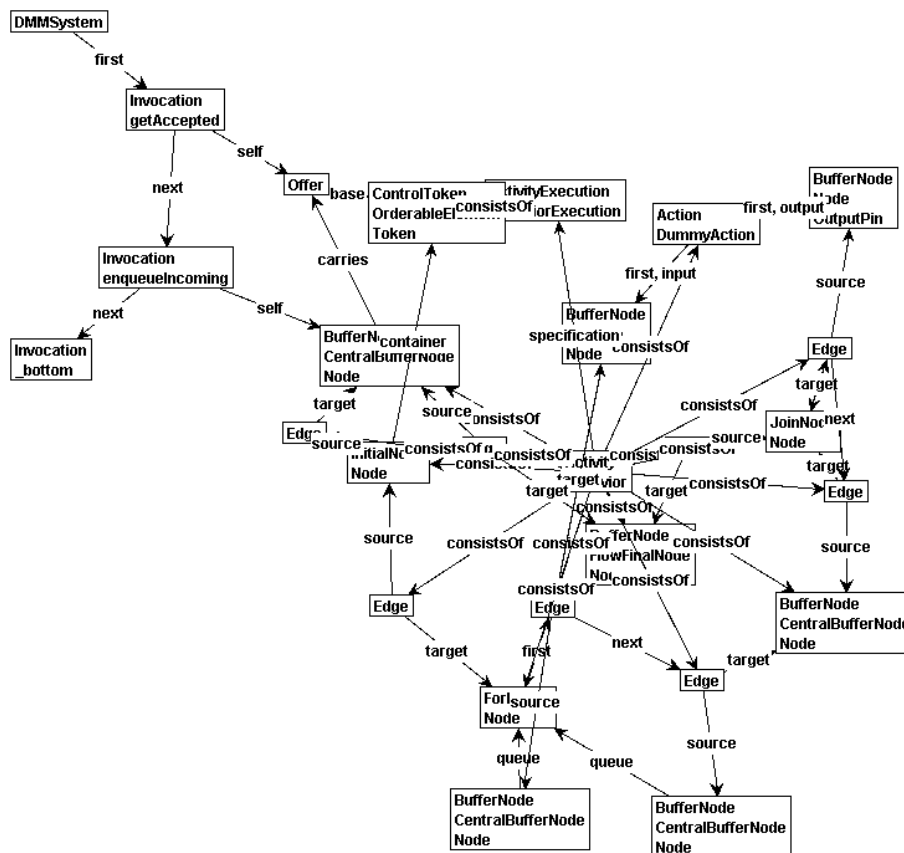


Figure VIII.14: Example State of the Simple AD LTS (original GROOVE layout)

unacceptably long time (30-60 seconds). From the efficiency point of view, this component is thus the weakest point of the GROOVE tool set at the moment.

Looking beyond the efficiency issues of the tool set we also detect a number of problems for an efficient human comprehension of the generated results. Fig. VIII.13 gives an impression of the smallest LTS generated by our examples (the LTS of Simple AD). Note that the figure shows the complete LTS, i.e., no reductions have been performed. Each of the nodes in the LTS in the figure represents a state graph which itself contains around 30 nodes (as depicted in Fig. VIII.14). Even if we assume a reduction of the LTS to big-step rules only (see the last column of table VIII.4), the comprehension of transition systems of this size remains a considerable task.

Essential for comprehending a complete LTS is the comprehension of its states. The display in Fig. VIII.14 shows a state of the LTS generated by the Simple AD example in GROOVE's standard (spring) layout. Several severe problems are evident:

Overlaps The first obvious deficiency of the state display is a weakness of the layouting algorithm of the GROOVE imager component. Many labels overlap each other and make it impossible to perceive the graph's elements. The graph must be manually untangled before it can be comprehended.

Element overload The graph has a high degree of connectivity, making a clean layout very hard. Especially the central **Activity** node which aggregates all of its elements (via the **consistsOf** links) introduces a lot of edges which carry little information.

Loss of diagram layout Comprehending the state fundamentally means comprehending its purely extensional elements, i.e. the **Token**, its **Offer** and their respective connections to the **Activity Graph**. To quickly grasp this crucial information it would be helpful to keep constantly present elements in a fixed spatial arrangement, preferably that of the original diagram.

Lack of element distinction Since the state graph is a GROOVE graph, all of its elements are labeled nodes, uniformly rendered as boxes. For the comprehension of a state graph, however, the re-introduction of concrete syntax would be most helpful to quickly distinguish between the different types of elements.

As GROOVE is only a prototype for a DMM interpreter we can use these shortcomings to derive a vision of an ideal visualization of an LTS. For the complete LTS such a visualization would not only comprise an automatic projection to big-step rule applications only but also an improved (overlap-free) layout and additional information in the transition labels. Fig. VIII.15 gives an impression of the effect of these improvements.

The visualization of information in a state graph can be enhanced even further. Fig. VIII.16 displays a possible visualization of the information also present in Fig. VIII.14. Here, we retained the original diagram notation and layout and added symbols to indicate semantic entities. A shaded square is used to designate the token, a shaded circle to indicate an offer and a dashed line to signify their connection. The box in the upper right hand side of the figure indicates the state of the invocation stack. Upon choosing a derivation sequence

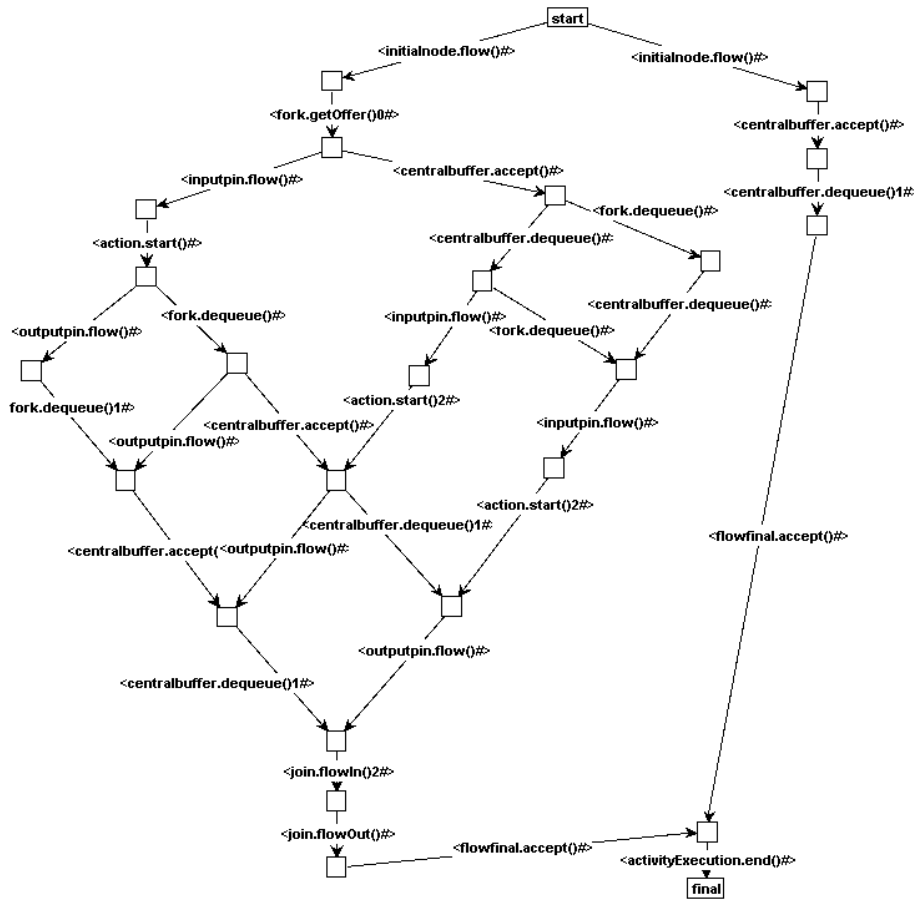


Figure VIII.15: Manually improved view on the Simple AD LTS

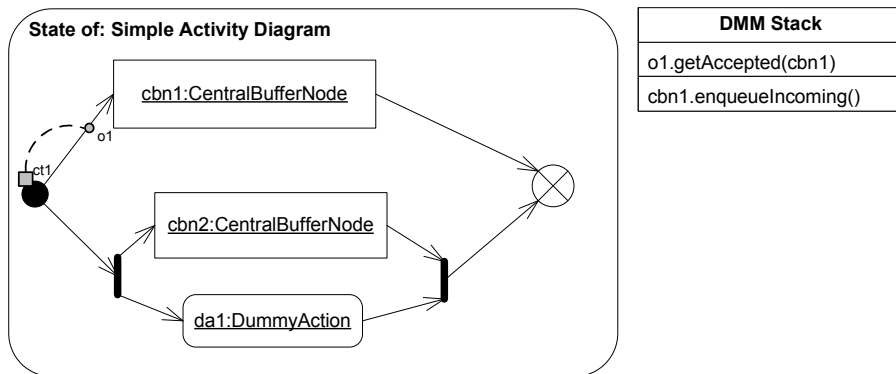


Figure VIII.16: Example of a state graph visualization

through the LTS, a user could even watch an animation of the changing elements in such a kind of visualization.

To achieve these improvements in presenting the results of an automatic DMM rule application, a tight integration of an editor component with the imager would be necessary to transfer the required information on the model's symbols and their spatial placement. A suitable technology to complement extensional elements with concrete syntax symbols would also be required. Such works toward an integrated editing/generation/visualization environment for DMM specifications are not the topic of this thesis anymore but remain open problems to be solved in future work along the topic of DMM.

VIII.5.2 On the Impact of the DMM Interpreter Prototype on DMM

The prototype DMM interpreter proves a number of points we made in this thesis:

DMM specifications are precise and formal enough to warrant an automatic interpretation. Using the prototype and the DMM specification of Activity Diagrams it is now possible to derive the meaning of every activity diagram (using only the basic and intermediate constructs) automatically. Debates on the meaning of certain models can thus be settled without need for human interpretation.

DMM allows for the testing of sample specifications. While the automatic interpretation does not add any information which cannot be gained from a manual inspection of the rules, it shifts the emphasis to the generated LTS. Here, the step-wise behavior induced by the operational semantics component of DMM is displayed very prominently. Our experiences showed us that it is easily possible to detect errors in the DMM specification by applying it to test models, to trace the origins of these errors, and to rectify them.

A major help in this process is the syntactic closeness of the test model, its semantic representation, and the rules defining its semantics. This is a further advantage which DMM holds against the denotational approaches presented in Sect. II.3 which use specific languages for the formulation of the semantic domain, making the tracing of errors and interpretation of analysis results rather hard.

The ability to perform a step-wise generation of the LTS also allows for a useful interpretation of models which produce unlimited LTS. While full generation of such LTS is not possible, users can still gain useful information about their behavior.

In terms of analyzability, the automatic interpretation also reveals an urgent need for notions of abstraction. The constructed LTS grows quickly and the rich structure of its states (each containing the whole system state graph) forms a vast body of information. To quickly gain useful results from such a system, information must be reduced, either by reducing the state information (in the extreme case discarding all state internals) or by reducing the set of observable

transitions (e.g., observing only big-step rules or observing only a selected set of rule applications).

To summarize: GROOVE allows for the convenient and efficient application of DMM specifications to concrete examples, proving the feasibility of the DMM approach. The practical application of DMM with such a tool has been demonstrated by validating the DMM rule set for Activity Diagrams. For visualization as well as for analysis purposes, the quantity of the generated information present problems to be tackled in future work.

Chapter IX

Summary and Conclusions

In this chapter we summarize the contributions of this thesis (Section. IX.1). Complementing this summary is a brief overview of related publications on DMM and their connections to the contents presented here (Section. IX.2). We also cast a critical glance at our results and discuss achievements and open questions in Sect. IX.3. Finally, a summarizing conclusion is given in Sect. IX.4.

IX.1 Summary of the Contributions of this Thesis

The main contribution of this thesis is the definition of Dynamic Meta Modeling, a technique to express the semantics of Visual Modeling Languages. The technique is especially geared toward understandability for a wide audience with the stated goal of being fit to serve in a published language standard. It reaches this goal by combining established techniques in an innovative way to form a precise and formal core. To present this formal core it re-uses established notations of the UML which are very well known to the intended audience. In contrast to many other approaches, DMM is also very flexible in the way a Language Engineer may formalize his intentions. Both the semantic domain meta model and the rules can be specifically tailored to express the intended concepts. A methodology to guide the user in this task has also been provided.

Two main technical parts of DMM are innovative contributions in themselves: Meta Relations have been the subject of two publications [HK03, Hau03] and form a necessary addition to the OMG's meta modeling framework. The invocation mechanism of DMM rules is a novel concept for controlling the application of Graph Transformation rules. It expresses this control within the rules and in a way which it is aligned to object-oriented structurization concepts.

A final contribution is the content of the case study. It forms—to our knowledge—the first formalization of UML 2's Activity Diagrams which takes the fundamental traverse-to-completion mechanism into regard.

IX.2 Overview of Publications on DMM

The first mention of the term Dynamic Meta Modeling and an outline of its construction (as a pure operational semantics description then) can be found in [EHS99]. The elaboration of these ideas was the topic of a master's thesis [Hau01] and resulted in a contribution to the UML 2000 conference [EHHS00].

Issues of tooling for the approach were explored in [HHS00] and the theoretic relations between Structured Operational Semantics and Graph Transformations were elaborated in [CHM00]. The extensibility of UML and its support in DMM were the topic of [HHS01]. This work also broadened the underlying example from UML Statecharts to UML Sequence Diagrams. The combination of these two diagram types in a consistency testing approach employing DMM was outlined in [EHHS02]. The handling of diagrams which express temporal information by DMM was introduced in [HHS02a] and elaborated in [HHS04].

A first work toward Meta Relations was an application of Graph Transformations to Stuart Kent's Relationship pattern [HHS02b]. In cooperation with Stuart Kent the concept of Meta Relations was developed during a research stay at the University of Kent at Canterbury. The language of Meta Relations was introduced in [HK03] and its use for MDA laid out in [Hau03].

The final conception of DMM as a hybrid semantic approach has been presented at a Dagstuhl seminar on Language Engineering [BH05]. The elaboration of the Activity Diagrams case study formed the basis for scientific tutorials [HH04, HH05] and a cooperation with Harald Störrle, resulting in [SH05].

IX.3 Discussion of DMM

Some critical questions can be posed to the claims made in this thesis:

▷ *Are DMM specification really as understandable as you claim?*

Understandability for the intended user group has permeated every aspect of DMM's construction. No special formal notations/concepts need to be grasped to read or to actually write DMM specifications. We do thus believe that DMM has strong potential to be very understandable.

In the scientific community, the DMM idea has been received very favorably. The underlying concept of expressing operational semantics by Graph Transformations has since been taken up by other researchers. Experiences from a lecture/exercise at the 2004 Segravis Summer School also indicate that the technique is rather easy to pick up. Participants were able to specify Statechart semantics with simple rules very quickly and got to the point of debating the realization of finer points like inter-level priorities.

What we can not say at the moment is how the non-scientific audience reacts to DMM. To actually embark upon acceptance tests in this group, a suitable tool environment (beyond the GROOVE based prototype of a DMM interpreter) would be necessary.

Future work toward improving the understandability of our approach should provide a way to present DMM specifications (i.e., DMM rules) as well as model semantics (i.e., LTSs) using the concrete notation of the formalism. A first example for such a style of presentation has been provided in Fig. VIII.16.

▷ *Is DMM really universal with regard to the UML? Can it formalize all of UML's features?*

We have provided formalizations of (substantial subsets of) Activity Diagrams in this thesis, of UML Statecharts in [EHHS00], and of an prescriptive interpretation of Sequence Diagrams in [HHS04]. We do not see any fundamental problems in formalizing the remaining elements of UML's prescriptive behavioral diagrams with DMM. Interaction Diagrams, however, follow a descriptive style of specification, i.e., they define which interaction traces are allowed/forbidden without exactly detailing the cause-effect relation between these interactions. The semantics of these diagrams could be captured in the DMM technique by interpreting their contents as constraints on the derivation sequences in the LTS. The exact nature of this constraining relation is as of yet unclear. First attempts to cover these descriptive diagrams with DMM can be found in [HKS01] which differentiates semantic dimensions of Sequence Diagrams and in [EHHS02] which tests the consistency of prescriptive against descriptive behavior specifications.

Static constructs have been mentioned only in passing in our works as they pose no great challenge. Such elements (Classes, Instance Specifications etc.) are supported well by the denotational meta modeling framework which is integrated in DMM.

Some parts of UML, however, have a informal semantics only. Most notably Use Cases defy every formalization since they are intentionally imprecise and informal. Such elements cannot be formalized without altering their character.

▷ *DMM seems tailored to the UML's needs. But how well does it support the definition of other VMLs?*

DMM's claim to understandability rests for large parts on its UML-like appearance. For users of other visual languages who do not know UML this appearance effects no advantage. While it is possible to envisage other concrete presentations for DMM specification (resembling the modeling notion to be defined), we do not currently see other candidates readily available. But UML knowledge is rather far spread, thus we believe that DMM specifications can also be understood by users of other VMLs.

Another issue is the question which concepts can adequately be addressed by DMM specifications. We have to be aware that DMM expresses dynamic semantics in discrete steps. This is sufficient to support the discrete behavioral notions of the UML but if other modeling languages provide notations for truly continuous concepts, DMM will not be able to correctly express their semantics.

▷ *What are strengths, what are weaknesses of the DMM technique?*

The strength of the DMM as a whole is its flexibility in expressing the semantics of a language in adequate terms. It can thus be employed to express the semantics in a way which is not only visually appealing but which also allows targeting the intended audience by employing notions already known to it. A

weakness of this approach can be seen in the fact that the resulting semantic domain is newly invented, thus there are no theoretic results or tools readily available to analyze/process DMM specifications.

The Meta Relation Technique has its main strength in situations where single elements of both the syntax and the semantic meta model correspond. The more elements of the syntax need to be expressed by complex patterns on the semantic side (a sign for a wide semantic gap and an inadequate semantic domain) the less appealing Meta Relations become. Meta Relations thus advocate the use of rich semantic domain meta models in which the granularity of semantic concepts corresponds to the granularity of syntax elements.

The DMM rules employed for the definition of dynamic semantics are very suitable to describe complex local state manipulations. By using the invocation mechanism, such local manipulations can be combined to express more complex behavior. Behavior which either depends on global conditions or which has global effects can only be expressed in DMM by means of loop constructs.

▷ *Will applying DMM guarantee that UML only has one consistent, precise and stable semantics?*

DMM is a technique to *formalize* the semantic concepts expressed by a language. If these concepts are not consistent to begin with, the formalization will only make it easier to reveal such problems, it will not necessarily resolve them. The same holds for the precision of the semantics: If elements are supposed to have multiple possible meanings, a DMM formalization will only reflect this fact. The precision of DMM concerns the *form* of the semantics definition, not its *content*.

DMM does, however, allow for a *more focused* discussion on what the semantics of the specified language actually should be. It allows for clear demonstrations of the impacts of semantic decisions.

For the UML, we expect that the formalization of its semantics by DMM would result in a strengthening of the language. The (semantic) extensibility notion of DMM also suggests the definition of a common UML core with domain specific extensions allowing for additional interpretations and concepts. Multiple possible interpretations of UML elements could thus coexist but in a clearly structured framework, avoiding the semblance of semantic arbitrariness which today often prevails in debates.

▷ *Which tools are available to start using DMM?*

Currently only the *application* of (translated) DMM rule sets on a state graph is automated (by using the GROOVE tool set). The *design* of DMM specifications, the *translation* of the rule set and the creation of a suitable *start state* for a concrete model are currently done by hand. All of these activities should be supported by a semantic workbench in the future. Developing such tool support is the next step in realizing DMM's potential as only with these tools the benefits of DMM can be demonstrated efficiently to our intended audience.

▷ *What are the tangible benefits in using a DMM-specified language?*

The most tangible benefit in having a DMM specification is that disputes over the exact meaning of certain combinations of language constructs, special situations, etc. can be unambiguously settled. Users either inspect the semantics

specification directly or create sample models to be interpreted automatically. Models of DMM-specified languages are thus more reliable in that they can be exchanged without fear of diverging interpretations.

Especially for suppliers of language related technology, DMM specifications provide a clear base against which to build their approaches. This concerns researchers (interested in, e.g., model analysis or consistency) and tool builders (interested in, e.g., model transformation or code generation) alike. For DMM-specified languages research efforts toward these topics can be concentrated on a common base. We expect such language-related technologies to become available faster with a higher interoperability by using DMM in a language specification.

IX.4 Closure

Visual Modeling has found wide-spread acceptance in Software Engineering. Especially the UML has managed to permeate industry and academia alike. Yet, its informal semantics definition gives rise to endless discussions on the meaning of certain diagrams. Models are thus relegated to being informal sketches rather than precise and reliable development artifacts.

Many approaches have promised to rectify this situation by proposing formal techniques for the definition of the UML. What sets DMM apart from these previous works is that it takes the expected (advanced) users into account and aims for a semantics definition which both in its concepts and in its notations appeals to their previous knowledge. In this user-orientation, its integration of static and dynamic semantics, and its advanced features like support for extensibility of semantics, DMM is a unique proposal to realize the potential of formal semantics in a practical context. We believe that DMM stands a good chance of shaping the way Visual Modeling Languages will be defined in the future.

Appendix A

Overview of Activity Diagrams

For readers not familiar with UML Activity Diagrams this chapter provides a brief overview of their purpose, usage, and notations. An understanding of this diagram type must take its history and its role in UML 2.0 into account. We describe these aspects in Sect. A.1 and A.2, respectively. In Section A.3 we provide an overview of the central elements of the notation. The examples in this section revolve around a simple and intuitive party planning example also used in [HH04, HH05].

A.1 History of Activity Diagrams

Activity Diagrams are UML's version of dataflow modeling techniques. When the UML was inceptioned, flow-modeling techniques were first dismissed as being not object-oriented. But their popularity lead to a late integration in the UML 1.1 standard as a special kind of Statechart. While this design prevented an extension of the UML syntax meta model, it implied that Activity Diagrams had to obey the semantics for Statecharts. This entailed two major drawbacks:

- ◆ Forks and joins in Activity diagrams corresponded to concurrent states (also called And-States) in Statecharts. As such, they had to be well nested, i.e., a join could only integrate flows that originated in a corresponding fork. This was unsuitable for a number of application areas, notably workflow modeling (cf. [Pen03]), where behaviors from a whole range of participants have to be orchestrated in a number of ways that do not necessarily conform to this notion of well-formedness.
- ◆ UML Statecharts follow the run-to-completion (RTC) semantics. Essentially, this means that all events present during an execution step of the Statechart have to be completely processed before new events can be accepted. Since Activity Diagrams were also considered to be (a kind of) Statechart, this lead to a rather unwanted synchronization of concurrent activities.

Being rather a last-minute addition to UML, Activity Diagrams also lacked notational and conceptual support for modeling program logic (no high-level concepts to express loops, exceptions, etc.), had only a very weak model of resource assignment (swim lanes were not even represented in the meta model), and were generally not supported by many tools in a useful way.

Due to these deficiencies, the Request for Proposals for the UML 2 Superstructure states one of its four main goals as *”Remove restrictions on activity graph modeling due to the mapping to state machines.”* [Obj00].

A.2 The Role of Activity Diagrams in UML 2.0

Consequently, Activities (and Activity Diagrams as their primary notation) are one of the concepts which changed most in the move from UML 1.x to UML 2.0. They are now supposed to cover a wide range of behavioral models from the specification of use cases, over workflow modeling, down to the implementation of operations [Boc03d]. Together with the Actions definition, they form a computationally complete language. Integration to the other diagrams of the UML can happen in one of two ways:

- ◆ Activities as behaviors of classifiers: If classifiers have behavior (e.g., by declaring operations) an Activity Diagram can be used to specify this behavior. In Fig. A.1 (reproduced from [Boc03a]), the activity `deliver mail` can be used to “implement” the corresponding operation of the class `POEmployee`.
- ◆ Activities as stand-alone behavior: Activities can also represent behavior that is important (e.g., in a given problem domain) but which is not (yet) aligned with an object structure. Workflow definitions are examples for such stand-alone behavior. In Fig. A.1, the action `deliver mail` can be such a stand-alone behavior, independent of any class containing it. In this case it can define properties, associations, and (meta-)operations of its own.

Having these two styles of using Activities available in UML allows for a gradual introduction of object orientation during a development. One can, e.g., move from Activities which describe essential behaviors of the problem domain to Activities which describe the behavior of computational objects solving problems in this domain.

Activities have become a first class behavior model in the UML, i.e., they now have their own meta model representation and are independent of Statecharts or other formalisms of the UML. Their role as compared to the other behavioral models (Statecharts and Sequence/Communication Diagrams) is described in the specification [Obj04], p.317 as follows: *”Activity modeling emphasizes the sequence and conditions for coordinating other behaviors, rather than which classifiers own the behavior.”*

An important distinction is that between Activity and Activity Diagram. The former is the model element (i.e., Activity is a class of the UML syntax meta model) which expresses an behavior in a certain way. The latter is the (usual) graphical rendering of this kind of behavior model. Other equivalent renderings

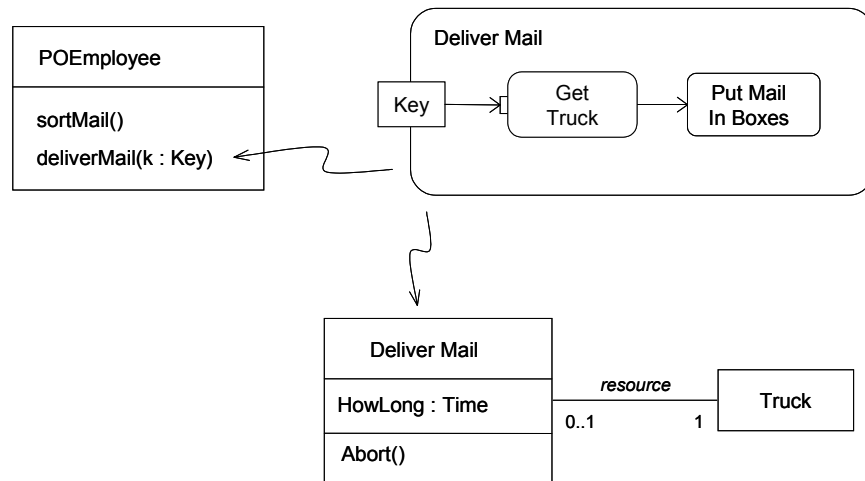


Figure A.1: Activities as behaviors and classifiers, reproduced from [Boc03a]

of the content of an Activity exist [Boc03d]. Usually, however, an Activity Diagram in the prescribed UML syntax will be used to represent an Activity. We thus use both terms synonymously.

The following subsection provides a brief overview of the main syntax elements of Activity Diagrams, enabling readers unfamiliar with this notation to follow the more in-depth discussions later on.

A.3 Activity Diagram Elements

Activity Diagrams are a diagrammatic way to represent Activities. An Activity Diagram consists of an enclosing rectangle with rounded corners, the name of the activity, and a directed graph. The graph is made up from three kinds of nodes (control, action, and object nodes) and two kind of edges (control and object flows). Note that a change of terminology has taken place in UML 2: Activities are now the high level units of behavior specification and their steps are called Actions (in UML 1 both levels were called activity).

The general notion of semantics as given in [Obj04] is that of token flow, i.e., when executing an activity, there exists a number of tokens flowing through the graph. Tokens can either represent control or objects/data. Action nodes will start executing upon the arrival of the necessary tokens, object nodes will temporarily store tokens, and edges and control nodes direct the tokens flows.

Action Nodes Action nodes are usually notated as small rectangles with rounded corners (examples for this and other notations are displayed in Fig. A.2). They capture (a part of) the actual behavior that is to be carried out within an execution of the enclosing activity, while all other elements only determine the order of actions to be executed. Each action node represents

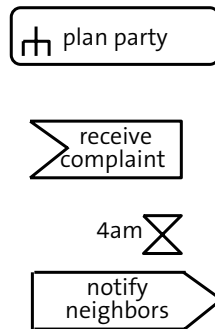


Figure A.2: Different kinds of UML actions. From top to bottom: CallAction with rake notation, AcceptEventAction, AcceptEventAction for TimeEvents, SendSignalAction

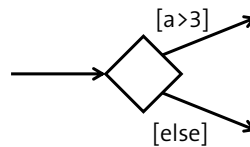


Figure A.3: Example for a decision node

a certain UML action that specifies the behavior executed by the node. Actions are UML's behavioral primitives. They can be used to perform low level modifications of the runtime state of a system (read/write variables/objects), carry out communication tasks (send/receive signals) and invoke other behaviors. Using the latter kind of action allows the construction of hierarchical Activity Diagrams, where one Activity Diagram might at some point execute a CallAction to invoke a behavior which is in turn specified by another Activity Diagram.

Each action node can have a number of incoming and a number of outgoing edges. To start executing the contained action, tokens must be offered on all incoming edges of the node. After the execution of the action is finished, a token is offered to each outgoing edge.

Control Nodes Control nodes do not have a behavior of their own. They are used to specify the sequences of actions to be executed in an activity. There are a number of different control nodes for this purpose, namely decision and merge nodes for alternative branching, join and fork nodes for concurrent branching, and initial and final nodes for starting and terminating activity execution. The following paragraphs will introduce these nodes in more detail. Common to all control nodes is the fact that they are unable to buffer tokens, i.e., a token will only pass control nodes to reach executable or object nodes.

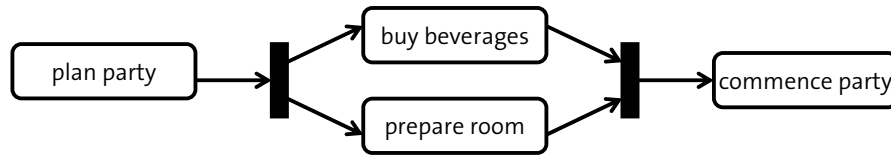


Figure A.4: Example for the use of fork and join node

Decision nodes Decision nodes are notated as a diamond with one incoming and multiple outgoing flows. Each of the outgoing flows is adorned with a guard condition. The intention of the decision node is to guide incoming tokens along one of the outgoing flows depending on certain conditions. A token will only move down one outgoing flow (so called XOR-branch) even if multiple guards are satisfied (in this case the choice is non-deterministic). The example in Fig. A.3 shows a decision node which will pass tokens along the upper flow if variable *a* is greater than 3, along the lower flow otherwise (the guard *else* is predefined and holds if all other guards fail).

Merge nodes Merge nodes are also notated as diamond shapes, only with multiple incoming and one outgoing flow. Their semantics is to pass every incoming token down the outgoing flow. Merge nodes can be used to re-integrate flows that have been split up at a decision node (analogous to an *endif* statement which integrates the *then* and *else* part of an *if* statement), but they can also be used without directly corresponding to a decision node. One thing to note is that merge nodes will pass on *all* offered tokens. In that regard they resemble a multi-merge or OR-merge and are thus not perfectly symmetric to decision nodes.

Fork nodes Fork nodes are notated as a bar with one incoming and multiple outgoing flows. The semantics of a fork node is to pass (duplicates of) each incoming token along all outgoing flows. This kind of node is used to indicate that the following portions of the activity can be executed concurrently. In the example in Fig.A.4 the end of the activity *plan party* enables the execution of *buy beverages* as well as *prepare room*. The different outgoing flows are not synchronized to achieve parallelism but can execute independently, i.e., it is possible to buy beverages first and prepare the room later on, or to clean the room first, or to do both of these things in an interleaved way.

Join nodes Join nodes complement fork nodes in that they integrate multiple flows. Using also a bar as the notational element, they are distinguished from fork nodes by having multiple incoming and only one outgoing flow. The semantics of join nodes is to produce an outgoing token if all incoming flows offer a token. Thus, a join node has the ability to re-unite concurrent flows that emerged from a fork node. In contrast to UML 1.x Activity Diagrams, no correspondence between forks and joins is required. Referring to the example

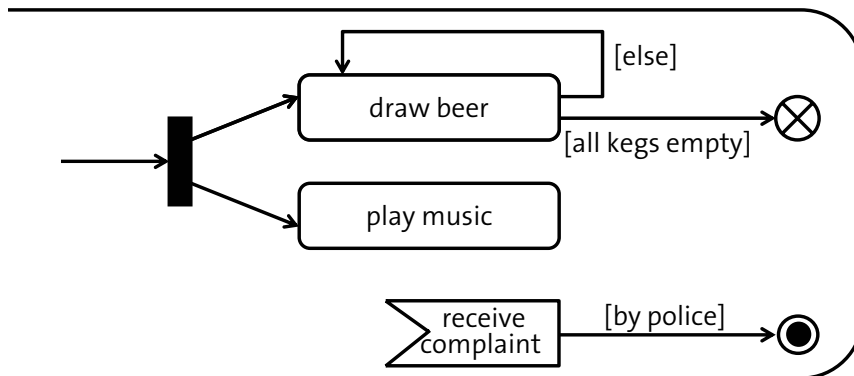


Figure A.5: Example for the use of final nodes

in Fig.A.4, the join node requires the actions buy beverages and prepare room to be complete before enabling commence party.

Initial nodes Initial nodes indicate the start of an activity. If the activity is invoked, each initial node produces a token. Initial nodes are notated as a small filled circle. Several flows may originate at an initial node. Guard conditions should ensure that the choice between these outgoing flows is deterministic.

Final nodes Final nodes consume tokens and thus stop the execution of an activity. There are two kinds of final nodes: flow final nodes and activity final nodes.

Flow final nodes consume all incoming tokens, thus ending a single flow of execution within an activity. If other tokens exist in the activity, they remain unaffected. In the example in Fig. A.5 a flow final node terminates the loop around the action draw beer if the kegs are empty. This does not affect any of the other flows, i.e., play music may still be executed.

Activity final nodes have an abort semantics: Once a token reaches an activity final node, *all* tokens currently existing in the activity will be destroyed, effecting a termination of all behavior. In the example (Fig. A.5) the reception of a complaint by the police will terminate the actions draw beer and play music immediately.

Object nodes Sometimes the triggering of subsequent actions is not facilitated by an explicit passing of control (i.e., a control token) but rather by supplying some kind of information or an object. To handle these situations Activity Diagrams supply object nodes and object flows.

Object nodes have angular corners and come in three kinds: They may be attached to an action node, then they are called pins and represent input and output data of that action. Other possibilities are that they describe parameters of an activity or appear as stand-alone elements in the form of data store

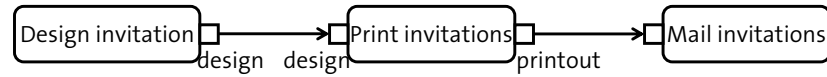


Figure A.6: Example for the use of pins

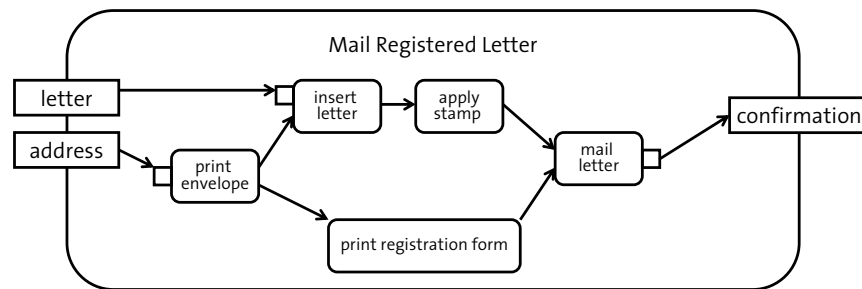


Figure A.7: Activity with in and out parameters and control and object flows

or central buffer nodes. The next paragraphs will provide details on these specializations.

Common to all object nodes is the ability to buffer one or more object tokens. Object nodes may specify the type and state of objects that they can store and may also specify a maximum buffering capacity. For nodes with a capacity greater than one, a selection scheme can be specified which determines the next token to be taken out of the buffer (pre-defined schemes include FIFO and LIFO). A notable semantic property of object nodes is that if multiple flows emerge from an object node, a token will only move down one of these (token competition).

Pins Pins represent data inputs and outputs to action nodes and are notated as small boxes attached to its border. When an action has both control and data dependencies (i.e. incoming flow both going directly into the action and into its input pins) all of these dependencies must be satisfied for the action to start. In Fig.A.6 the input pin attached to **Print Invitation** indicates that this action is to start when an invitation design is provided. Similarly, the output pin of the action indicates that the output produced by this action consists of (printed) invitations.

In the Fig. A.7 most action nodes are adorned with pins. An example for mixed dependencies is **insert letter** which requires a control token from **print envelope** and an object token **letter**.

Parameters Analogous to pins passing data into or out of an action execution, parameters capture data present when a whole activity starts or terminates its

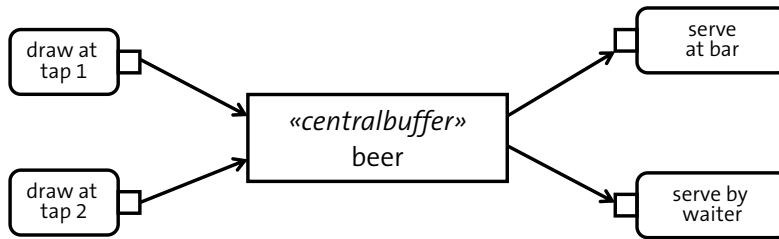


Figure A.8: Central buffer nodes collecting from multiple producers and distributing to multiple consumers

execution. Parameters are notated as boxes overlapping the borders of activities. The example in Fig. A.7 states that the mailing of a registered letter has two input (letter and address) and one output parameter (registration).

Central buffer and data store nodes Central buffer nodes are nodes in the activity graph that are notated as rectangles and carry the stereotype «central-buffer». Their role is to serve as a buffer for object tokens independent from any action. Buffering is useful in situations where the output from multiple actions is to be collected at a central place for later consumption or where multiple consuming actions draw from a common source. In the example in Fig. A.8 these possibilities are combined in that beers either drawn at tap 1 or tap 2 are collected in a central buffer node to be served at the bar or to be taken by a waiter.

Data store nodes specialize central buffer nodes in that only copies of buffered tokens are supplied to outgoing flows. Thus, data stores are a non-depleting form of buffer and retain all tokens ever passed to them.

Flows The arcs connecting the different kinds of nodes are called flows in Activity Diagrams. One can distinguish between control flows (carrying control tokens) and object flows (carrying object tokens). Each flow connects two nodes which can be of any of the above enumerated types. A flow can be adorned with a guard which specifies the conditions under which a token can pass the edge. If a weight is given for a flow, it specifies the number of tokens that pass the edge together.

A.4 Advanced Activity Diagram Elements

The previous paragraphs introduce the basic elements of Activity Diagrams but there are a number of concepts that have not been covered. Some of these are orthogonal to the basic elements (e.g., interrupt modeling), others extend the basic elements presented here (e.g., join specifications). The full details on all of these elements can be found in the UML specification [Obj04]. Good descriptions are also available in textbooks, e.g., [Stö05b, Pen03].

Appendix B

The DMM Specification of UML Activity Diagrams

This chapter provides the formalization of Activity Diagram' semantics in terms of DMM. We present this formalization by first giving a high-level overview of the package structure (Sect. B.1) of the semantic domain meta model. Along these packages we then introduce the details of the semantic classes, their operations, and the Meta Relations they participate in (Sect. B.2 to B.10). An excerpt of the information presented here is contained in Section. VI.2.

Note that this case study only formalizes the core elements of UML's Activity Diagrams, i.e., those elements described in App. A.

B.1 Overview of the SD Meta Model for Activity Diagrams

The semantic domain meta model for (the core elements of) Activity Diagrams which we provide as a case study for the feasibility of the DMM approach contains more than 30 classes. It is therefore subject to a high-level structurization by packages. Fig. B.1 provides an overview of the different packages we defined and their relationships.

In the remainder of this section we proceed to introduce the different packages and the classes they define. All descriptions are given in a standardized format which resembles the structure of the UML specification. We proceed in (roughly) the hierarchical order provided by the vertical placement of the packages in Fig. B.1. For each package we provide an overview which provides the rationale for grouping the elements of the package. Then we detail the classes contained in the package and provide information on their properties. Especially the operations of these classes are detailed in the form of DMM rules. For concepts which also appear in the UML specifications (either as syntax elements or as concepts in the semantics description) we specifically point out if we performed

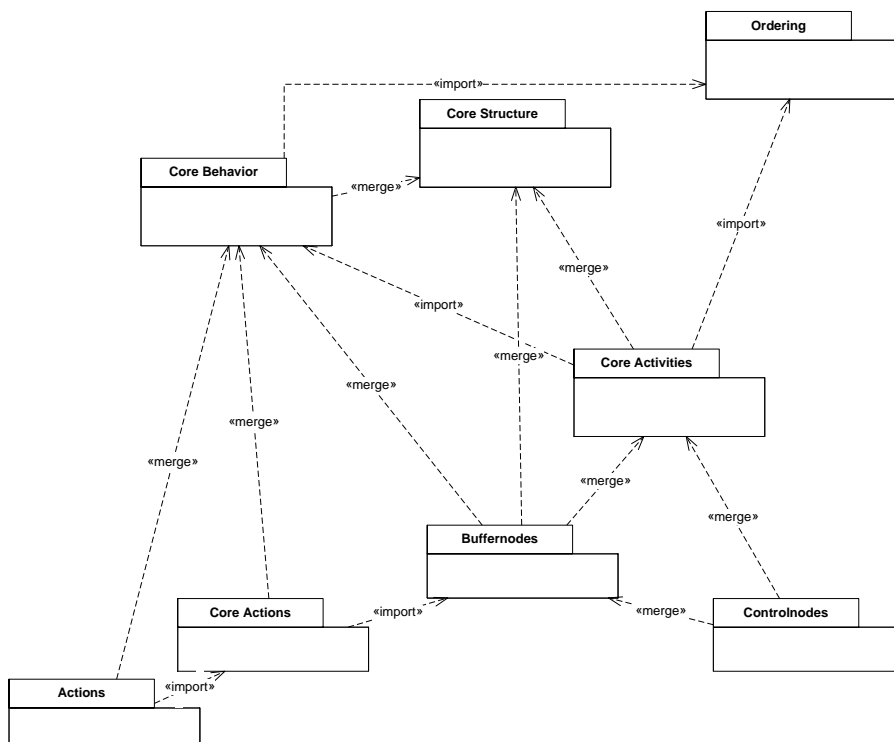


Figure B.1: The package structure of the semantic domain meta model for Activity Diagrams

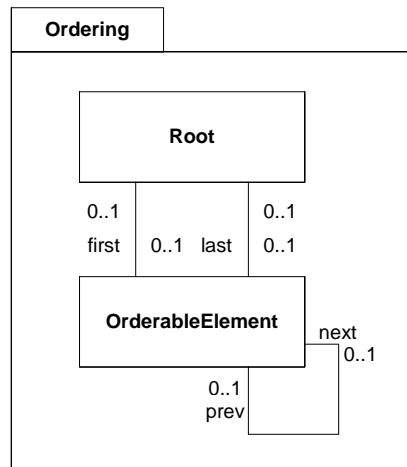


Figure B.2: The contents of the Ordering package

any modification to their original meaning. This is especially helpful for readers deeply familiar with the UML specification.

Note that for improved readability we included *all* information on a class in the package in which it is originally defined (even if some of its properties are not defined in that package but elsewhere). As we only have very few elements added by importing packages, this style of presentation works well here and allows for quickly localizing relevant information.

B.2 Package Ordering

Description The Ordering package (Fig. B.2) is an auxiliary package which provides means to order lists of elements. This facility is imported by multiple packages to provide structures which can easily be traversed by loop constructs.

B.2.1 Class Root

Description The Root Class is part of the order pattern illustrated in Fig. B.3. It denotes an element that holds a pointer to the first or last element of an ordered set (i.e., a list).

Package Root is defined in the Ordering package.

Associations

first [0..1] The first entry of the list.

last [0..1] The last entry of the list.

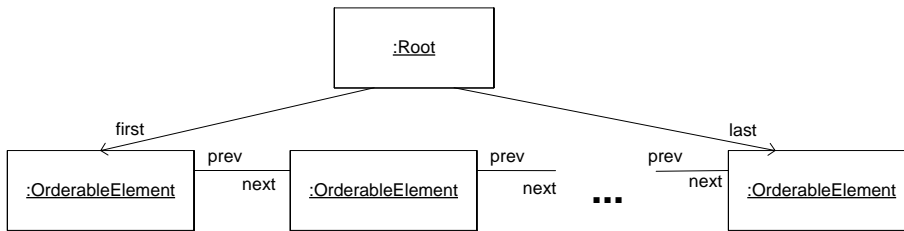


Figure B.3: Illustration of the ordering pattern

Constraints

```

context Root
inv
  self.first.prev->isEmpty()
  self.last.next->isEmpty()

```

The first and last element of a list must not be connected to further elements.

Operations (none)

Semantics A Root holds pointer to the head or tail of an ordered list. It is possible that both pointers are set, point to the same object (list with a single element), or are both null (empty list). Lists are not explicitly typed, but we expect them to mostly contain elements of a single type only.

Differences to standard UML Root is not a part of the UML syntax. It is an auxiliary element introduced to order sets.

B.2.2 Class OrderableElement

Description An OrderableElement is part of a list of elements of the same type. Lists can be ordered forward, backward or both.

Package OrderableElement is defined in the Ordering package.

Associations

next [0..1] Points to the next element in the list.
 prev [0..1] Points to the previous element in the list.

Constraints

```

context OrderableElement
  OrderableElement::allSuccs:Set(OrderableElement)
  OrderableElement::allPrevs:Set(OrderableElement)
  OrderableElement::selfAllSuccs:Set(OrderableElement)
  OrderableElement::selfAllPrevs:Set(OrderableElement)

  allSuccs= next->union(next->collect(e|e.allSuccs()))
  allPrevs= next->union(next->collect(e|e.allPrevs()))
  selfAllSuccs= allSuccs->union(self)
  selfAllPrevs= allPrevs->union(self)
inv:
  not(allSuccs->includes(self))
  not(allPrevs->includes(self))

```

Lists must never contain loops. The (recursive) queries `allSuccs` and `allPrevs` can be used for this check. The queries `selfAllSuccs` and `selfAllPrevs` yield the set of all succeeding/preceding elements including the element the query was invoked on. These queries are employed by elements using the ordering feature to ensure the totality of the order.

Operations (none)

Semantics An `OrderableElement` can be part of an ordered list. The ordering between elements of a list is bidirectional, i.e., an element knows its predecessor as well as its successor.

Differences to standard UML `OrderableElement` is not part of the UML syntax. It is an auxiliary element introduced to order sets.

B.2.3 Mappings

The classes of the `Ordering` package are auxiliary elements which are not directly related to elements of the syntax.

B.3 Package Core Structure

Description The package `Core Structure` as depicted in Fig. B.4 provides the basic structural parts of the semantic domain. For the current purpose (formulation of the semantics of activity diagrams), a very sparse population by just two classes is sufficient. A formalization of, say, `Class Diagrams` would add considerable detail in this package.

B.3.1 Class Class

Description A class is an instantiable concept. In the context of activity diagrams it is mostly used to type parameters of data objects.

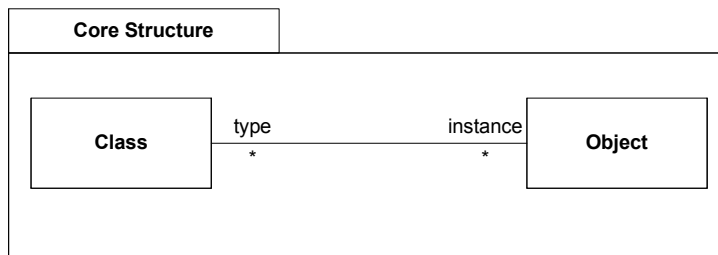


Figure B.4: The contents of the Core Structure package

Package Class is defined in the Core Structure package and extended by the packages Core Behavior and Buffernodes

Associations

Object [*] The objects that are direct or indirect instances of this class.

Parameter [*] (*added by the Basic Behavior package*) The Parameters which are of this type.

BufferNode[*] (*added by the Buffernodes package*) A class can define the type of a buffer node. Only object tokens with an underlying object of this class can be buffered on such a node.

Constraints (none)

Operations (none)

Semantics Classes are only used as the definition of data types in this domain and do thus not contain any dynamic semantics.

Differences to standard UML (none)

B.3.2 Class Object

Description An object is an instance of a class.

Package Object is defined in the Core Structure package and extended by the packages Core Behavior and Core Activities

Associations

type [1..*] The class(es) that form the type(s) of the object. An object always has a direct type which is the class its instantiated from. If that class inherits from other classes, however, the object can also claim to be of the type of the super classes. Thus the type association also connects an object with all classes in the inheritance hierarchy above its direct type.

Slot [*] (*added by the Core Behavior package*) An object can be used to fill a slot, i.e. act as the concrete value in a behavior invocation. This also implicates that in this particular formalization, parameters are always passed as references and not as values.

ObjectToken [*] (*added by the Core Activities package*) An object can be referenced by ObjectTokens. There can be many such references, with different tokens flowing through the same or through concurrent activities.

Constraints (none)

Operations (none)

Semantics Objects represent only (complex) data values in this domain. They do thus not have any dynamic semantics.

Differences to standard UML It is noteworthy that the association type references the direct type and all supertypes. In the UML meta model, type edges usually only indicate the direct type.

B.3.3 Mappings

The Meta relations targeting the Core Structure package are depicted in Fig. B.5. Two Relations define the connection between the syntax (UML meta model, left hand side of the figure) and the semantics (package Core Structure, right hand side of the figure)¹

CRep - The Class Replication Relation

```
context CRep
inv
  domelement.name=ranelement.name
```

The name of the class has to be preserved in the mapping.

¹As a presentation convention, we always place the UML meta model at the left hand of the mapping figures and assume it to form the *domain* of all Meta relations. The *range* of these Relations is always placed at the right hand side of the figures and labeled with the name of the semantic package under consideration.

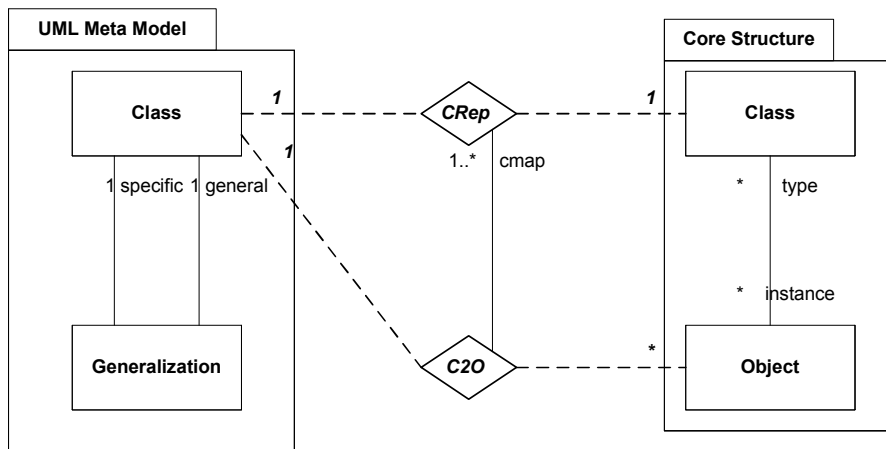


Figure B.5: Semantics mappings of the Core Structure package

C2O - The Class to Object Relation

context C2O

inv

```
cmap.domelement=domelement.allParents->union(domelement)
cmap.ranelement=self.ranelement.type
```

The constraints of the C2O Relation ensure that a) the objects of a (syntactic) class are always typed over its (semantic) counterpart and b) that all (syntactical) types are flattened. An object in the semantic domain will thus be typed over all classes which are (transitive) super types of its direct type².

B.4 Package Core Behavior

Description The Core Behavior package (Fig. B.6) provides the basic notions which underlie all behavioral semantics in UML. These are partially the equivalents of UML's core behavior package and partially their runtime equivalents. The Core Behavior package builds upon the Core Structure package (since parameters are typed) and the Ordering package (since parameters are ordered with respect to their defining behavior).

B.4.1 Class Behavior

Description A behavior is an executable specification.

Package Behavior is defined in the Core Behavior package and extended by the Actions package.

²This flattening does not aim to resolve name conflicts of provide semantics for multiple inheritance.

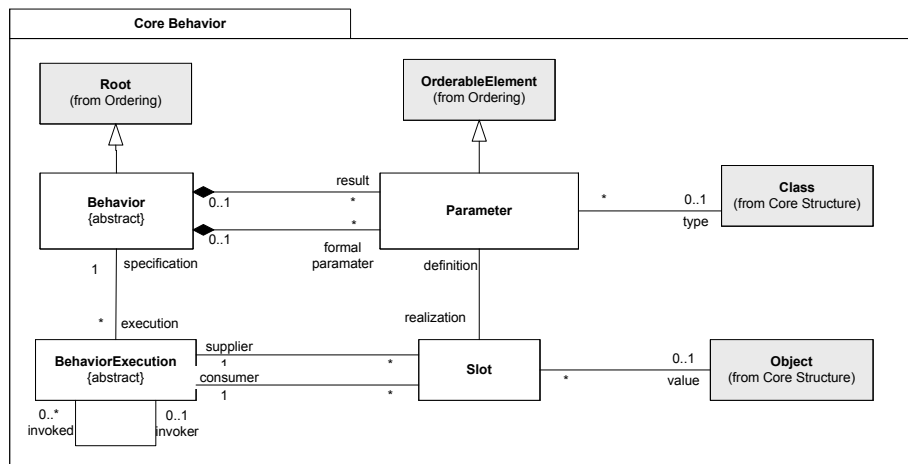


Figure B.6: The contents of the Core Behavior package

Associations

Parameter (as formal parameter) [*] A behavior can depend on a number of objects that are passed at its invocation. The type of these objects is defined by the formal parameters of an behavior.

Parameter (as result) [*] A behavior might produce a number of objects during its execution which it needs to pass on for further computation. The type of these objects is defined by the result parameters.

BehaviorExecution [*] Whenever a behavior is executed, a **BehaviorExecution** is created for it which controls the execution of the behavior. Since concurrent execution can occur, there might be a number of executions for one behavior simultaneously.

CallBehaviorAction (*added by the Actions package*) Behaviors can be invoked by other behaviors. A **CallBehaviorAction** is one such behavior which is able to invoke a behavior. In this formalization it is the only behavior with this ability, but there are more in the scope of the full UML.

Constraints (none)

Operations (none)

Semantics Every kind of conceivable executable specification is a behavior in UML. This includes the behavioral diagrams in UML (Activity Diagrams, Statecharts, Interaction Diagrams) as well as operations and other specifications.

Differences to standard UML (none)

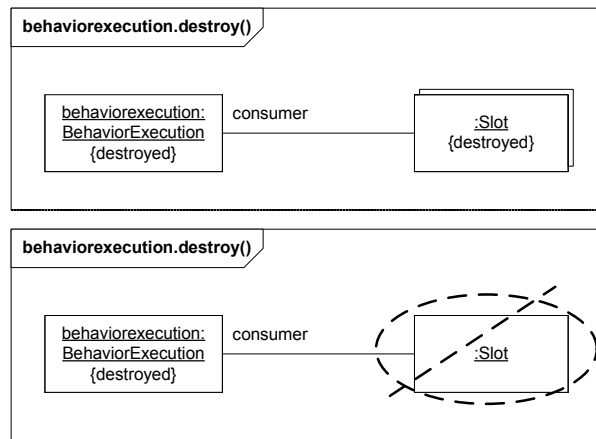


Figure B.7: DMM rules describing the destructor of a behavior execution

B.4.2 Class BehaviorExecution

Description A BehaviorExecution is the actual instance of a behavior. An instance of this class is present for every running behavior.

Package BehaviorExecution is defined in the Core Behavior package.

Associations specification [1] The behavior which is being executed

Slot [*] (*as supplier*) Slots which supply values to the execution, i.e. concrete values for formal parameters.

Slot [*] (*as consumer*) Slots which hold result objects of the execution.

BehaviorExecution [0..1] (*as invoker*) The behavior execution which has invoked the current execution.

BehaviorExecution [*] (*as invoked*) The behavior executions which the current execution has invoked. In the case of asynchronous invocation, there might be multiple such invoked executions.

Constraints

```
context BehaviorExecution
  BehaviorExecution::allInvokedExecutions:Set(Behaviorexecution)
  allInvokedExecutions= invoked->union(invoked->collect(e |
    e.allInvokedExecutions()))
inv:
  not(allInvokedExecutions->includes(self))
```

A behavior execution might not be its own invoker.

Operations `destroy()` The operation `destroy` facilitates the destruction of the `BehaviorExecution` itself and all slots for which it forms the consumer (cf. Fig. B.7). Slots for which the execution is the supplier are untouched by this rule since they might contain results that an invoking execution might want to consume later.

Semantics A `BehaviorExecution` is the representation of an execution of the specifying behavior. Its concrete semantics are determined by its subclasses as different behaviors have different executions (see semantics of `ActionExecution` and `ActivityExecution`). The only fixed semantics for all forms of `BehaviorExecution` is the responsibility for removing slot elements upon the execution's end.

Differences to standard UML Behavior Execution is not present in the UML syntax.

B.4.3 Class Parameter

Description A `Parameter` is the definition of a data value that is to be passed into or out of an invoked behavior.

Package `Parameter` is defined in the `Core Behavior` package and extended by the `Buffernodes` package

Associations

Behavior [1] (*as formal parameter or result*) A parameter always belongs to a behavior. It can either be a formal parameter and represent data that is to be passed to an execution of this behavior or it can be a result and represent data passed from this execution.

Slot [*] Whenever a behavior is executed, slots are created which can hold objects for the different parameters a behavior has. Since behaviors can be called concurrently, multiple such slots can exist for a parameter.

ParameterNode[*] (*added by the Buffernodes package*) Parameters can get represented in activity diagrams by the graphical element `ParameterNode`. Each such node must correspond to one parameter, but multiple such representations can exist. Note that `ParameterNode`s represent parameters "internally. i.e. from the viewpoint of the executed behavior, while input and output pins are used to pass data to a behavior that is to be invoked ("external view).

Class [1] Each parameter has a fixed type, i.e. it can only represent objects instantiated from a certain class (or one of its subclasses).

InputPin [*] (*added by the Core Actions package*) A formal parameter of a behavior can be represented in an activity as an input pin on a `CallAction`. Thus, input pins are mapped to the parameter they represent.

OutputPin [*] (*added by the Core Actions package*) A result of a behavior can be represented in an activity as an output pin on a **CallAction**. Thus, output pins are mapped to the parameter they represent.

Constraints

```
context ParameterNode
inv
not(self.Behavior->isEmpty())
self.represents.oclType=OutputPin implies
  self.Behavior.result->includes(self)
self.represents.oclType=InputPin implies
  self.Behavior.formalParameter->includes(self)
```

A Parameter is always owned by a Behavior (either as a formal parameter or as a result). Parameters mapped to output pins must always be results for their owning behavior. Parameters mapped to input pins must be formal parameters for their owning behavior.

Operations (none)

Semantics Parameters simply define slots which are used to pass data into and out of behavior executions in a defined way. They do not have dynamic semantics.

Differences to standard UML (none)

B.4.4 Class Slot

Description A slot can temporarily hold objects for the transfer into or out of behavior executions.

Package Slot is defined in the Core Behavior package.

Associations

BehaviorExecution The **BehaviorExecution** for which this slot holds values. The slot can either play the role of supplier or consumer (holding formal and result parameters respectively)

Parameter[1] Each slot is defined by a parameter.

Object [0..1] Each slot can hold at most one object as its value. Since some slots represent results, they do not carry any values as long as the execution runs. Likewise the objects from the formal parameters can be removed or modified during the execution of the behavior.

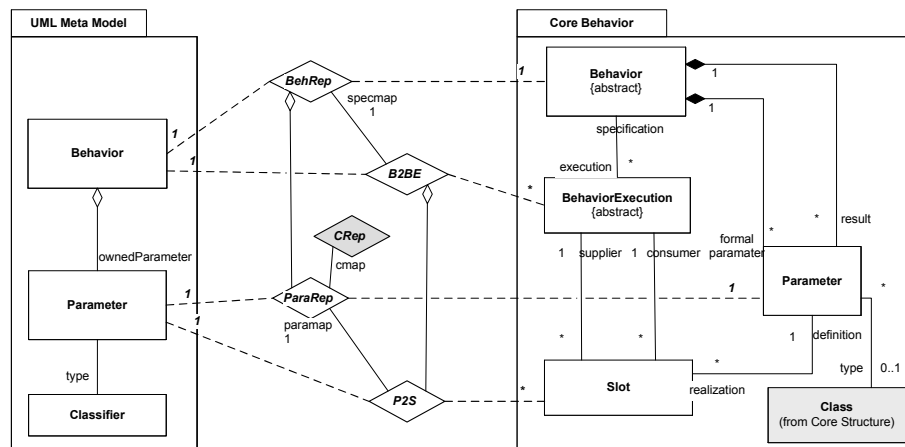


Figure B.8: Semantics mappings of the Core Behavior package

Constraints

```
context Slot
inv:
self.value.type->includes(self.definition.type)
```

The value objects must conform to the type of the underlying parameter.

Operations (none)

Semantics Slots are the concrete places where the values for parameters are stored. They are created and destroyed with the execution of the activity they belong to. They do not have dynamic semantics of their own.

Differences to standard UML (none)

B.4.5 Mappings

The semantic mappings targeting the Core Behavior package are BehRep, B2BE, ParaRep, and P2S (cf. Fig. B.8)

BehRep - The Behavior Replicaton Relation

```
context BehRep
inv:
domelement.name=ranelement.name
```

The names of all behaviors must be preserved in the semantic domain.

B2BE - The Behavior to Behavior Execution Relation

```
context B2BE
inv:
  specmap.domelement=self.domelement
  specmap.ranelement=self.ranelement.specification
```

These constraints ensure that the BehRep mapping is correctly respected when mapping BehaviorExecutions.

ParaRep - The Parameter Replication Relation

```
context ParaRep
domain=scope.ranelement.ownedParameter
range=scope.domelement.Parameter
inv:
  domelement.name=ranelement.name
  (scope.ranelement.ownedParameter->select(par|par.direction=in or
    par.direction=inout))->includes(self.domelement) implies
    scope.ranelement.formalparameter->includes(self.ranelement)
  (scope.ranelement.ownedParameter->select(par|par.direction=out or
    par.direction=result))->includes(self.domelement) implies
    scope.ranelement.result->includes(self.ranelement)
  cmap.domelement=self.domelement.type
  cmap.ranelement=self.ranelement.type
```

The domain and range definition of the Parameter mapping ensure that all Parameters of a behavior are mapped with it to the semantic domain. The invariants ensure three things: First, the name of the mapped parameter is preserved. Second, the parameter direction is preserved³. Third, the typing of the parameter is preserved.

P2S - The Parameter to Slot Relation

```
context P2S
domain=scope.domelement.ownedBehavior
range=scope.ranelement.Slot
inv
  parmmap.domelement=self.domelement
  parmmap.ranelement=self.ranelement.definition
  (scope.ranelement.ownedParameter->select(par|par.direction=in or
    par.direction=inout))->includes(self.domelement) implies
    self.ranelement.consumer=scope.ranelement
  (scope.ranelement.ownedParameter->select(par|par.direction=out or
    par.direction=result))->includes(self.domelement) implies
    self.ranelement.supplier=scope.ranelement
```

The last two invariants ensure that the Slot is connected to its BehaviorExecution in the correct role. Slots for formal parameters of a behavior are read

³The preservice of the parameter direction has become more complicated as the OMG changed the encoding of what is a result parameter during the finalization of UML 2.0. What was an association before (as it still is in the semantic domain meta model) is now a specific attribute only. Employing the flexibility which the semantic mapping provides, we can address this change without adapting the semantic domain meta model.

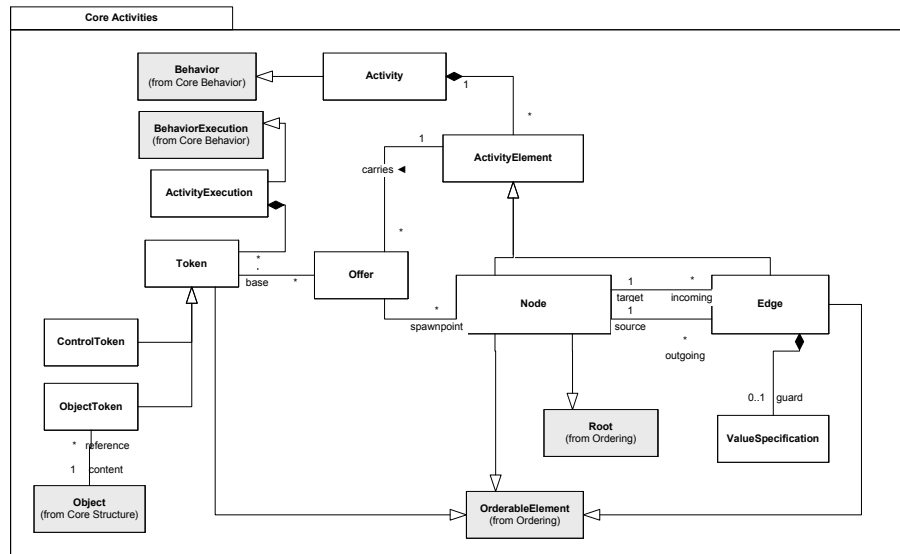


Figure B.9: The contents of the Core Activities package

(consumed) by the execution of the behavior, slots for results are written (supplied) by the execution of the behavior.

B.5 Package Core Activities

Description The Core Activities package (Fig. B.9) defines the core concepts of activities, namely the (syntactic) structure of nodes and edges and the runtime concepts of tokens and offers. Again, the Ordering package is employed to allow for traversing collections of elements and the Core Structure package is merged to associate an Object to an ObjectToken.

B.5.1 Class Activity

Description Activity is the base class that contains all other structural and runtime elements.

Package Activity is defined in the Core Activities package.

Associations

ActivityElement[*] The elements defining the Activity's behavior.

Constraints

```

context Activity
inv::
  self.first.selfAllSuccs=self.Parameter

```

All parameters (formal as well as result parameters) of an Activity must be in a total order.

Operations none

Semantics An Activity captures a definition of behavior in the form of an activity graph. The ActivityElements define the graph of nodes and edges which specify the inner structure of the Activity. This behavior is twofold: On the one hand the flow semantics of activities specify the order in which actions contained in the graph can be executed. The actions on the other hand specify the single steps of behavior which, when combined, form the activity's behavior. An activity is only the structural composition of elements whose dynamic semantics are combined to form the semantics of an ActivityExecution.

Differences to standard UML (none)

B.5.2 Class ActivityElement

Description ActivityElement is the fundamental building block of activities. An ActivityElement can carry any number of offers.

Package ActivityElement is defined in the Core Activities package.

Associations

Offer [*] the set of offers carried by the ActivityElement.

Activity [1] the Activity of which this ActivityElement is a part.

Constraints (none)

Operations (none)

Semantics Offers are carried by an ActivityElement. An ActivityElement does not have dynamic semantics of its own. The subclasses of ActivityElement contain rules how offers can move onto and from an ActivityElement.

Differences to standard UML ActivityElement is not part of the UML syntax. The UML meta model defines the notion of ActivityGraph instead which contains Nodes and Edges.

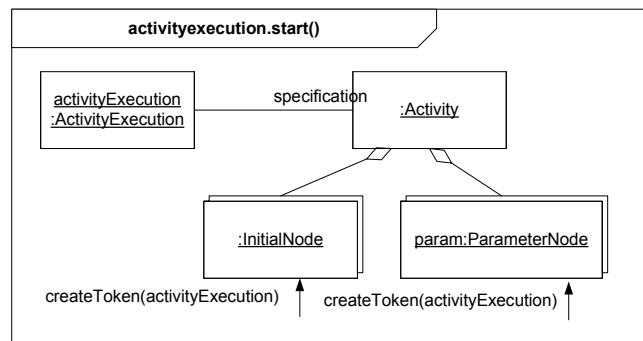


Figure B.10: DMM rule describing the start of an ActivityExecution

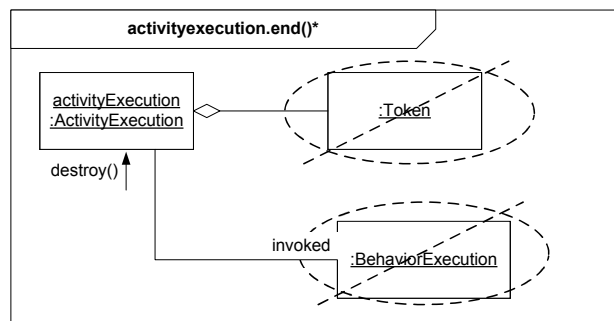


Figure B.11: DMM rule describing the end of an ActivityExecution

B.5.3 Class ActivityExecution

Description An ActivityExecution is the representation of an actual execution of an Activity.

Package ActivityExecution is defined in the Core Activities package.

Associations Token [*] The token(s) which represent the current execution state of the activity.

Constraints (none)

Operations

`start()` If an ActivityExecution is started, control tokens are created on all initial nodes and object tokens are created on all parameter nodes of the respective activity. The rule in B.10 provides the formal specification of this operation.

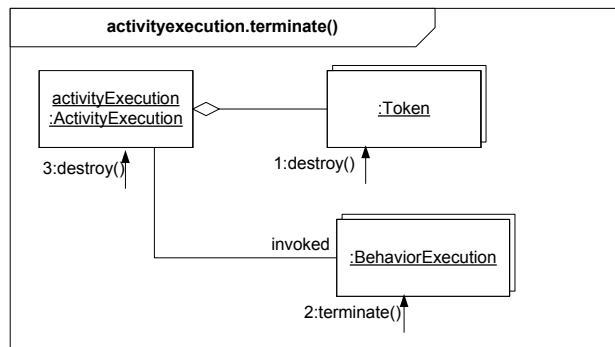


Figure B.12: DMM rule describing the cancellation of an `ActivityExecution`

`end()`* An `ActivityExecution` ends (cf. Fig. B.12) if no more tokens can advance the state of the activity execution and there are no more running behavior executions invoked by the activity execution (in particular there are no more action executions of this activity). In this case the activity execution is being destroyed (see `BehaviorExecution` for semantics of the `destroy` operation)

`terminate()` The `terminate` operation (Fig. B.12) can be invoked to stop a running execution. Termination of an `ActivityExecution` entails the termination of all subordinate behaviors and the deletion of all currently existing tokens of this `ActivityExecution`. The difference between ending and terminating an activity execution is that termination is an active process, triggered from the outside, which interrupts running behaviors, while ending an activity execution only consists of the recognition that the execution has come to its natural end and may thus be deleted.

Semantics An `ActivityExecution` represents the actual execution of a behavior specified by an activity. An `ActivityExecution` keeps track of the different tokens and invoked `ActionExecutions` which represent the state of the execution. If all tokens have been consumed and no contained action executes anymore, the `ActivityExecution` ends.

Differences to standard UML (none)

B.5.4 Class Token

Description Tokens are used to represent the location of control or data.

Package `Token` is defined in the Core Activities package and extended by the `Buffernodes` package.

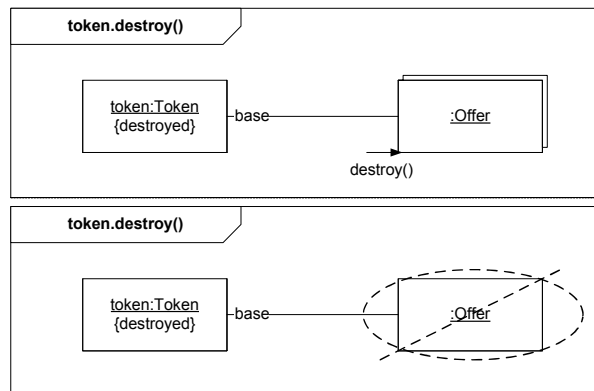


Figure B.13: DMM rules for the token destructor

Associations

ActivityExecution [1] A Token is owned by an ActivityExecution.

Buffernode [1] (*as extended by the Buffernodes package*) A Token is always contained by a BufferNode.

Constraints

context Token

inv:

```
this.activityExecution.specification=this.container.activity
```

Tokens of an ActivityExecution can only be contained by nodes of the defining activity.

Operations

destroy() When destroying a token (see Fig. B.13) all offers representing this token have to be destroyed as well.

copyContent(original:ObjectToken) A Token has the ability to copy its content to a newly created token. This operation has different semantics for control and object tokens. Since control tokens do not have any content, the operation does nothing (see Fig. B.14, bottom). For Object Tokens (see Fig. B.14, top) it copies the link to the content object to the new token. This overloading of the operation allows a more uniform handling of tokens by other rules.

withdrawOffers() The withdrawing of offers becomes necessary if changes happen to the underlying token which invalidate its currently emitted offers. In this formalization this withdrawing is only triggered when the token moves. All offers (if any are emitted) are deleted by this operation (cf. the rule in Fig. B.15).

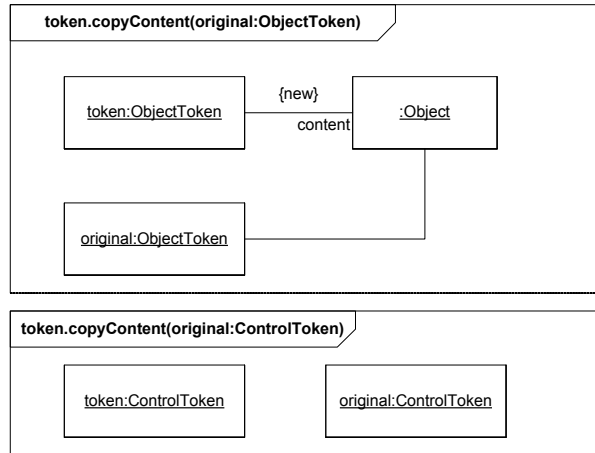


Figure B.14: DMM rules for copying the content of an object or control token

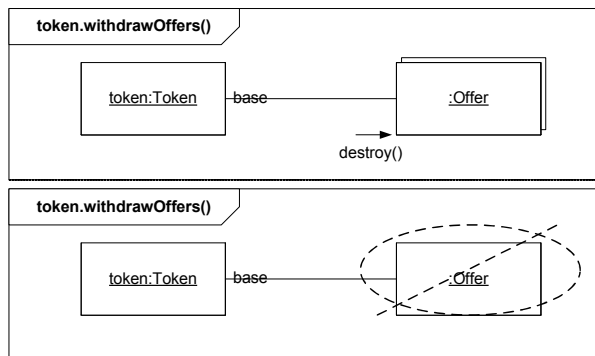


Figure B.15: DMM rules for withdrawing the offers of a token

Semantics Tokens represent control or data that is currently available for the execution of actions. Tokens move to another node if that node has accepted one of their offers. Actions need tokens to execute. When an action executes, all input tokens are consumed (since the data/control is currently used) and after the execution of the action, new tokens become available on its outputs. See the operations of `ActionExecution` for the respective rules. Tokens do not move on their own upon any element. They emit offers along the downstream paths and elements decide (by their respective semantics) whether they accept the offered token.

Differences to standard UML The class `Token` represents the term `Token` from the text of the UML specification.

B.5.5 Class `ControlToken`

Description A `ControlToken` represents a locus of control.

Package `ControlToken` is defined in the `Core Activities` package.

Associations (none)

Constraints (none)

Operations (none)

Semantics `ControlTokens` do not have semantics that differ from general tokens. They rather represent the standard case, whereas `ObjectTokens` have special features.

Differences to standard UML `ControlTokens` formalize the identical term from the text of the UML specification. They differ from the UML specification in that they can be temporarily stored in untyped `BufferNodes`, particularly in `InputPins` and `OutputPins`.

B.5.6 Class `ObjectToken`

Description An `ObjectToken` represents an object.

Package `ObjectToken` is defined in the `Core Activities` package.

Associations content [1] The object that is represented by the `ObjectToken`. Multiple `ObjectTokens` can represent the same object.

Constraints (none)

Operations (none)

Semantics An `ObjectToken` represents a piece of data that is routed through the activity graph.

Differences to standard UML `ObjectTokens` formalize the respective term from the text of the UML specification. In this case study we do not consider object tokens which represent not objects but rather single data values (the UML specification sometimes refers to this second interpretation of object tokens).

B.5.7 Class Offer

Description An `Offer` represents the fact that a token might move from its current position to the element where the offer resides. Offers are used to mark partially evaluated paths through control structures. They are passed along by the different nodes and edges which make up the control structures.

Package `Offer` is defined in the `Core Activities` package and extended by the `Controlnodes` package.

Associations

Token [1..*] The base token that is represented by the offer. If an offer is formed by joining several offers at a join node, an offer can have more than one base token. Multiple base tokens are necessarily control tokens.

ActivityElement [1] The `ActivityElement` that currently carries the offer. Both nodes and edges can carry an offer.

Node [0..*] If the offer has been spawned at a fork node, it retains a link to the queue of the outgoing edge it travels along. This queue is called its spawnpoint. An offer can have multiple spawnpoints if it is joined from multiple offers with different spawnpoints. Note that while this association is defined in the `Core Behavior` package, it is only used in connection with fork nodes (from the `Controlnodes` package).

Constraints

context `Offer`

inv:

```
this.base->count>2 implies this.base.OCLtype=Controltoken
this.spawnpoint.OCLtype=CentralBufferNode
```

If multiple base tokens exists, they have to be control tokens. The spawnpoint must always be a central buffer.

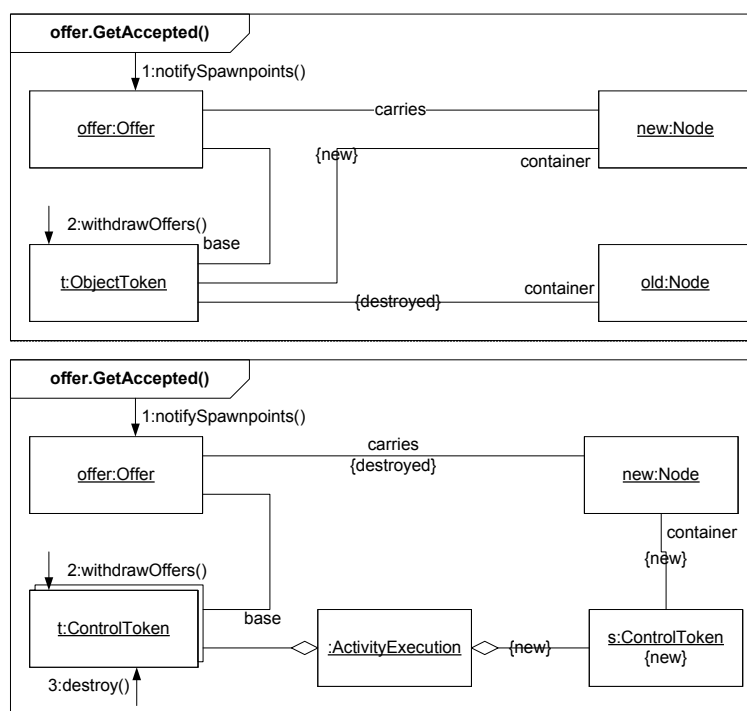


Figure B.16: DMM rules describing how an offer is being accepted

Operations

getAccepted() One of the most crucial behaviors in the token/offer mechanism is the acceptance of an offer. If an offer reaches a node that is able to store/process its underlying token, the base token moves to the accepting node. The rules in Fig. B.16 describe this behavior for control and object tokens respectively. In both cases, the remaining offers are withdrawn and potential spawnpoints are informed of the offer's acceptance. A significant difference is that offers can have only one object token as their base but several control tokens (due to joining). The joining of the underlying tokens is facilitated by deleting all base tokens and creating a new control token at the target node. No information loss occurs in this process as control tokens do not carry any information. An important property of both rules for **getAccepted** is that only a single token arrives at the accepting node, even though a number of tokens might depart from their source nodes. In the case of control tokens, the arriving token is newly created, i.e., it is not even part of the departing token set. This is important since other rules cannot match the token previous to the application of **getAccepted**.

destroy() The **destroy** operation is the standard destructor for offers and deletes an offer (cf. Fig. B.17).

notifySpawnpoints() When an offer is accepted, its spawnpoints need to be informed of this process as they need to enqueue copies of the underlying



Figure B.17: DMM rule for the destructor of Offer

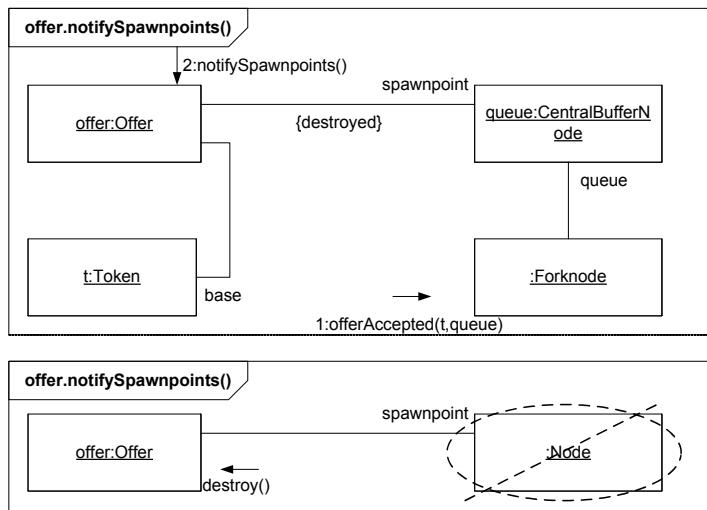


Figure B.18: DMM rule describing the notification of spawnpoints by an Offer

token (see. Fig. VI.6 for an illustration of this process). The rules in Fig. B.18 specify this process by a recursive loop. The upper rule applies to cases where a **spawnpoint** link is present. It triggers the enqueueing of token copies at the relevant fork node, deletes the spawnpoint link and class itself recursively. The bottom rule in the figure provides the recursion end which is reached if no more spawnpoint links need to be processed. this rule also forms the default case for offers which have not been spawned at all.

Semantics Offers represent partial evaluations. They get moved (flow) as further evaluation steps occur and get accepted whenever they reach a node which might process or store their base token. On acceptance an offer is deleted, the base token is moved, and all competing offers of this token are withdrawn. At join nodes several offers are fused into one. This is only possible for offers of control tokens or offers which represent the same base token.

Differences to standard UML Offers are a representation of the term offer used in the semantics description of the UML specification. They have been made more precise in that their meaning and possible operations on them has been clarified. In particular, we made the notion of acceptance more clear: if a node accepts an offer this implicates the movement of the underlying token(s). Offer acceptance is thus clearly distinguished from offer flows. The UML specification uses the term acceptance more freely and ambiguously. The concept of a spawnpoint is newly added to allow for an operationalization of the new semantics of fork nodes.

B.5.8 Class Node

Description Nodes are one type of element which forms an Activity.

Package Node is defined in the Core Activities package.

Associations

Edge [*] *as incoming* The edges which run into the node.

Edge [*] *as outgoing* The edges which run out of the node.

Constraints (none)

Operations (none)

Semantics Nodes in general have no dynamic semantics. The different subclasses add dynamic semantics for specific types of nodes.

Differences to standard UML No changes.

B.5.9 Class Edge

Description Activity edges are directed connection between the nodes of the activity. They determine the way tokens/offers can move between the nodes.

Package Edge is defined in the Core Activities package.

Associations

Node (*as target*) [1] The node which the edge is targeting. Tokens can move over the edge *to* this node.

Node (*as source*) [1] The node where the edge is originating. Tokens can move over the edge *from* this node.

ValueSpecification [0..1] An edge can be adorned with a guard condition.

Constraints (none)

Operations

$P_canCarry(o:Offer)$, $P_canCarry(t:Token)$ The operation $P_canCarry$ is a predicate that tests whether an offered token meets the requirements of the edge. Since edges can have guards, it needs to be checked whether an offered token meets the requirements of the guard. The two upper hand rules in Fig. B.19 depict this situation. They pass the token under consideration on to the guard which has to evaluate it in a positive way for the rule to succeed. Two rules are necessary since for reasons of convenience the operation takes either the token directly as its input parameter or an offer based on this token. If no guard is present (lower hand rules), the only condition to be met is that no other offer is currently occupying the edge.

$P_hasOffer()$ The predicate $P_hasOffer$ (cf. Fig. B.20) checks whether the edge in question does currently hold an offer.

Semantics Edges guide the token flow. They do not play an active role in this but rather contain offers which are moved according to the rules specified for the various node types. Edges can only carry one offer at a time.

Differences to standard UML The UML syntax distinguishes between edges which can be traversed only by object tokens (so called **ObjectFlows**) and edges which can only be traversed by control tokens (**ControlFlows**). By using this distinction, the correct typing of control structures can be checked. Semantically, however, there is no difference in how the different kinds of edges treat

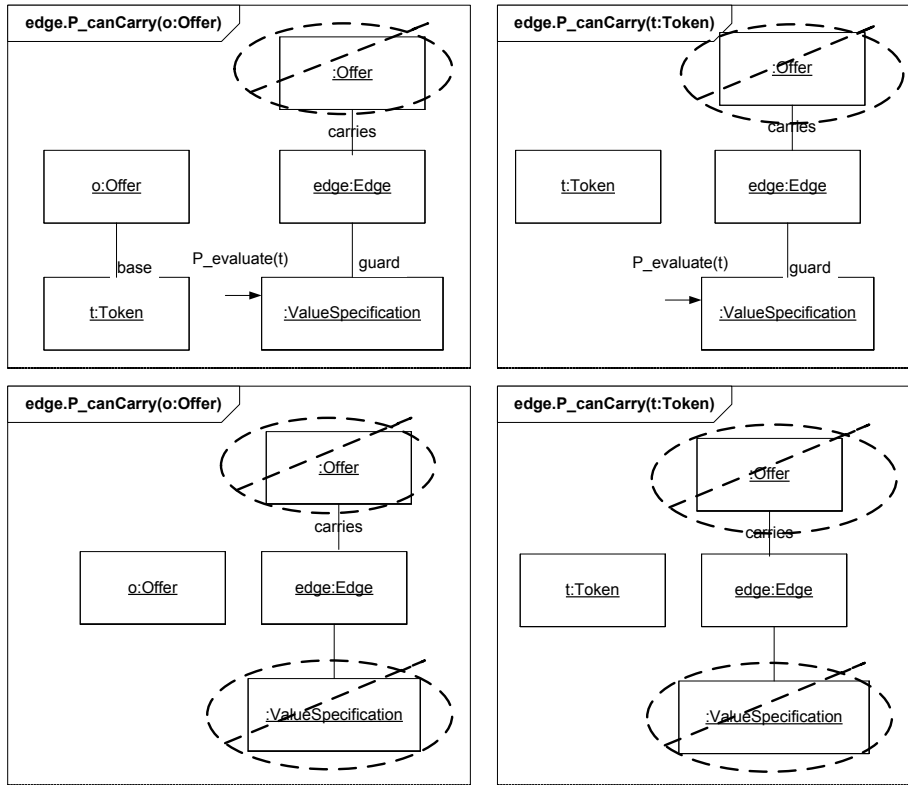


Figure B.19: DMM rules describing how edges determine whether they can carry an offer

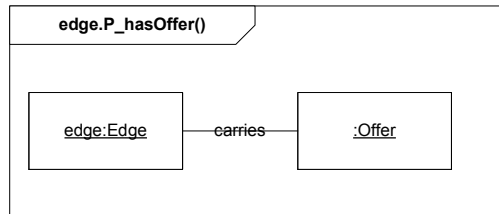


Figure B.20: DMM predicate rule for determining whether an edge carries an offer

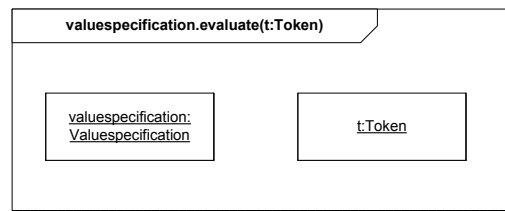


Figure B.21: DMM rule for evaluating a ValueSpecification

the passing tokens. We do thus integrate both concepts into a single class in this formalization. There are furthermore advanced elements which can adorn an edge to express a modification of tokens and the passing of token groups. These elements are not regarded in this case study.

B.5.10 Class ValueSpecification

Description A ValueSpecification contains a specification which yields a truth value when being evaluated.

Associations

Edge[1] In this domain, ValueSpecifications only occur as guards attached to edges.

Constraints (none)

Operations

evaluate(t:Token) The single operation of this class is `evaluate()`. Evaluating a ValueSpecification yields a boolean truth value, with respect to the token under consideration. Since UML does not provide any language for actually writing down ValueSpecifications the rule in Fig. B.21) is the identity rule and thus always returns successfully. Since DMM does not support explicit return values, failure (i.e., evaluation to false) would be modeled by not finding a correct rule application for `evaluate` and thus creating an error situation.

Semantics ValueSpecifications always evaluate to true in our formalization.

Differences to standard UML UML provides the element ValueSpecification without any concrete syntax. We have added ValueSpecification for means of completeness but have fixed its semantics to always be true. If a notation for the actual expression of ValueSpecifications is standardized, the rule for `evaluate` needs to be adapted.

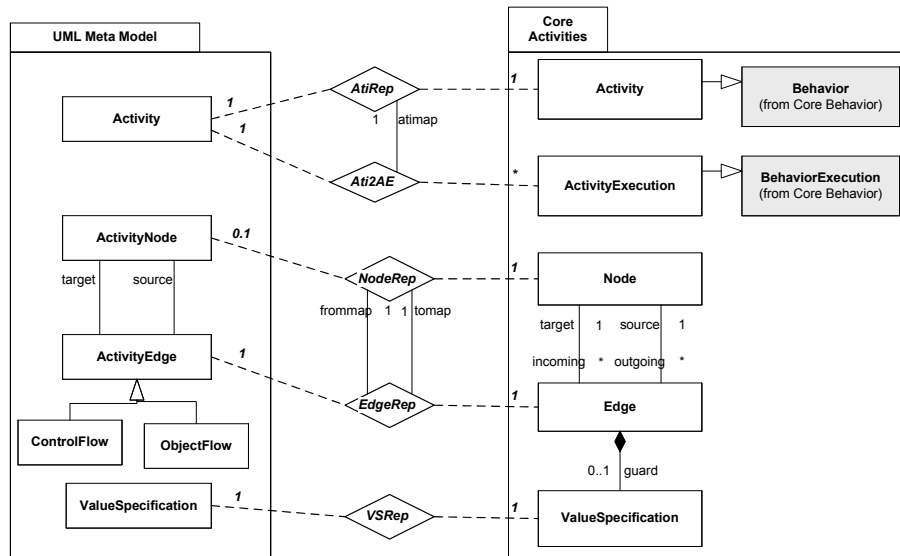


Figure B.22: Semantic mapping appings of the Core Activities package

B.5.11 Mappings

The mappings of the Core Activities package are depicted in Fig. B.22.

AtiRep - The Activity Replication Relation The Activity Replication Relation has no intrinsic details which haven't been addressed by the more general Behavior Replication Relation. It only serves to ensure that Activities are indeed mapped to Activities and to no other form of behavior.

NodeRep - The Node Replication Relation

```
context NodeRep
domain=scope.domelement.node
range=scope.ranelement.Node
inv::
  domelement.name=ranelement.name
```

Note that the syntactic end of this Relations allows for Nodes without a corresponding ActivityNode. This is necessary to account for differences between the semantic and the syntactic domain (treatment of control inputs and implementation of fork behavior).

EdgeRep - The Edge Replication Relation

```
context EdgeRep
domain=scope.domelement.edge
range=scope.ranelement.Edge
inv:
  frommap.domelement=self.domelement.source
```

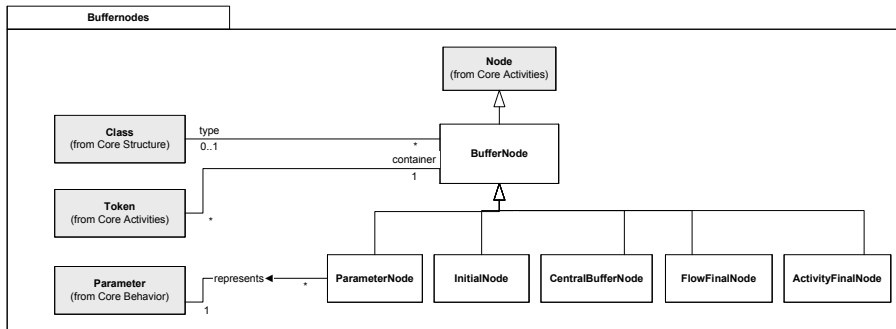


Figure B.23: The contents of the Buffernodes package

```

frommap.ranelement=(self.ranelement.source)or
                    (self.ranelement.source.Forknode)or
                    (self.ranelement.source.Action)
tomap.domelement=self.domelement.target
tomap.ranelement=(self.ranelement.target) or
                    (self.ranelement.target.Action)
  
```

The invariants of the Edge Replication Relation address the differences between syntactic and semantic domain meta model. While each edge gets translated, in some cases the endpoint of the replicated edge are modified to suit the semantic domain better. In particular for outgoing edges of forknodes, the introduction of queues is allowed and for edges attached to Actions, the introduction of (control-valued) pins is allowed. See the InRep, OutRep and ForkMod Relation for details. Note that the specific edge types Controlflow and ObjectFlow are not directly matched. All edges are thus unified in their semantic interpretation.

VSRep - The ValueSpecification Replication Relation

```

context VSREp
inv:
  domelement.expression = ranelement.expression
  
```

The replication of a ValueSpecification ensures that its defining expression gets replicated to the semantic domain.

B.6 Package Buffernodes

Description The Buffernodes package contains the definition of the BufferNode class and its subclasses (cf. Fig. B.23). Since buffernodes can be typed, the package Core Structure is merged as well as the packages Core Behavior and Core Activities. Note that due to their close semantic entanglement with actions the classes InputPin and OutputPin are defined in the Core Actions package.

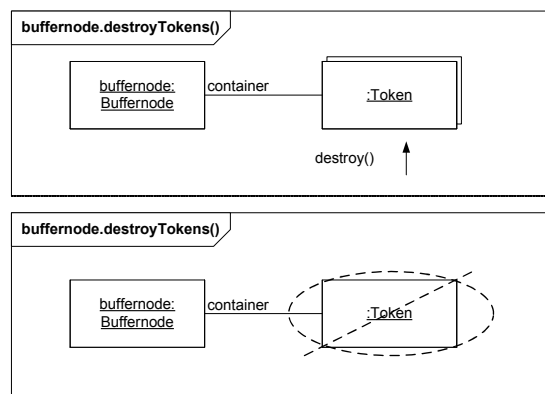


Figure B.24: DMM rule for destroying all tokens in a node

B.6.1 Class Buffernode

Description BufferNodes are nodes that can buffer tokens.

Package BufferNode is defined in the Buffernodes package.

Associations

Class [0..1] indicates the class that determines the type of the buffer node.

Token [*] the tokens which are currently contained by the node.

Constraints

```
context Buffernode
inv:
  not(self.type.isEmpty()) implies
    (self.token.object.type->includes(self.type))
```

If a buffernode is typed, only object tokens with objects of the correct type may be contained within it.

Operations

`destroyTokens()` Upon the termination of an activity execution (e.g., by activation of an `ActivityFinalNode`) a node can be requested to destroy all of its tokens. This is facilitated by calling the destructor operation of all tokens which reside in the node (cf. Fig. B.24).

Semantics A `BufferNode` is an abstract superclass. Common to all its subclasses is that they can store tokens (either object or control tokens). The rules for accepting and releasing these tokens are special to each subclass. Typed `BufferNodes` can only store `ObjectTokens` referring to objects of the indicated

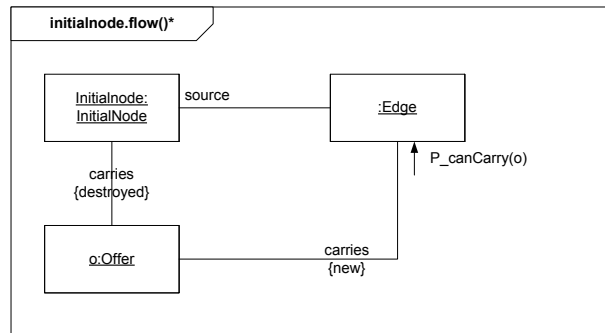


Figure B.25: DMM rule describing the flow of offers from an initial node

type (or one of its subclasses). Untyped buffer nodes can store any kind of token (including control tokens). No special behavior is prescribed by this superclass apart from the destruction operation.

Differences to standard UML The `BufferNode` extends the notion of UML's `ObjectNode` in that it allows the buffering of `ControlTokens`. The different name hints at this more general definition. Note also that our semantic domain meta model does not provide a special class to group non-buffering nodes (i.e., we have no counterpart to UML's control node).

B.6.2 Class `InitialNode`

Description Initial nodes are supplied with control tokens if an activity starts.

Package `InitialNode` is defined in the `Controlnodes` package.

Associations (none)

Constraints

```
context InitialNode
inv:
  not(this.token->isEmpty()) implies this.token.OCLtype=Controltoken
  self.type->isNull()
```

An initial node can only contain control tokens, it is thus untyped

Operations

`flow()*` This operation describes how offers flow from the initial node into the activity graph (cf. the rule in Fig. B.25). Note that if multiple edges leave the initial node, one of them is chosen non-deterministically.

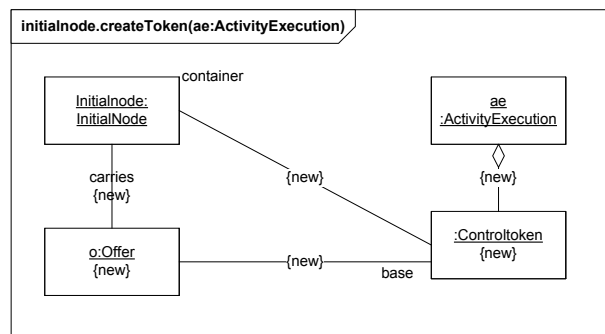


Figure B.26: DMM rule describing creation of tokens on an initial node

`createToken(ae:ActivityExecution)` The rule shown in Fig. B.26 describes the process of creating a new control token and a corresponding offer at an initial node. This rule is invoked once for each initial node when an activity execution starts. The `ActivityExecution` passed as the parameter forms the context for the newly created token.

Semantics Initial nodes emit their offer into the activity graph, thereby initializing the behavior expressed by the activity.

Differences to standard UML No changes.

B.6.3 Class ParameterNode

Description A `ParameterNode` holds the tokens for objects which are passed as parameters to the activity or which leave the activity as parameters.

Package `ParameterNode` is defined in the `Buffernodes` package.

Associations

Parameter [1] The parameter which the `ParameterNode` represents.

Constraints

```
context ParameterNode
inv:
  not(self.type->isEmpty())
  self.type=self.Parameter.type
```

`ParameterNodes` are always be typed. The type of a parameter node must be the same as the type of the parameter it represents.

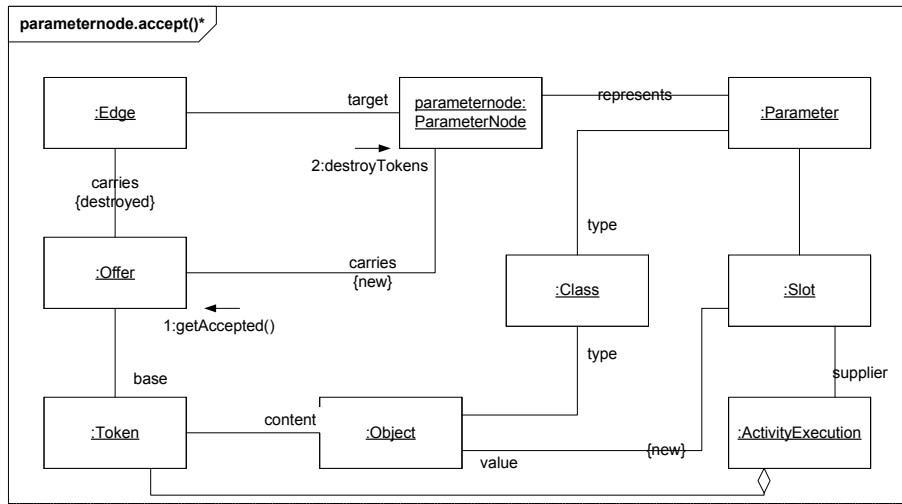


Figure B.27: DMM rule describing the acceptance of offers on a parameter node

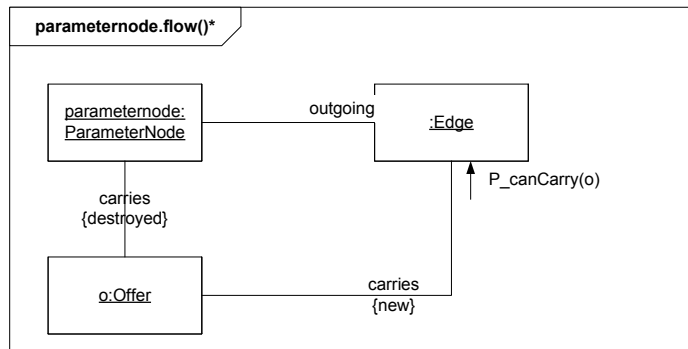


Figure B.28: DMM rule describing the flow of offers from a parameter node

Operations

accept()* A parameter node will accept offers if their tokens have the correct type. The underlying object is passed to the corresponding slot (cf. Fig. B.27) and the token is being deleted as it has fulfilled its purpose.

flow()* Offers can flow out of a parameter node if an outgoing edge can carry them. The rule in Fig. B.28 implements this operation.

createToken(ae:ActivityExecution) The rule shown in the top half of Fig. B.29 describes the process of creating a new object token representing the value of the parameter and a corresponding offer at a **ParameterNode**. The new token is furthermore registered in the activity execution to which it belongs. For **ParameterNodes** which represent a result of an activity, the operation **createToken** is not useful, thus the lower half of Fig. B.29 provides a rule which amounts to a no-op in this case. Curiously, while UML distinguishes different types of tokens, edges, and nodes by subclassing, the

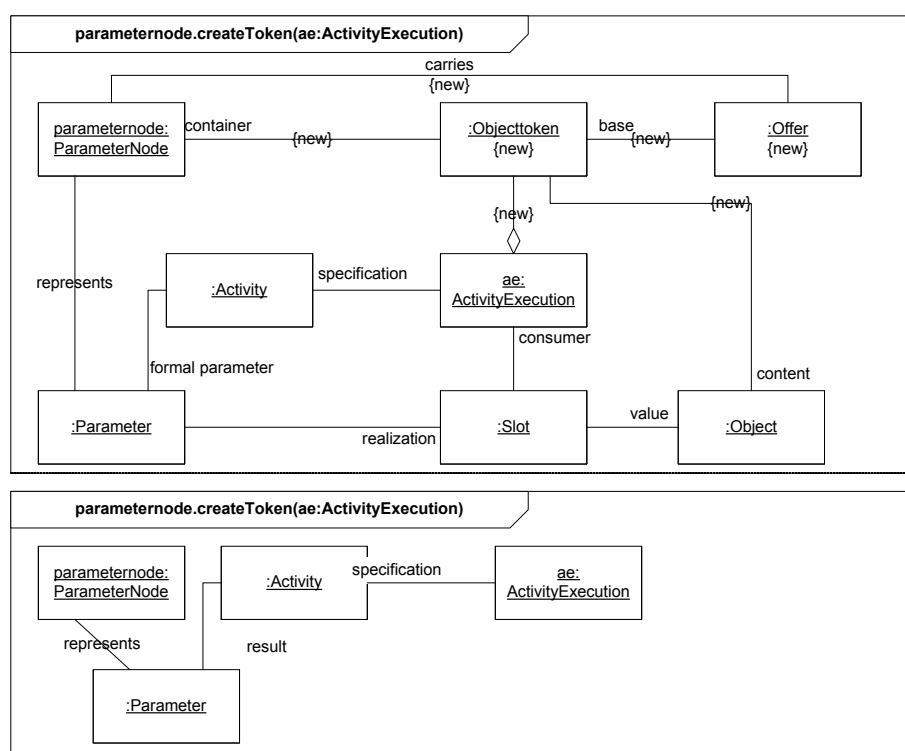


Figure B.29: DMM rules describing creation of tokens on a parameter node

different kinds of `ParameterNode` are distinguished by association only⁴. We stuck to the UML design, although it enforces this kind of "empty" rule.

Semantics `ParameterNodes` capture the data being passed at the invocation of an activity as parameters. They emit offers into the activity graph to initialize the behavior described therein. At the end of this behavior they store the data that is to be passed out of the behavior in a slot for a parameter.

Differences to standard UML `ParameterNodes` formalize the semantics of the UML metaclass `ActivityParameterNode`.

B.6.4 Class `CentralBufferNode`

Description `CentralBufferNodes` (CBNs for short) are buffer nodes which occur explicitly in the activity graph. CBNs can store multiple tokens (typically object tokens) and order them according to specified schemes (FIFO, LIFO). CBNs also serve as queues for forknodes.

Package `CentralBufferNode` is defined in the `BufferNodes` package.

Associations

Offer [*] The offer for which the CBN is a spawnpoint.

Forknode [0..1] CBNs can be used to model the queues of fork nodes. In this case they are aggregated with the forknode and sit between the fork node and the outgoing edges.

Token [*] The token(s) which are currently buffered in the CBN.

Constraints

context `CentralBufferNode`

inv:

```
exists(o:Offer|o.spawnpoint=self) implies exists(f:ForkNode|f.queue=self)
exists(f:ForkNode|f.queue=self) implies (self.OrderingScheme=FIFO) and
(self.type->isEmpty())
```

CBNs can only be spawnpoints if they act as queues. If a CBN acts as a queue, it can only have FIFO ordering and it is untyped.

Operations

`accept()*` There are two scenarios in which a central buffer node accepts an offer that is carried by an incoming edge: Either the CBN is typed (left hand

⁴A fact which is furthermore inconsistent with the handling of parameters in the UML 2 Infrastructure, cf. [FTF], issue 7343

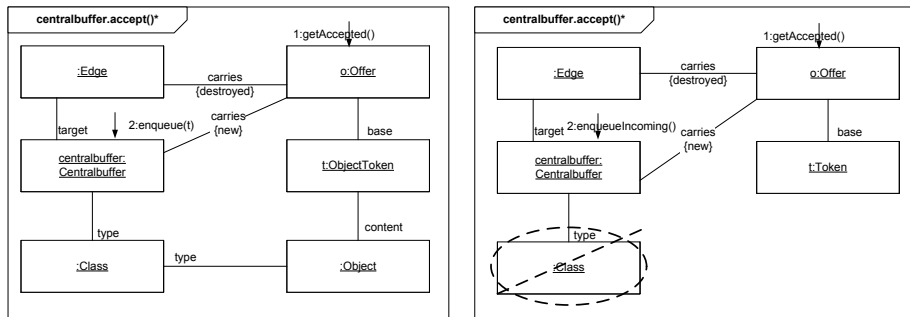


Figure B.30: DMM rules describing the acceptance of offers on typed and untyped central buffer nodes

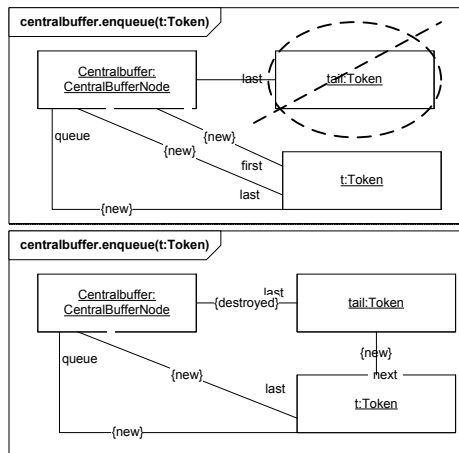


Figure B.31: DMM rules describing the enqueueing of arriving tokens in a central buffer node

rule of Fig. B.30) then it will only accept offers for object tokens of the correct type, or it is untyped (right hand rule of Fig. B.30) and will accept any offer.

enqueue(t:Token) The operation enqueue is described by the two rules in Fig. B.31. The upper rule covers the case of an empty queue and results in the creation of the first and last links to the enqueued token. The lower rule describes enqueueing a token in a non-empty queue and comprises the redirection of the last link as well as the linking of the list elements.

dequeue()* The dequeuing operation is strongly dependent on the ordering scheme which has been specified for the node. In Fig. B.32 we provide the rules for the dequeuing under a FIFO scheme. Dequeuing can either happen on a list with at least two elements (upper rule of Fig. B.32) or it can empty the list (lower rule). Dequeuing will only occur if an outgoing edge can carry a newly created offer for the dequeued token. Note that as each rule checks for the non-existence of already offered tokens, a cen-

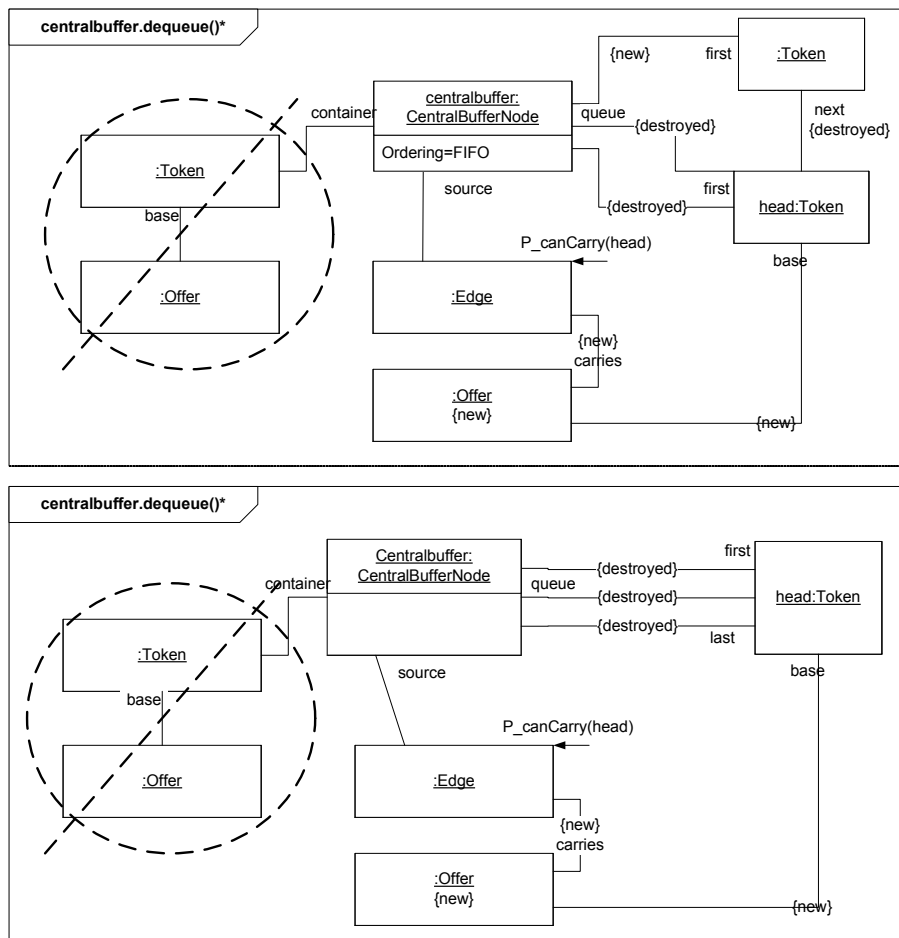


Figure B.32: DMM rule describing the dequeuing of tokens in a central buffer node

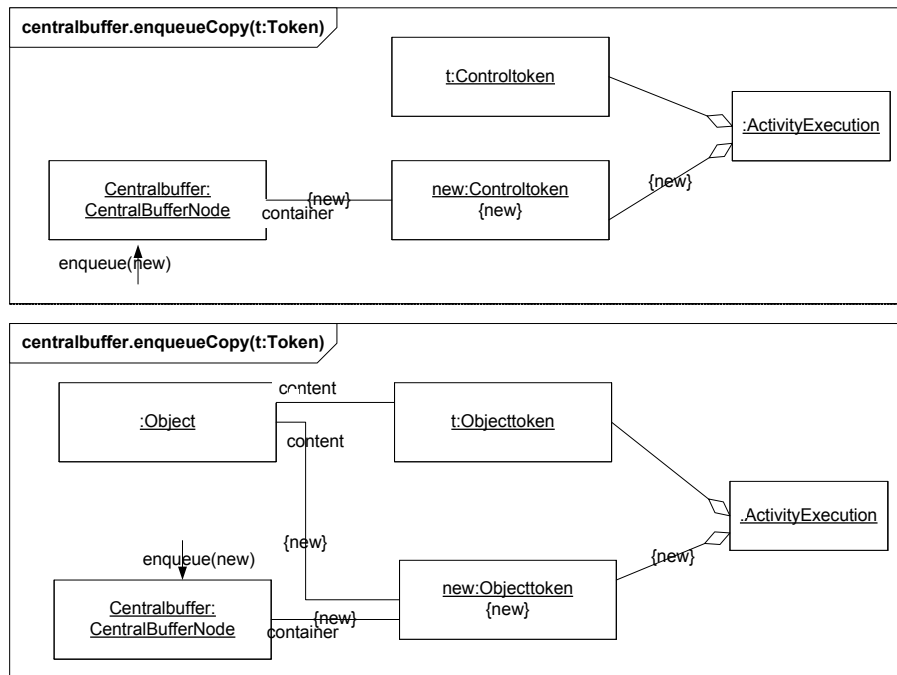


Figure B.33: DMM rules specifying the duplication of tokens on a central buffer node

tral buffer node will only release a token from the queue if the previously dequeued one has moved on.

enqueueCopy(t:Token) The operation `enqueueCopy` duplicates the token passed as the parameter (see Fig. B.33) and places this copy in the queue. Two rules are necessary to describe this operation as control tokens and object tokens need different handling.

enqueueincoming() The operation `enqueueincoming` (see Fig. B.34) Has a special purpose. As the acceptance of a control token will result in a newly created token, it cannot be passed as a parameter between `buffernode.accept` and `buffernode.enqueue`. Rather, this operation determines the newly arrived token (already contained in the buffernode but not yet queued) and passes it on to the `enqueue` operation.

Semantics `CentralBufferNodes` are used to temporarily store tokens. Arriving offer will be accepted (if they are conformant to the type of the CBN). Limited capacity buffers are currently not supported. The semantic domain provided here does support FIFO and LIFO ordering schemes; other schemes (e.g., priority-based ones) might require additional structures to maintain that specific order. If directly attached to a fork node, CBNs work as queues which can store the tokens for outgoing edges in the case that an offer is accepted on only one of the outgoing paths.

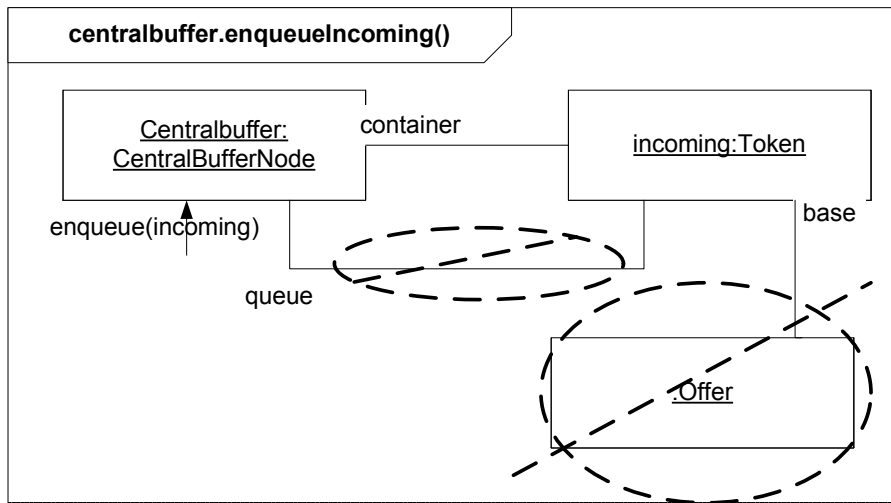


Figure B.34: DMM rules specifying the duplication of tokens on a central buffer node

Differences to standard UML CBNs support the basic operations of the corresponding type in UML. Not supported are exotic ordering schemes⁵, capacity limits and transformation behaviors.

B.6.5 Class FlowFinalNode

Description The FlowFinalNode accepts every offered token and destroys it, thereby terminating a single flow of control in the activity.

Package FlowFinalNode is defined in the Buffernodes package.

Associations (none)

Constraints (none)

Operations

accept()* The operation `accept()` as specified by the rule in Fig. B.35 accepts each offer that is provided by an incoming edge and proceeds to destroy it. The destruction itself has to be separated into another operation since acceptance might create a new (control) token on the final node which cannot be matched by this rule

⁵The UML specification hints at possible priority-based ordering but does not provide any details on it

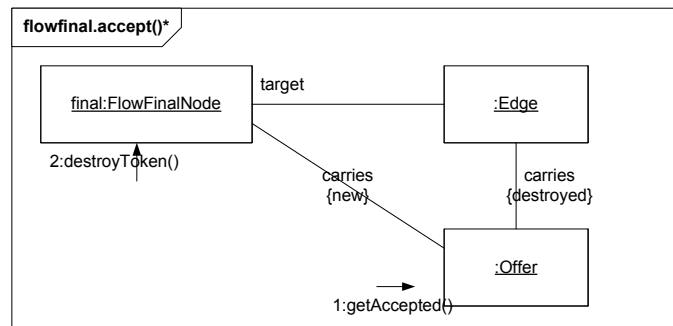


Figure B.35: DMM rule describing the acceptance of an offer by a flow final node

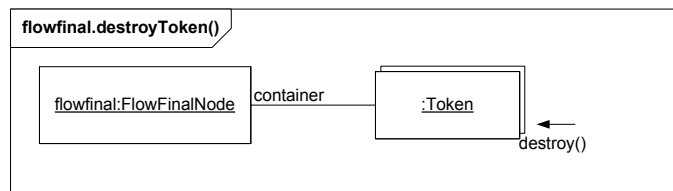


Figure B.36: DMM rule for destroying tokens at a flow final node

`destroyToken()` To destroy a token, the rule in Fig. B.36 simply calls the `destroy` operation of the accepted token.

Semantics Flow finals are used to end single threads of control in an activity.

Differences to standard UML In UML syntax flow final nodes are control nodes. In this semantic domain meta model we model them as `BufferNodes`. The reason for this difference is that flow final nodes have to accept offers to obtain their tokens. Thus, the token moves to the flow final node and briefly rests there until it is destroyed.

B.6.6 Class `ActivityFinalNode`

Description Upon receiving a token, activity final nodes terminate the execution of the whole activity instantly.

Package `ActivityFinal` is defined in the `Buffernodes` package.

Associations (none)

Constraints (none)

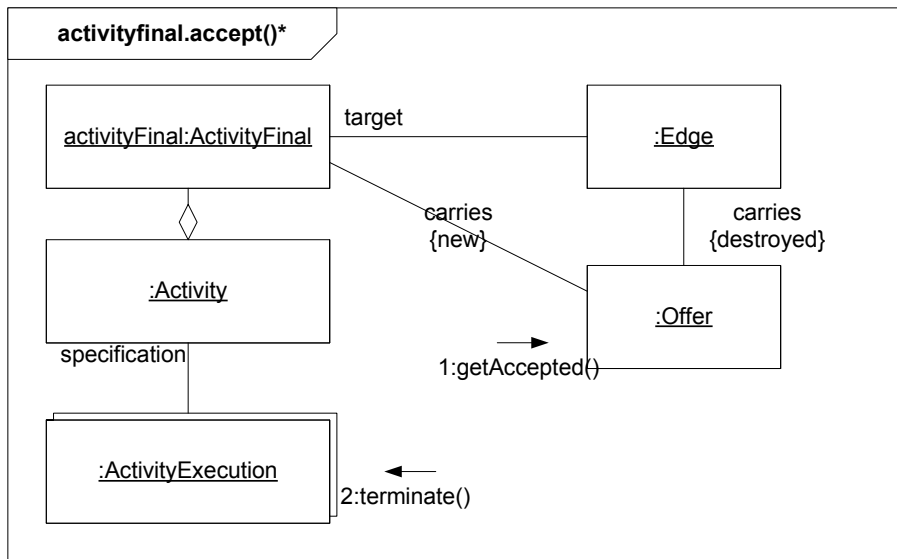


Figure B.37: DMM rule describing the acceptance of an offer by an ActivityFinalNode and the subsequent termination of all activity executions.

Operations

`accept()*` The operation `accept()` as specified by the rule in Fig. B.37 accepts an offered token and instantly triggers the termination of all executions of the current activity. Note that activity final nodes are not restricted to destroying just the execution they are part of. The UML specification states clearly: "A token reaching an activity final node terminates the activity. In particular, it stops all executing actions in the activity, and destroys all tokens in object nodes, except in the output activity parameter nodes" Note that in our formalization, object tokens arriving on parameter nodes directly pass on their information to the corresponding slots and the definition of `CallBehaviorActionExecution.end` creates the proper tokens for these results. If not all required results have been produced when the activity final node terminates the activity, an error situation occurs.

Semantics `ActivityFinalNodes` accept all tokens offered to them. Upon this acceptance they trigger a termination of all execution of the activity they reside in. This includes the termination of all invoked behaviors.

Differences to standard UML Like flow final nodes, activity final nodes are control nodes in UML but buffer nodes in our formalization. While there would be no semantic difference in terminating the activity without accepting the offer beforehand (the token is destroyed by the termination operation anyway), we modeled the semantics this way to be more consistent with the semantics of flow final nodes.

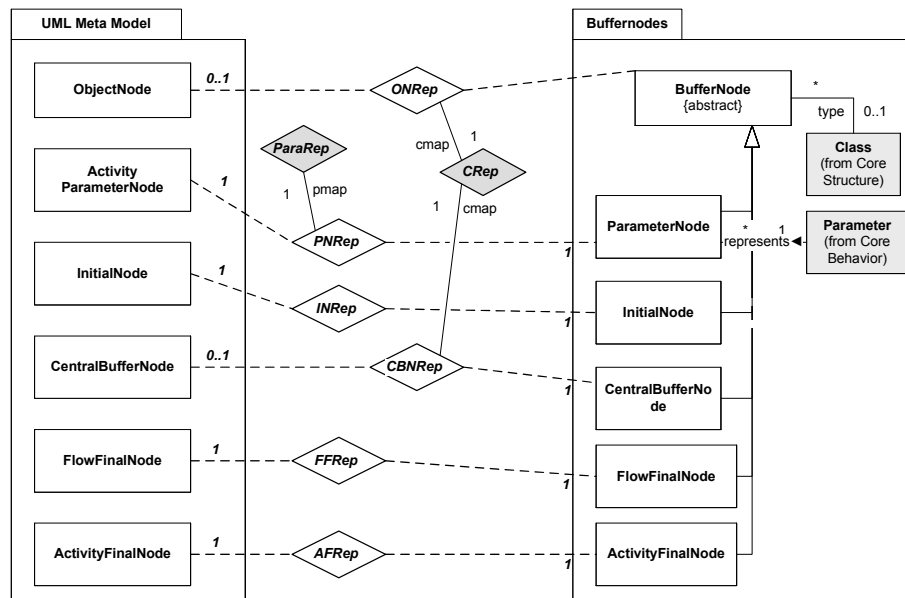


Figure B.38: Semantic mappings for the Buffernodes package

B.6.7 Mappings

The semantic mappings targeting the elements of the Buffernodes package are depicted in Fig. B.38.

ONRep - The ObjectNode Replication Relation

```
context ONRep
inv:
  cmap.domelement=self.domelement.type
  cmap.ranelement=self.ranelement.type
```

The ONRep mapping elicits the renaming that occurs to reflect the changes nature of Buffer nodes. This is also reflected by the cardinality at the syntax side which allows for BufferNodes which do not correspond to ObjectNodes. The constraints ensure that the typing of the node is preserved by the replication.

PNRep - The ParameterNode Replication Relation

```
context PNRep
inv:
  pmap.domelement=self.domelement.parameter
  pmap.ranelement=self.ranelement.Parameter
  domelement.name=ranelement.name
```

The ParameterNode replication references the replication of Parameters to ensure that the mapping respects the mapping of the underlying parameter.

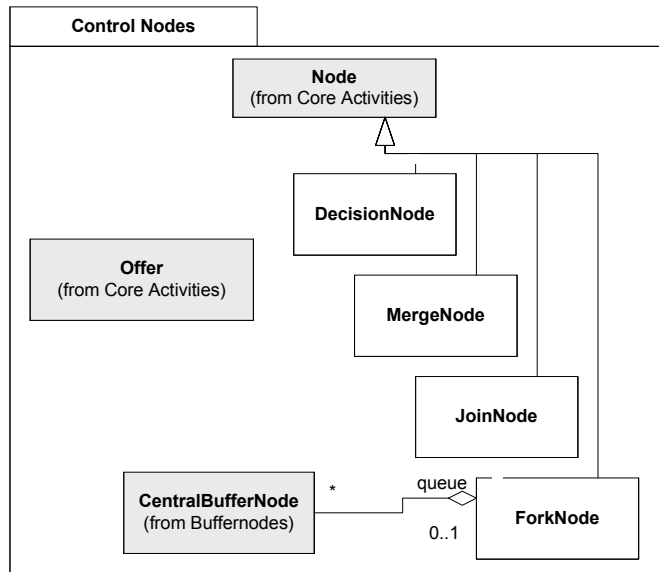


Figure B.39: The contents of the Controlnodes package

INRep - The InitialNode Replication Relation The InitialNode replication does not have any details.

CBNRep - The CentralBufferNode Replication Relation

```
context PNRep
inv:
  cmap.domelement=self.domelement.type
  cmap.ranelement=self.ranelement.type
  domelement.name=ranelement.name
```

The CBN replication has to respect the underlying type mapping of the CRep Relation. Note that not all semantic CBNs have a corresponding syntactic CBN.

FFRep - The FlowFinalNode Replication Relation FinalFlowNodes are replicated without additional details.

AFRep - The ActivityFinalNode Replication Relation ActivityFlowNodes are replicated without additional details.

B.7 Package Controlnodes

Description The Controlnodes package contains the definition of controlnodes in the activity diagram (Fig. B.39), i.e., all nodes which cannot contain a token. Note that due to the specific semantics of the fork node, this package is dependent upon the Buffernode package.

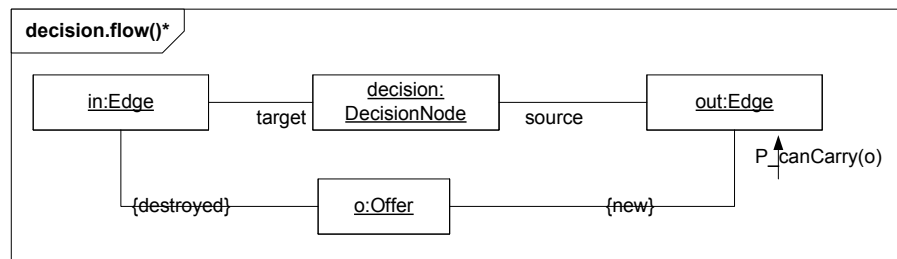


Figure B.40: DMM rule describing the flow of offers over a decision node

B.7.1 Class DecisionNode

Description At decision nodes an incoming flow can be directed into one of multiple alternative outgoing flows.

Package DecisionNode is defined in the Controlnodes package.

Associations (none)

Constraints (none)

Operations `flow()*` The operation flow (cf. Fig. B.40) checks whether the incoming edge holds an offer that an outgoing edge might carry and proceeds to move the offer to that edge.

Semantics Decisions guide the flow of offers according to the conditions on their outgoing edges. Note that if several outgoing edges might carry an offer (i.e., their guard conditions are not disjoint), the decision node will choose one of these edges nondeterministically.

B.7.2 Class MergeNode

Description At merge nodes offers from multiple incoming flows are directed into the single outgoing flow.

Package MergeNode is defined in the Controlnodes package.

Associations (none)

Constraints (none)

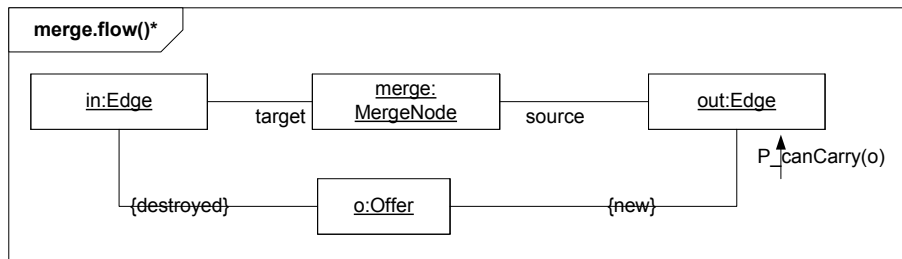


Figure B.41: DMM rule describing the flow of offers over a merge node

Operations `flow()*` The operation `flow()` (cf. Fig. B.41) checks whether one of the incoming edges holds an offer that the outgoing edge might carry and proceeds to move the offer to that edge.

Semantics `MergeNodes` are used to integrate alternative flows into a single outgoing flow. Therefore they pass on all incoming offers (provided these offers meet the requirements of the outgoing edge). If multiple incoming edges provide offers, the merge node will pass on all of these offers sequentially in a nondeterministic order.

B.7.3 Class `ForkNode`

Description Fork nodes split an incoming flow into multiple concurrent outgoing flows.

Package `ForkNode` is defined in the `ControlNodes` package.

Associations

`queue [*]` The buffernodes that queue up tokens for non-successful edges. Note that these queues connect the fork to its outgoing edges, i.e., a fork node does not have outgoing edges directly attached to it.

Constraints

```

context ForkNode
inv:
  self.outgoing->isEmpty()
  
```

Forks cannot have outgoing edges, these are all handled via its queues.

Operations

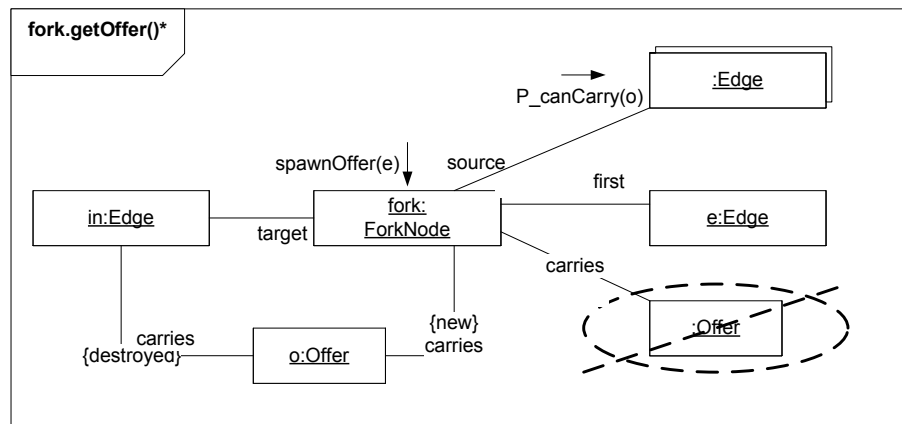


Figure B.42: DMM rule describing how a fork node acquires an offer from an incoming edge

getOffer()* The operation `getOffer` takes an offer from the incoming edge (cf. Fig. B.42) and triggers the mechanism of distributing spawns of this offer over the outgoing edges. The rule has the precondition that all outgoing flows must be able to carry the incoming offer.

spawnOffer(o:offer,e:Edge) The operation `spawn offer` creates a new copy of the offer indicated by its first parameter and places this new spawn on the edge indicated by the second parameter. To actually copy the offer, the newly created offer must have the same base token as the original one. The new offer must furthermore retain information about its spawnpoint by keeping a link to the corresponding queue. The two rules in Fig. B.43 differ in that the upper rule includes a recursive call of the operations while the lower rule forms the recursion end, if all edges have been supplied with spawns of the offer. The original incoming offer is then destroyed.

offerAccepted(t:token,exclude:Buffernode) if an offer spawned at a fork node is being accepted, there must be copies of the successful token enqueued for all other outgoing edges. The rule in Fig. B.44 performs this task by triggering the enqueueing of token copies on all of its queues, excluding only the queue for the already successful edge.

Semantics Forks perform a concurrent split of the incoming flow. This split has to be realized for offers and tokens in different ways: Offers will be picked from the incoming edge and copied to all outgoing edges. All of these spawned offers will retain information about their point of origin. This information is necessary because if one (or more) of the offers originating in a fork get accepted, the unsuccessful edges are provided with copies of the token. These copies are stored in the queues of the fork node and emit fresh offers which can in turn be accepted.

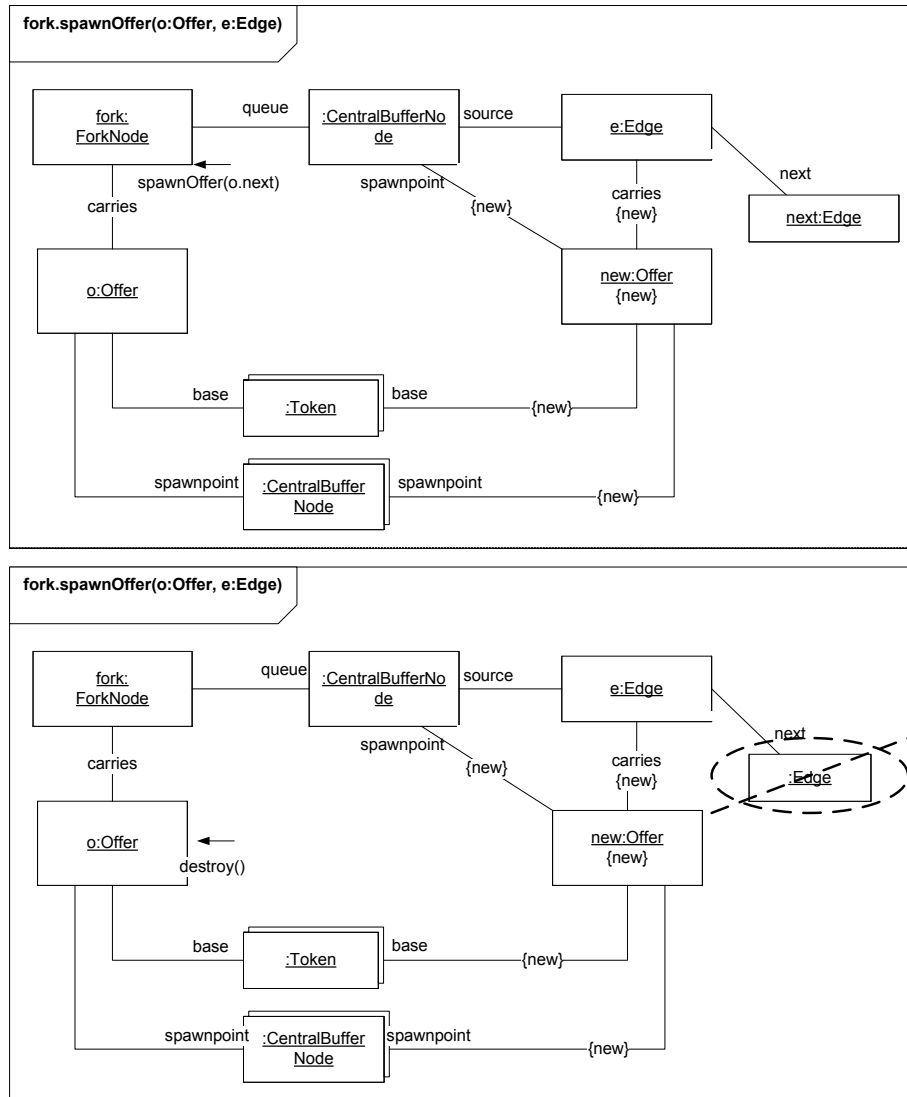


Figure B.43: DMM rule describing how a fork node spawns new offers

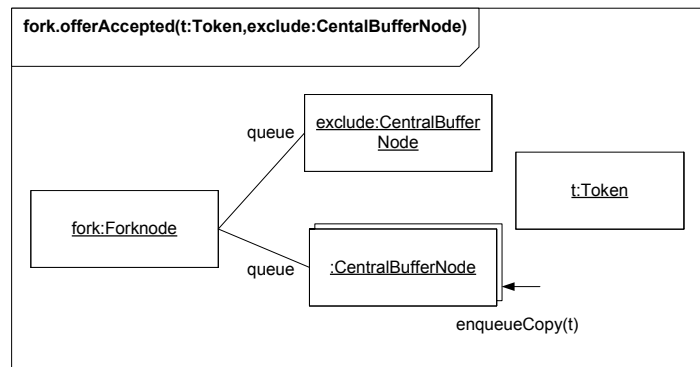


Figure B.44: DMM rule describing the information of a fork node of the fact that an offer spawned by it has been accepted

Differences to standard UML The semantics of fork nodes represented here correspond to the finalization changes suggested by Conrad Bock based upon our input (see Sect. VI.3 for a discussion). We have however excluded the case of outgoing arcs being blocked by unfulfilled guard conditions. Combining forks and guards makes for clumsy semantics and it can easily be avoided by inserting an extra decision node downstream from the fork.

B.7.4 Class JoinNode

Description A join node fuses offers from all incoming edges into a single outgoing offer.

Package JoinNode is defined in the Controlnodes package.

Associations (none)

Constraints

```

context JoinNode
inv:
  self.incoming=self.selfAllSuccs)
  
```

All incoming edges of a join node must be ordered.

Operations

flowin()* The Operation `flowIn` (see the rule in Fig. B.45) triggers the collection of offers under the condition that no such process is currently running and that all incoming edges carry an offer.

collectOffers() / collectOffer (in:Edge) The collection of offers from the incoming edges is a sequential process which works along the ordered incoming

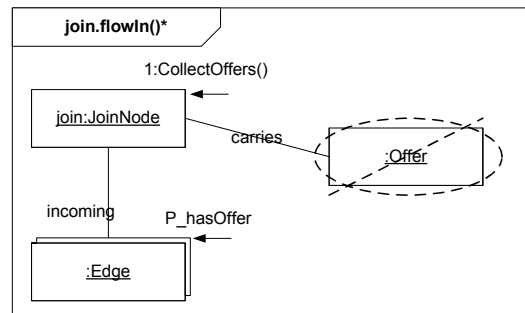


Figure B.45: DMM rule describing how a join node starts acquiring offers

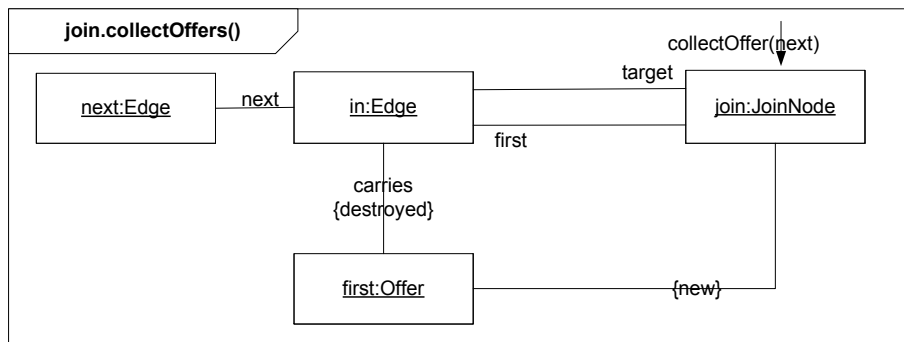


Figure B.46: DMM rules describing how a join node starts collecting offers from incoming edges

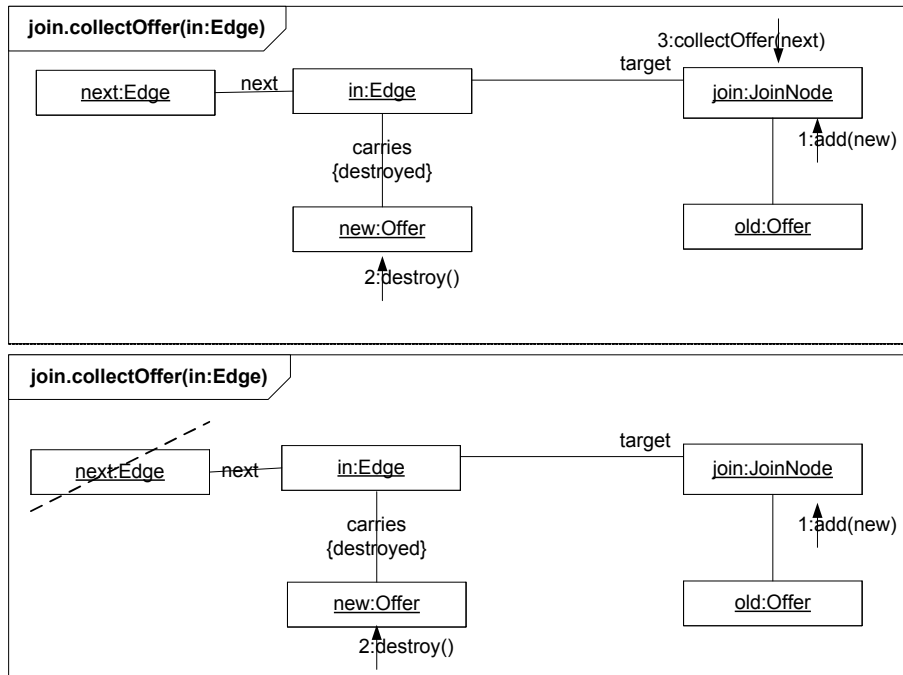


Figure B.47: DMM rules describing how a join node sequentially collects offers from incoming edges

edges. The operation `collectOffers` (Fig. B.46) starts the process by moving the offer from the first edge to the join node. The operation `collectOffer(in:Edge)` (Fig. B.47) continues this process by processing the succeeding edges. Offers from these edges are added to the existing offer, resulting in a fusion of the information contained in the offers.

`add(new:Offer)` The `add` operation takes an offer as parameter. It adds the information contained in this offer to the offer already residing in the join node. Two rules describe this operation. In general, offers for the same token can be merged. The lower hand rule in Fig. B.48 describes this case. Here, the information about the spawnpoints of the new token is copied to the existing token. This rule works for control and object tokens. For control tokens there is also the option of adding offers being based on different tokens (since control tokens do not carry any information). The upper hand rule in Fig. B.48 specifies this case.

`flowout()` After all offers have been collected from the incoming edges, the newly formed offer can move downstream. The rule (Fig. B.49) is decoupled from the actual joining process to allow for temporary blocks on the outgoing edge to clear.

Semantics Join nodes combine different incoming flows by fusing several offers into one. The resulting offer carries the combined information of the original tokens. Since control tokens do not have identity or carry information, control

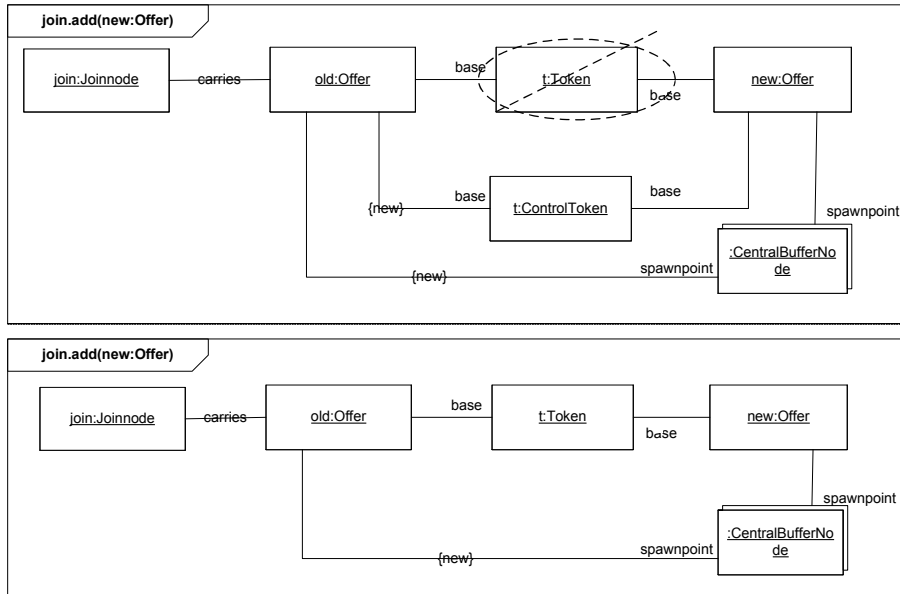


Figure B.48: DMM rules describing how a join node adds the information of a new offer to an already existing offer

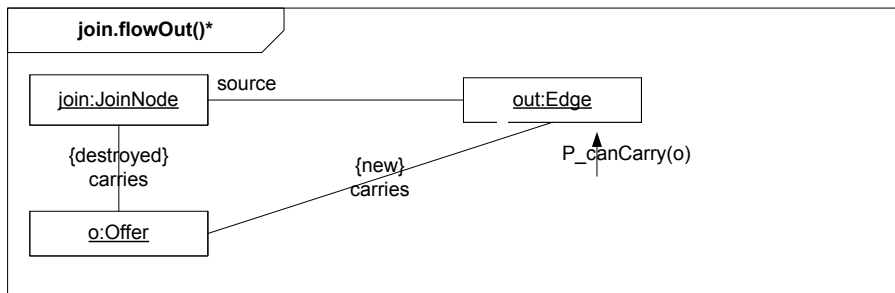


Figure B.49: DMM rule describing the passing of an offer out of a joinnode

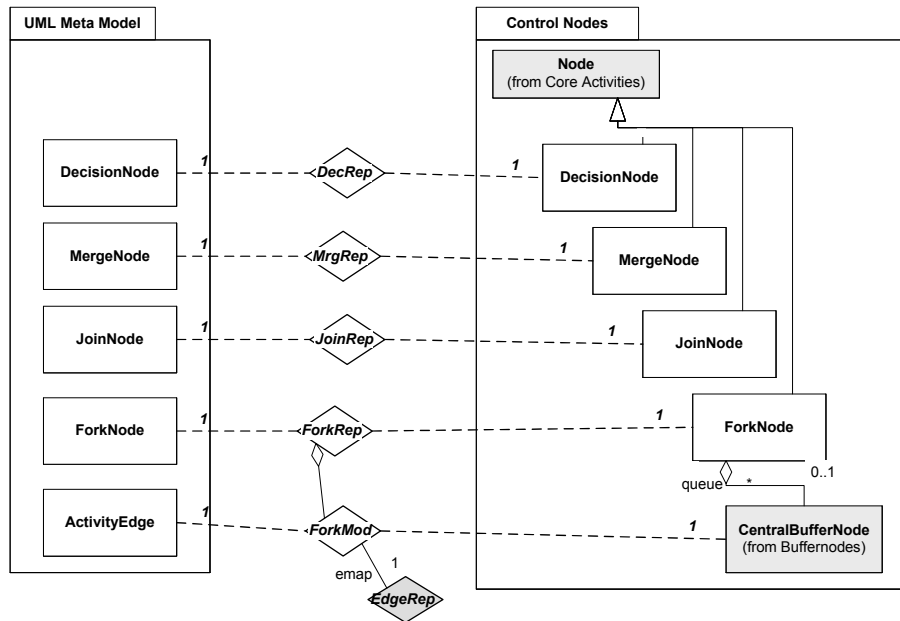


Figure B.50: The semantic mappings of the Controlnode package

tokens can be joined regardless of their base tokens. Object tokens on the other hand can only be joined if they belong to the same base token. An offer flowing out of a joinnode can thus have several base tokens and several spawnpoints. This needs to be taken into account when accepting such an offer (see `offer.getaccepted`).

Differences to standard UML The UML semantics description does not really exclude the joining of offers for different object tokens. It also does not especially mention this situation or give any semantics for it. In the finalization phase there was a proposal to output a sequence of different tokens in this case. We have not adopted this interpretation since we believe the idea of a single fused offer emerging from a joinnode should be identical for all inputs.

B.7.5 Mappings

The semantics mappings targeting the elements defined in the Controlnodes package are depicted in Fig. B.50.

DecRep - The DecisionNode Replication Relation DecisionNodes are replicated without additional details.

MrgRep - The MergeNode Replication Relation MergeNodes are replicated without additional details.

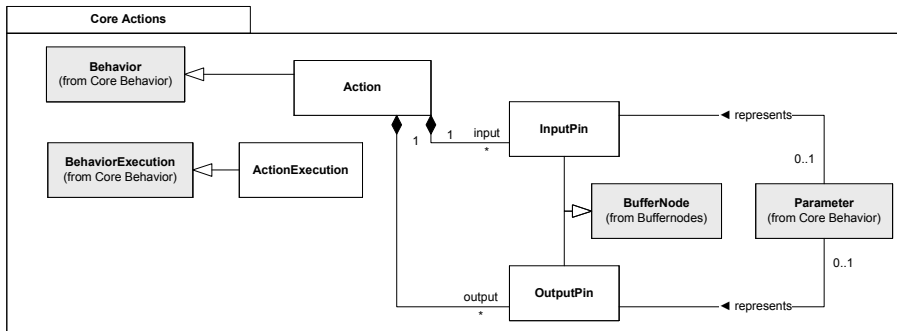


Figure B.51: The contents of the Core Actions package

JoinRep - The JoinNode Replication Relation JoinNodes are replicated without additional details.

ForkRep - The ForkNode Replication Relation ForkNodes are replicated without additional details.

ForkMod - The ForkNode Modification Relation

```

context ForkMod
domain=scope.domelement.outgoing
range=scope.ranelement.CentralBufferNode
inv:
  emap.domelement=self.domelement
  emap.ranelement.source=self.domelement
  
```

The ForkMod Relation captures the modifications which are performed in mapping Fork nodes. Outgoing flows of fork nodes are not directly connected to the forknode in the semantic domain but a `CentralBufferNode` which forms the queue for this outgoing edge is inserted. Note that the definition of domain and range of this nested Relation ensure a bijective mapping of the relevant elements only. The Relation refers to `EdgeRep` to ensure a correct connection.

B.8 Package Core Actions

Description The Core Actions package (see Fig. B.51) defines the core concepts of actions. These are in particular the classes `Action` and `ActionExecution`. By including `InputPin` and `OutputPin` in this package, we made it dependent on the package `BufferNodes` which disallows the use of Actions outside of Activities. The UML specification has a more general notion of actions being usable in every kind of behavior specification. To correctly support this general semantics, the introduction of abstract input and output classes as well as another package where the pins would inherit from these abstract classes would be needed. We decided to forgo this more general solution in favor of brevity of our specification.

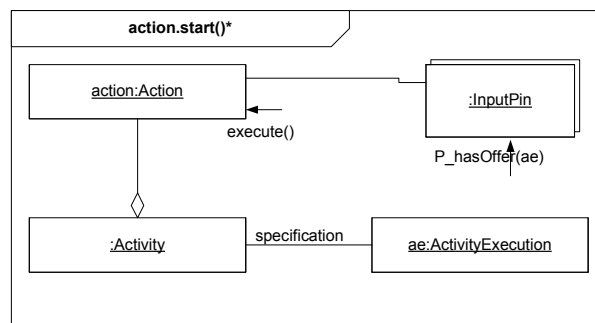


Figure B.52: DMM rule describing the starting of an action

B.8.1 Class Action

Description Actions are predefined basic units of behavior in UML. The execution of actions is controlled by an enclosing activity which determines the sequence(s) of action executions.

Package Action is defined in the Core Actions package.

Associations

InputPin [*] The input pins which must hold tokens for the action to execute

OutputPin [*] The output pins which hold tokens after the action executes

Constraints

context Action

inv:

```
self.input=(self.first->select(OCLtype=InputPin)).selfAllSuccs
self.output=(self.first->select(OCLtype=OutputPin)).selfAllSuccs
```

Operations

`start()*` An action is a specification of behavior. The operation `execute` starts the execution of this behavior. This entails a number of steps as displayed by the rule in Fig. B.52. A precondition (`P_hasOffer`) ensures that all inputs of the action contain a valid offer, i.e., the action can acquire all tokens necessary for its execution (`collectInputs`). The concrete execution of the behavior is handled by a new instance of `ActionExecution`, which is created, initialized, and started. The `ActionExecution` occurs in the context of the current `ActivityExecution` (thus the invoker edge).

Additionally, `Action` specifies an operation (interface) to be implemented by its concrete subclasses:

`execute(context:ActivityExecution)` The `execute` operation must be implemented by all subtypes of `Action`. It usually creates a specific execution instance, registers it with the invoking `ActivityExecution` (which is passed as a parameter) and process their inputs.

Semantics An action specifies a unit of behavior which is being invoked in the course of an `ActivityExecution`. An action can execute if it can obtain tokens on all inputs (i.e., control as well as data tokens). Upon execution it accepts all offers on its inputs simultaneously and executes its behavior. Since an action might execute multiple times, a separate `ActionExecution` is created for each invocation. This mechanism is generic for all actions. The different subclasses of action differ in the way their respective executions initialize and execute.

Differences to standard UML The class `Action` as presented here serves as both, a part of the structure of an activity (since it connects input and output ports) and as a specification of behavior. It thus integrates the UML concepts `Action` and `ActionNode`. Since the roles of input and output pins were extended to handle control tokens in this domain, no edges need to be directly attached to an action. It should also be noted that an action in this formalization only executes if it can obtain tokens stemming from the same `ActivityExecution`. This conforms to the notion of separate execution as described in [Obj04], Sect. 12.3.2.

B.8.2 Class `ActionExecution`

Description The `ActionExecution` is the runtime representation of the execution of an `Action`.

Package `ActionExecution` is defined in the Core Actions package.

Associations (none)

Constraints (none)

Operations `ActionExecution` provides a number of operations which specify uniform behavior for all of its subtypes.

`collectInputs()` / `collectInput(pin:InputPin)` The operation `collectInput` starts collecting the tokens from the input pins of the action. `collectInput(pin)` continues this task until all tokens have been collected. The rules in Figs. B.53 and B.54 provide the formalization of this collection loop.

`createOutputs()` / `createOutput(pin:OutputPin)` To produce the necessary outputs after an action has finished executing its behavior, the operation `createOutputs` (see Fig. B.55) is employed to locate the first output pin.

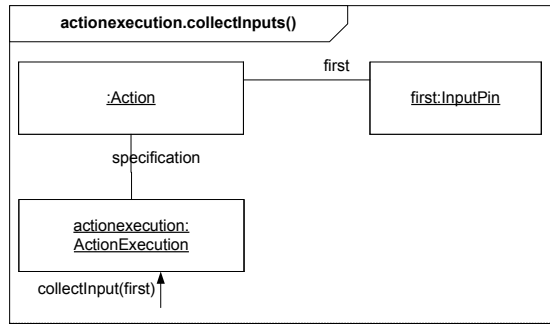


Figure B.53: DMM rules to start the collection of input tokens by an action

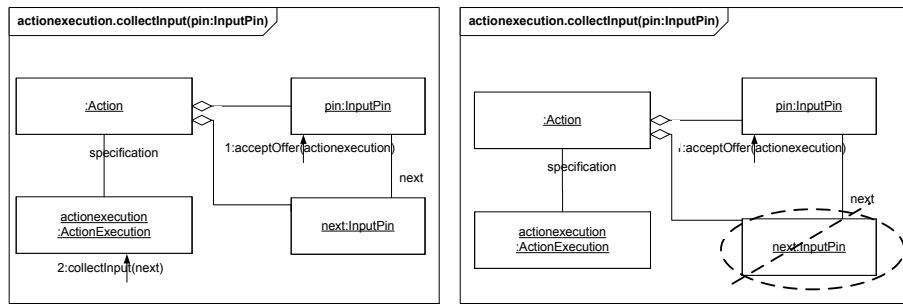


Figure B.54: DMM rule to continue/end the collection of input tokens by an action

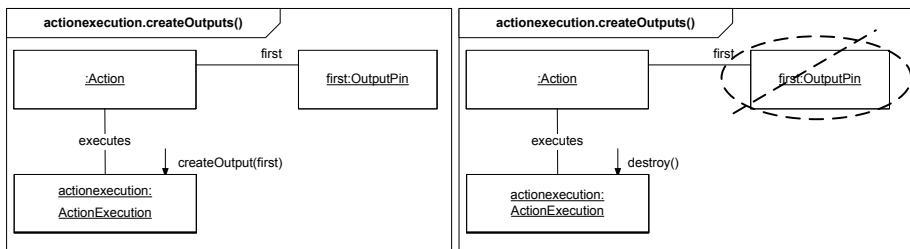


Figure B.55: DMM rules to start creating the outputs of an action

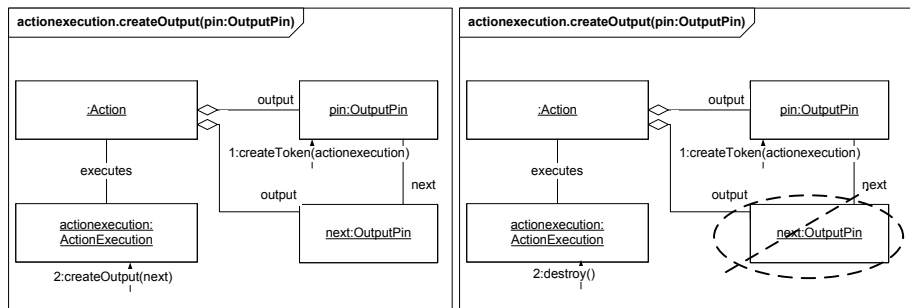


Figure B.56: DMM rules to continue/end creating the outputs of an action

Successively the operation `createOutput` is used to produce a token for each output pin (see Fig. B.56).

Additionally, `ActionExecution` defines three operations which are to be implemented by concrete subtypes of this abstract class:

`consumeData(ip:InputPin)` The operation `consumeData` is responsible for processing an input object of the action execution in a way as to make it accessible for the actions behavior.

`supplyData(op:OutputPin)` The operation `supplyData` needs to assign an internally produced value/object to an object token which flows in the surrounding activity graph.

`terminate()` All action executions must implement a `terminate` operation to enforce their external abortion.

Semantics The class `ActionExecution` encapsulates the actual execution of an Action. An `ActionExecution` is structured in three phases: initialization, execution and end. In the initialization phase, slots are created which can host data passed to the action by object tokens on input pins. The execution phase performs the intended behavior of the action. Note that these first phases are triggered in sequence by the operation `action.execute`. The end phase is triggered separately from this invocation to allow for interleaving (especially for `CallBehaviorActions`). The end phase deals with information passing out of the action, the creation of tokens on all output pins and the destruction of the execution instance and its associated slots. The general class `Action Execution` does only provide the standard functionality of collecting all tokens upon starting the execution and creating tokens upon finishing it. The initialization, execution and end phases are implemented by the concrete subclasses of Actions.

Differences to standard UML (none)

B.8.3 Class InputPin

Description `InputPins` collect the tokens necessary for the execution of actions.

Package `InputPin` is defined in the Core Actions package

Associations

Action [1] Each `InputPin` belongs to a specific action

Parameter [0..1] An `InputPin` can represent a parameter for the invoked behavior.

Constraints (none)

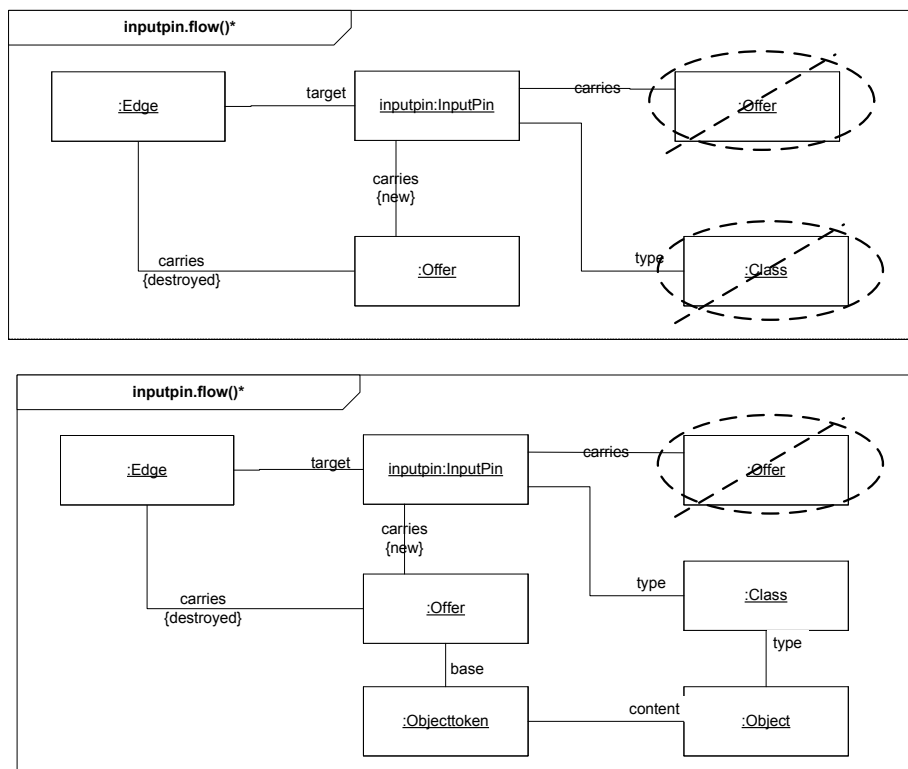


Figure B.57: DMM rules describing the flow of an offer onto an InputPin

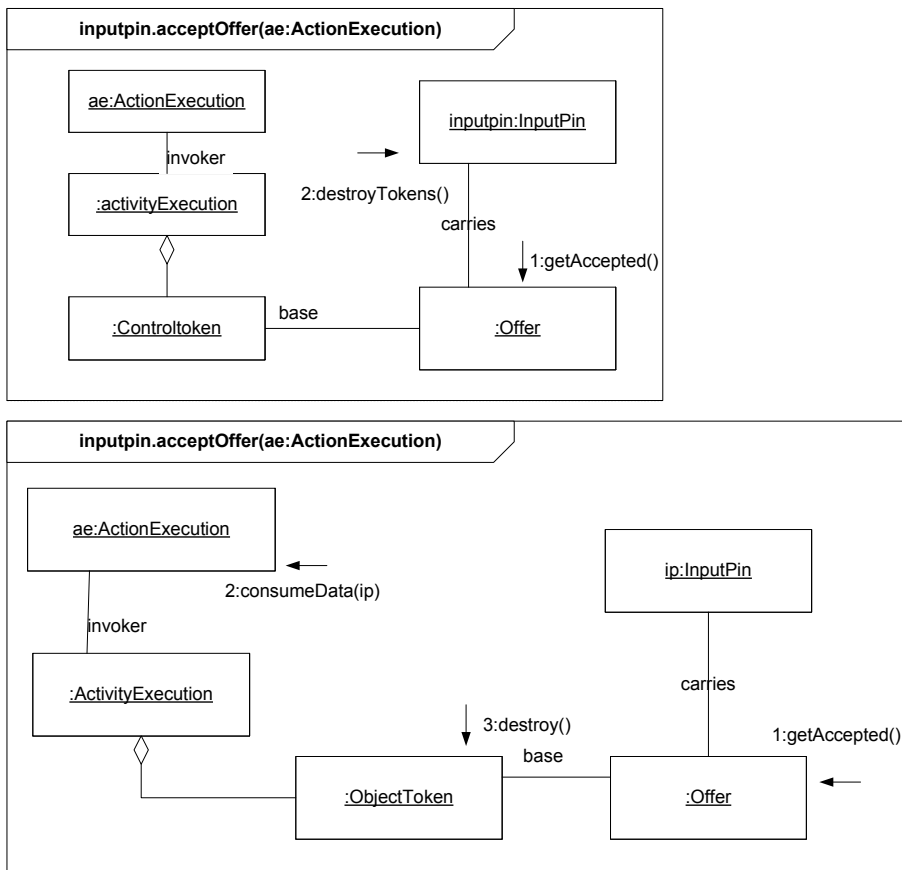


Figure B.58: DMM rules describing the acceptance of an offer by an InputPin

Operations

`flow()`* An offer can flow onto an input pin, if the node does not already contain an offer⁶ and if the input pin is untyped (upper rule of Fig. B.57). If the input is typed, however, then the typing of the underlying object has to be checked (lower rule of Fig. B.57)

`acceptOffer(ae:ActivityExecution)` The two rules in Fig. B.58 describe the acceptance of control tokens (upper rule) and object tokens (lower rule). The acceptance of an offer by an input pin is triggered by the execution of an action (since the acceptance of all input pins must be synchronized) thus the rules does pose no additional constraints.

The rules differ in the way the token resulting from the acceptance of the offer is being treated. Control tokens have fulfilled their purpose of enabling the execution of the action. They can thus be deleted upon being moved to the action. A separate operation is necessary to accomplish this deletion as the control token will be created in the process of moving to the

⁶we do not support input pins with multiplicity in this case study

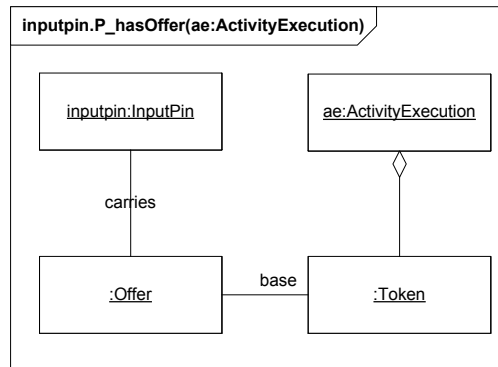


Figure B.59: DMM rules describing the acceptance of an offer by an InputPin

action (see the semantics of `offer.getAccepted`) and it cannot be matched beforehand.

Object tokens carry information which must be passed down to the behavior of the action. Since different actions can process this data in different ways, the action execution is invoked to consume the data obtained by the input pin.

P_hasOffer(ActionExecution) The predicate `hasOffer` is employed to check whether an input pin holds an offer for a token of a specific action execution (cf. Fig. B.59).

Semantics An input pin has the task of storing an offer until all input pins of an action can accept their offers at once, thus enabling the execution of the action. Since all tokens and offers are destroyed in this process of acceptance, no flow will occur out of an input node.

Differences to standard UML In contrast to the UML notational element `InputPin`, which is only able to store object tokens, `InputPins` in this semantic domain are more general as they buffer offers for object as well as control tokens. In fact, all inputs of an action are handled by `InputPins`. This deviation from the standard UML structure enables a more uniform handling of the inputs.

B.8.4 Class OutputPin

Description `OutputPins` store the tokens produced by the execution of actions.

Package `OutputPin` is defined in the Core Actions package.

Associations

Action [1] Each `OutputPin` belongs to a specific Action

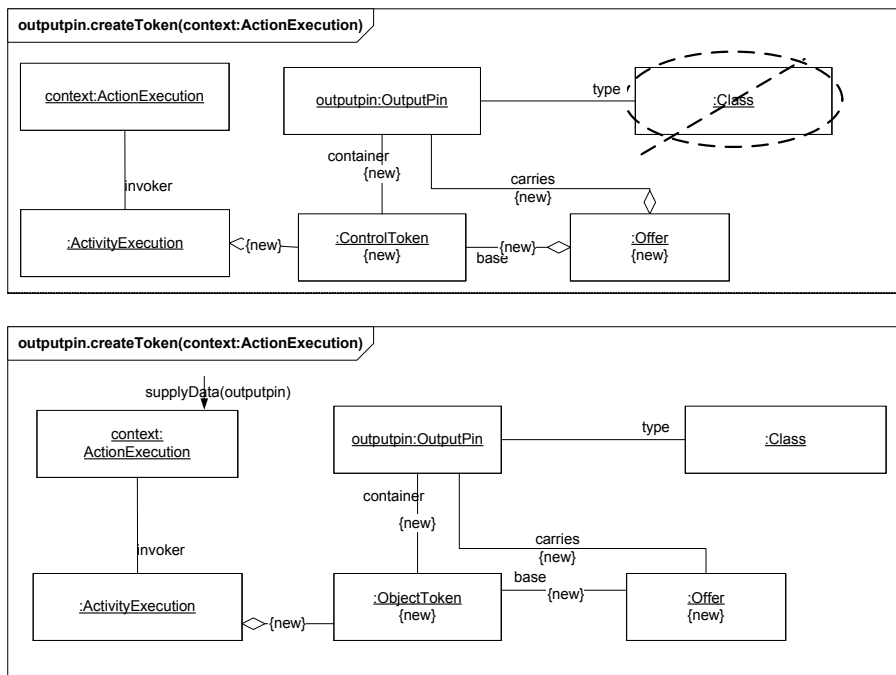


Figure B.60: DMM rules describing the creation of a token by an OutputPin

Parameter [0..1] OutputPins can be external representations for values passed out of an invoked behavior. On the behavior's level, this information is represented by a parameter.

Constraints (none)

Operations

createToken(context:ActionExecution) If an action finishes its execution it triggers the creation of tokens on its output pins. The two rules in Fig. B.60 describe this process for control and object tokens respectively. For control tokens this is rather easy as they are simply created along with an offer that may then move downstream.

The creation of object tokens entails a binding to the underlying object. This binding is highly reliant on the results produced by the executed action, thus there will be rules for the different kinds of actions available in UML. The rule provided in the lower half of Fig. B.60 thus calls the operation `supplyData` on the execution object to bind the correct data to the created token.

Both rules for this operation create an offer along with the new token. This offer will in turn flow out of the pin (see the operation flow).

flow()* The operation flow facilitates the flow of offers out of an OutputPin (see

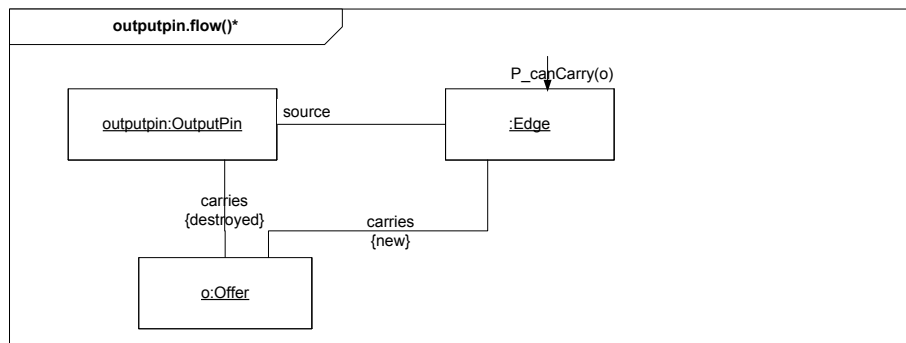


Figure B.61: DMM rule describing the flow of offers from an output pin

the rule in Fig. B.61). Note that in the case of multiple outgoing edges which might possible carry the offer, the rule will match on one of them non-deterministically.

Semantics An OutputPin has the task of storing tokens which result from the execution of an action until these tokens can move downstream.

Differences to standard UML Like InputPins, OutputPins have a broader meaning in this formalization than their syntactic counterparts in the UML meta model. They are mandatory elements between an action and its succeeding edges. They may contain object as well as control tokens. Using OutputPins in this way clarifies where a token (especially a control token) resides until it moves downstream and allows for a more uniform handling of in- and outputs of an action.

B.8.5 Mappings

The semantic mappings targeting elements of the Core Actions package are depicted in Fig. B.62

AtoRep - The Action Replication Relation

```
context AtoRep
inv:
  domelement.name=ranelement.name
```

The name of the action is preserved by the mapping.

IPRep - The InputPin Replication Relation

```
context IPRep
domain=scope.domelement
range=scope.ranelement.input
```

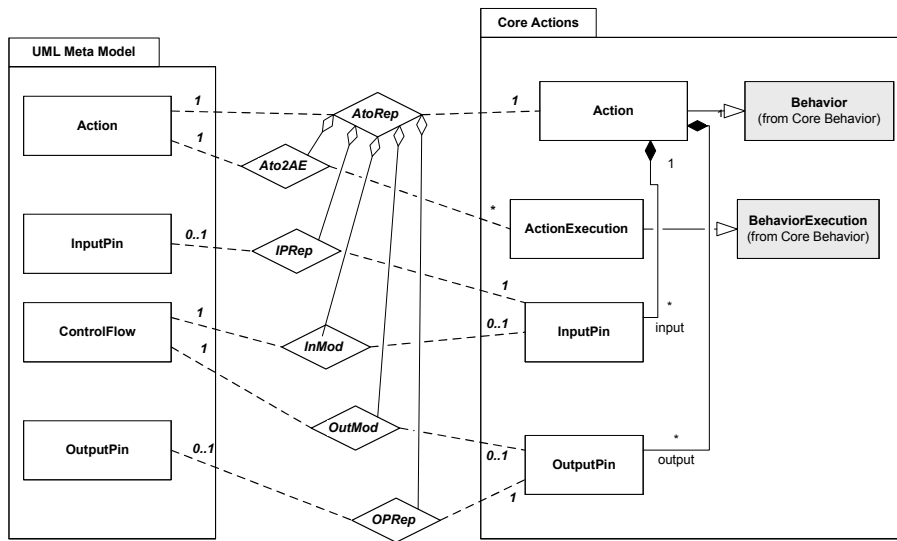


Figure B.62: Semantic mappings of the Core Actions package

The IPRep mapping does define special constraints. Note that as InputPins in the syntactic domain are ObjectNodes, the correct preservance of the typing is already guaranteed by the ONRep mapping. As the InpMod Relation will introduce additional InputPins on the semantic side, the domain side of IPRep is 0..1.

InpMod - The Input Modification Mapping

```
context InpMod
domain=scope.incoming->(select f|f.OCLtype=ControlFlow}
range=scope.input->(select i|i.type->isEmpty())
```

The InMod mapping ensures that for all directly incoming control flows of a (syntactic) action there is a untyped input pin on the semantic side. As UML also supports untyped input pins, this Relation is not onto.

OutMod - The Output Modification Mapping

```
context OutMod
domain=scope.outgoing->(select f|f.OCLtype=ControlFlow}
range=scope.output->(select i|i.type->isEmpty())
```

The OutMod Relation works symmetrically to the inpMod Relation.

OPRep - The OutputPin Replication Mapping

```
context OPRep
domain=scope.domelement
range=scope.ranelement.output
```

The OPRep Relation works symmetrically to the IPRep Relation.

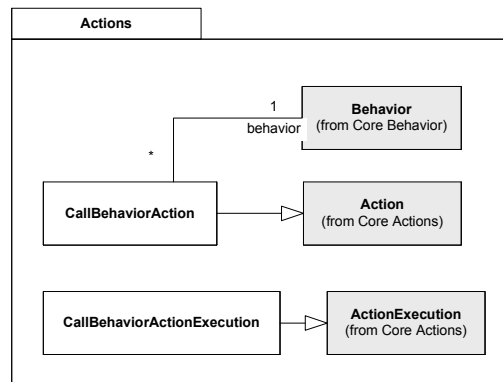


Figure B.63: The contents of the Actions package

B.9 Package Actions

Description The Actions package contains the concrete Actions as defined in the UML specification. We only provide semantics for the `CallBehaviorAction` here⁷ (cf. Fig. B.63) since it is one of the most common actions and also because it requires rather complex constructions for the correct passing of data and control back and forth. For each additional action type we would add another subtype of `Action` and its corresponding execution class.

B.9.1 Class `CallBehaviorAction`

Description A `CallBehaviorAction` is an action which invokes other behaviors.

Package `CallBehaviorAction` is defined in the Actions package.

Associations `behavior [1]` The behavior that is to be invoked by the action.

Constraints (none)

Operations

`execute(context:ActivityExecution)` The operation `execute` (cf. Fig. B.64) provides the details on a `CallBehaviorAction`'s execution. In particular it details that a new instance of `CallBehaviorActionExecutions` is created (in the context of the passed `ActivityExecution`. Then it processes the inputs of the action (by calling `createSlots` to create the slots to hold the objects passed as parameters of the call and `collectInputs` to fill these slots) and triggers the execution of the behavior to be called by it.

⁷and the auxiliary `DummyAction`, see Sect. B.10

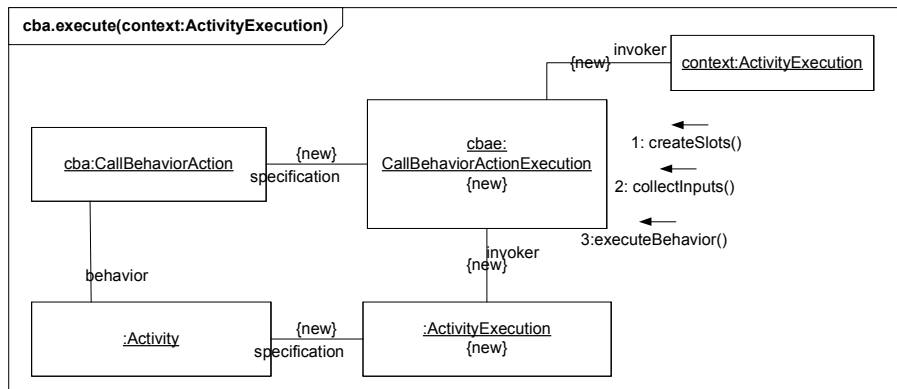


Figure B.64: DMM rule describing the execution of a CallBehaviorAction

Semantics A CallBehaviorAction will execute by invoking some other behavior. Here, this other behavior will be another activity. In general, however, other means can be used to specify this behavior. The execution of the Action itself is described in the superclass Action. The specific semantics of a call action result in different rules for an ActionExecution.

Differences to standard UML (none)

B.9.2 CallBehaviorActionExecution

Description A CallBehaviorActionExecution (CBAE for short) is the execution class for a CallBehaviorAction.

Package CBAE is defined in the Actions package.

Associations (none)

Constraints (none)

Operations

`createSlot(param:Parameter)/createSlots()` The operation `createSlot` creates slots for the execution of Call Actions. These slots will hold the data being passed into the invoked behavior as parameters or passed back from this behavior as results. Four rules (provided in Figs. B.65 and B.66) provide the different cases (again, we resort to recursive calls to implement a loop). Each rule application creates a new slot and registers it with the correct execution classes.

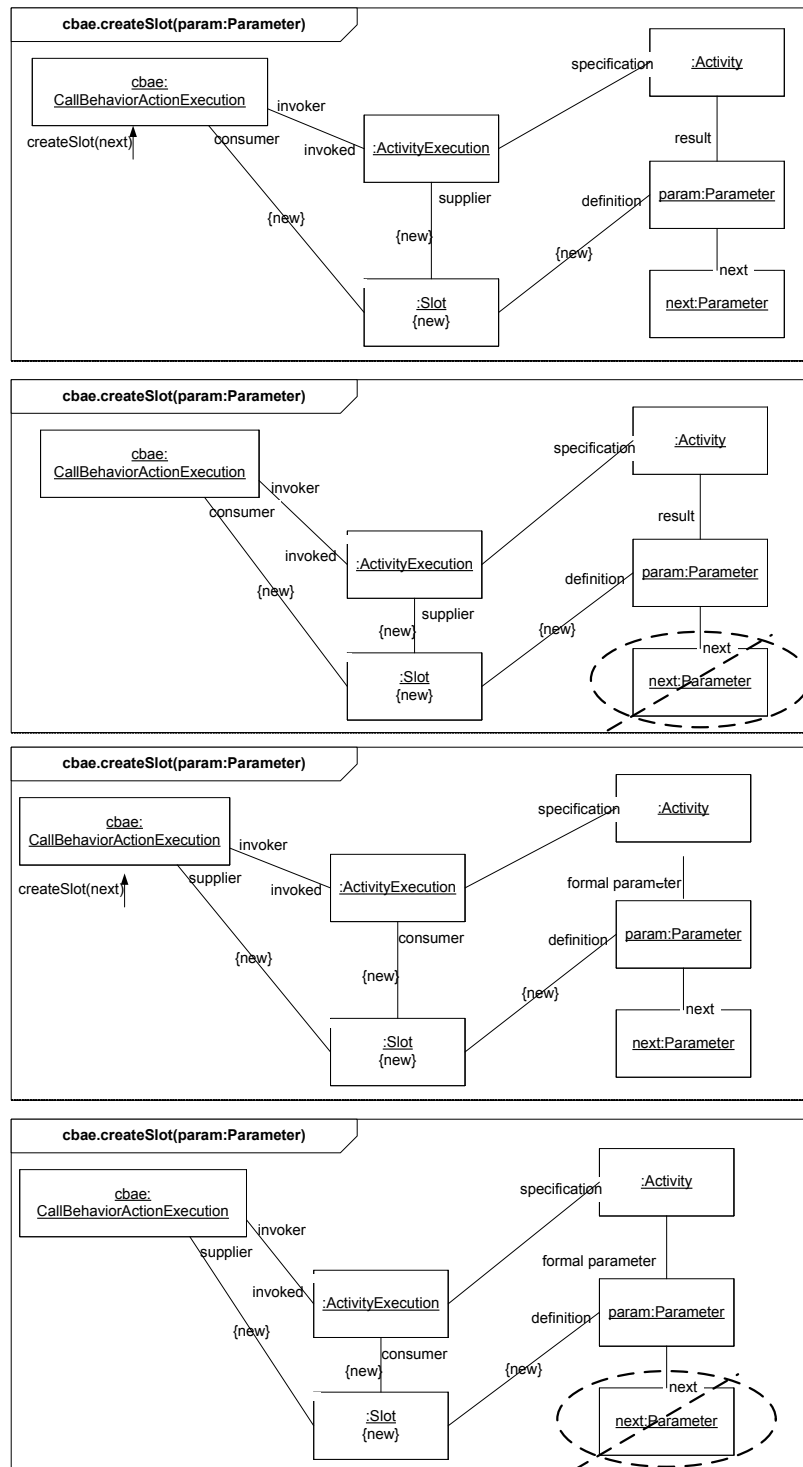


Figure B.65: DMM rules to create slots for parameter passing into an action execution

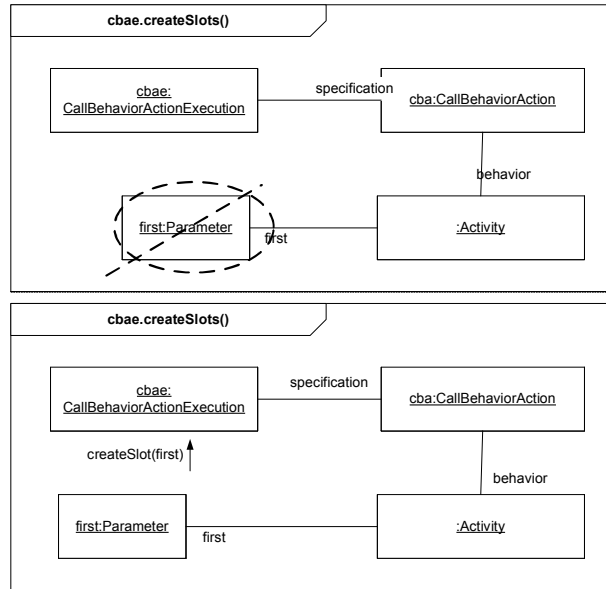


Figure B.66: DMM rule to begin the creation of slots

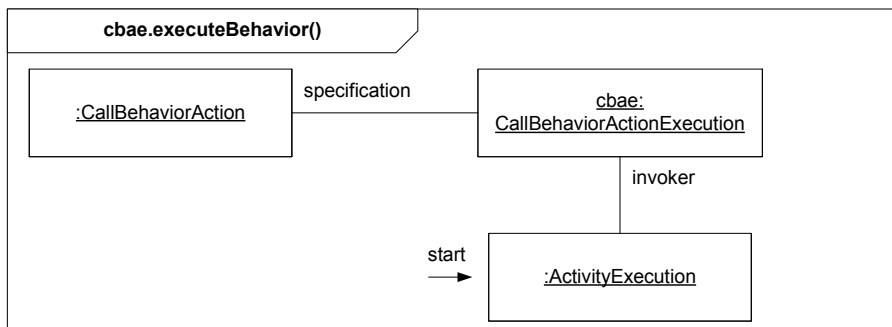


Figure B.67: DMM rule to start the actual behavior of an Action

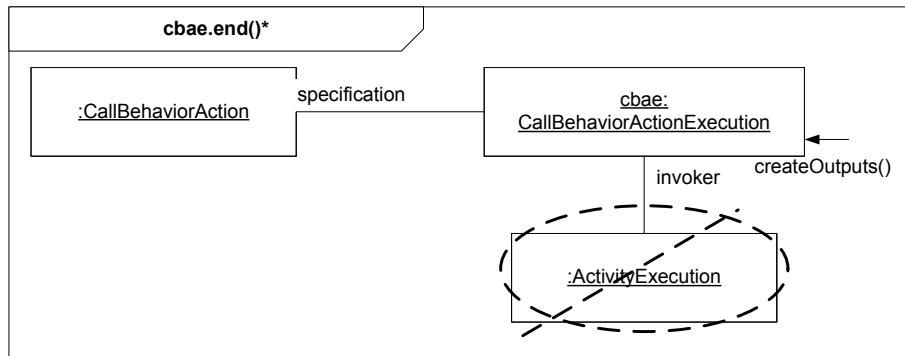


Figure B.68: DMM rule to end the execution of an Action

`executeBehavior()` The operation `executeBehavior` triggers the actual execution of the behavior specified by the action. The rule in Fig. B.67 describes the execution of a `CallBehaviorAction`, i.e., the invocation of the underlying behavior.

`end()*` The operation `end` specifies the end of an action execution. Here (Fig. B.68), the end of a `CallBehaviorAction` is indicated by the fact that the underlying behavior terminated (i.e. the execution instance is no longer present). If an `ActionExecution` is about to end, the necessary outputs need to be created to enable further execution of the enclosing activity. This is achieved by calling the `createOutputs` operation. Note that other types of actions (especially those with only a simple and clearly brief execution) may trigger the creation of outputs directly from their execution phase. Here, we were aiming for increased concurrency and a decoupling of the different behaviors.

`terminate()` The `terminate` operation is provided to stop an action execution prematurely. It can be invoked, e.g., if the invoking activity encounters an activity final node. For CBAE nodes this operation entails the termination of all invoked behaviors before deletion (see the rules in Fig. B.69). Other action types might react differently to the termination.

`consumeData(ip:InputPin)` While the collection of tokens on the input nodes is a common feature to all types of action (and thus handled by the general `ActionExecution` class, the processing of received data inputs is specific for each type of action. Thus the operation `consumeData` is invoked to request a special action execution instance to process an object token encountered on an `InputPin` (passed as the parameter). In the case of the CBAE (cf. Fig. B.70), all data inputs represent parameters of the behavior that is to be invoked. These inputs need to be stored in the respective slots for later consumption.

`supplyData(op:OutputPin)` Similar to the processing of data inputs, the creation of data outputs is specific to each action type. While, e.g., a `CreateObjectAction` passes on the newly created object, a CBAE passes all results of the invoked behavior on to its invoking activity. This passing is performed

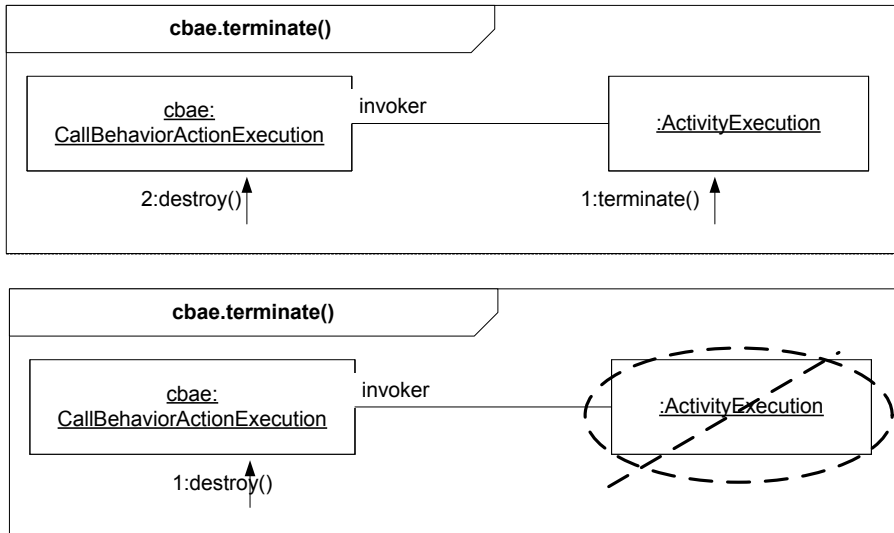


Figure B.69: DMM rule to terminate the execution of an action

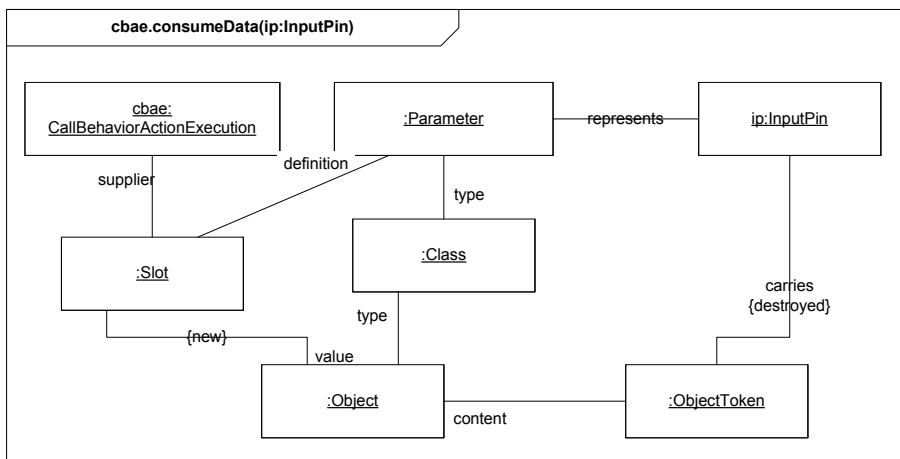


Figure B.70: DMM rule to end the execution of an action

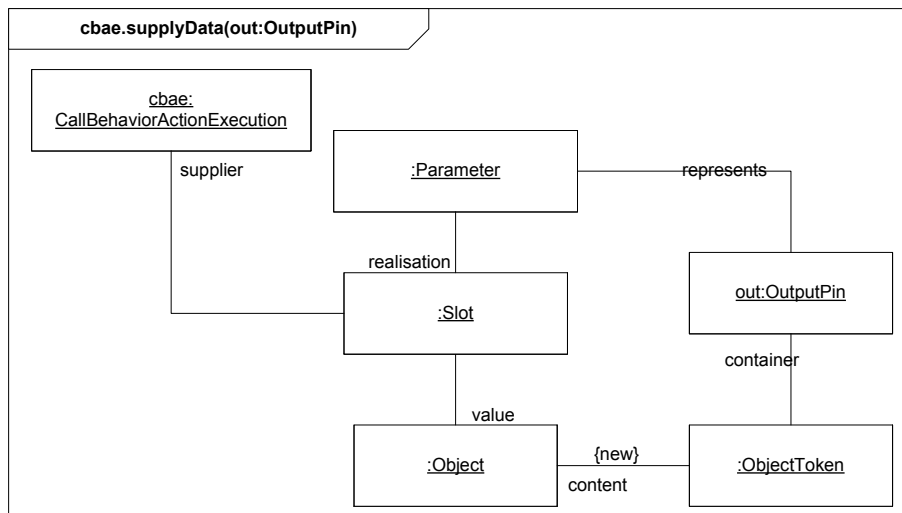


Figure B.71: DMM rule to end the execution of an action

by the operation `supplyData` (cf. Fig. B.71). In the operation, the object sitting in a slot `i` bound to an (yet unbound) object token on an output pin.

Semantics CBAE is the class which expresses the steps which a Call Action needs to perform when invoked. In particular it details how data slots for parameter passing are instantiated, how they are filled with data from the received object tokens and how the underlying behavior is invoked. Upon the end of this invoked behavior, the results are supplied to outgoing object tokens. Note that this concrete action execution class only needs to fill in detailed rules for operations not defined by its super classes.

Differences to standard UML The whole process of passing objects as parameters back and forth between different behaviors is not detailed in the UML semantics description.

B.9.3 Mappings

The semantic mappings targeting elements of the Actions package are depicted in Fig. B.72.

CBARep - The CallBehaviorAction Replication Relation

```
context CBARep
inv:
  rmap.domelement=self.domelement.behavior
  rmap.ranelement=self.ranelement.behavior
```

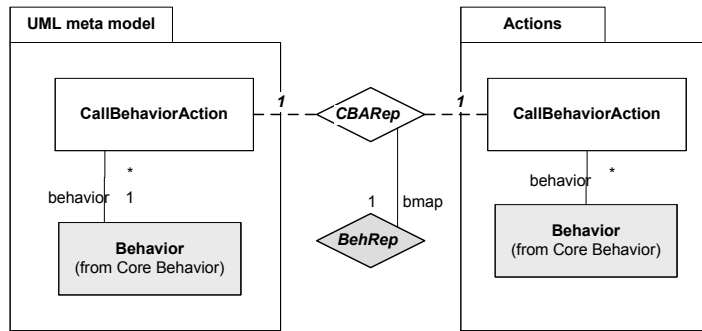


Figure B.72: The semantic mappings of the Actions package

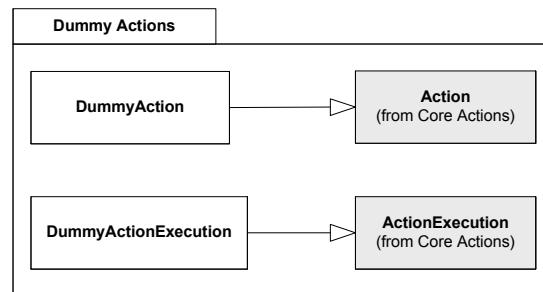


Figure B.73: Package Dummy Actions providing auxiliary elements for the formulation of incomplete Activity Diagrams

The behavior to be called must be preserved in the replication of the `CallBehaviorAction`.

B.10 Package Dummy Actions

The package Dummy Actions is not a regular package of the formalization of UML's Activity Diagrams. As discussed in Subsect. V.5.5., one possibility of handling incomplete models is the suppletion of auxiliary elements which provide some generic default behavior to allow for a complete interpretation of incomplete diagrams. The package Dummy Actions provides such auxiliary elements for Activity Diagrams.

The idea of this package is to provide a type of Action which is neither one of the base level actions which UML specifies nor is it refined into such base actions. When drawing Activity Diagrams, modelers often use actions which are characterized by their name only. Their exact behavior is left open as it is currently not of interest. Formally, however, such Actions must have one of the predefined types of the UML. The package Dummy Actions provides the formal means to interpret such sketched Activity Diagrams.

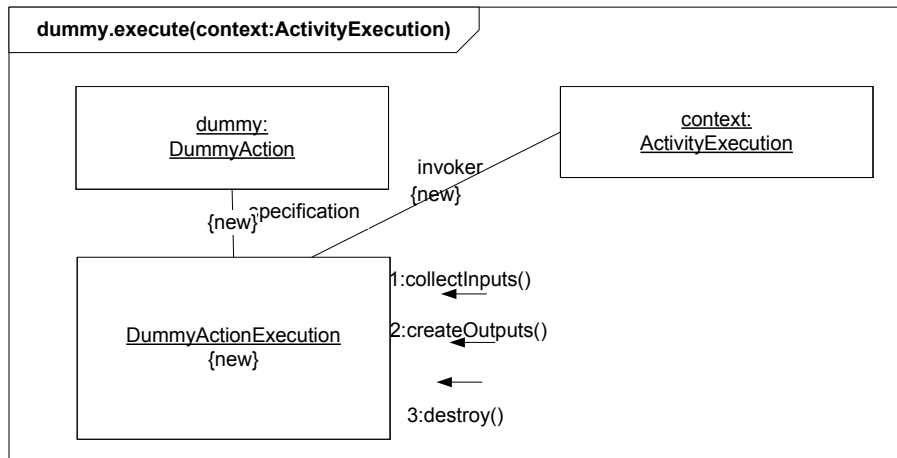


Figure B.74: DMM rule for executing a dummyAction

B.10.1 DummyAction

Description A DummyAction is an action which has no internal behavior but performs its external "duties" in an Activity Diagram.

Associations (none)

Constraints (none)

Operations

`execute(context:ActivityExecution)` The only operation of a dummy action is `execute` (cf. Fig. B.74). The execution of a dummy action entails the whole life cycle of this execution as a new execution is being created, inputs are collected, outputs created and the execution is destroyed again. No inner behavior is performed by this kind of action.

Semantics Dummy actions serve as placeholders for later refinements in an Activity Diagram. They may represent behavior which is not specified in any way yet and which is not relevant to the Activity Diagram at hand. When executed, Dummy Actions consume all offered inputs and (superficially) produce all required outputs (see also the semantics of `DummyActionExecution`).

Differences to standard UML Dummy Actions are an auxiliary construct which has no correspondence in the standard UML. To actually use Dummy Actions in a model, the construct has to be formally introduced in a UML Profile. We omitted this formal declaration here and simply use this kind of action in our examples.

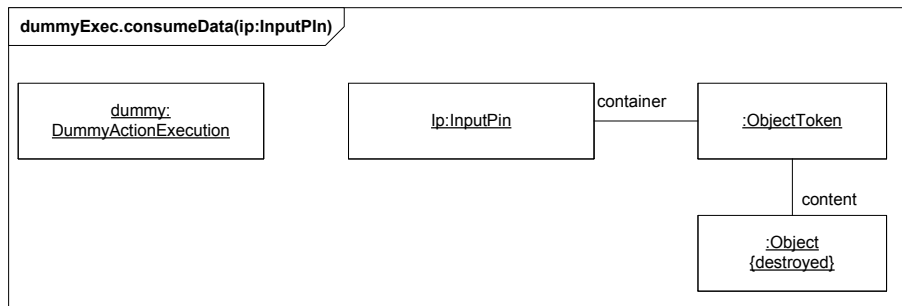


Figure B.75: DMM rule describing the consumption of input data by a `DummyActionExecution`

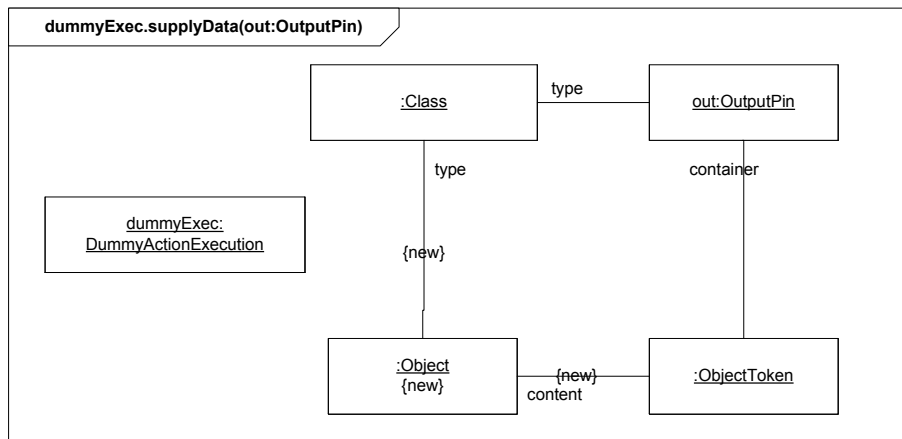


Figure B.76: DMM rule describing the supplication of output data by a `DummyActionExecution`

B.10.2 DummyActionExecution

Description A Dummy Action Execution represents the execution of a Dummy Action.

Associations (none)

Constraints (none)

Operations `DummyActionExecution` implements the minimal set of operations defined by `ActionExecution`:

`consumeData(ip:InputPin)` A `DummyAction` consumes destroys all objects passed to it (cf. Fig. B.75).

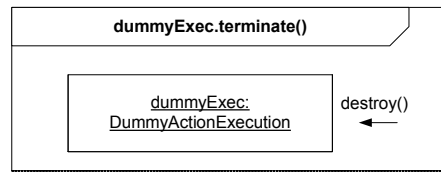


Figure B.77: DMM rule describing the termination of a DummyActionExecution

supplyData(op:OutputPin) A DummyAction creates empty objects (of the correct type) for its outputs. Note that this is a minimal default behavior which probably invalidates cardinalities or other constraints in many cases. It does, however, allow the surrounding activity to continue its execution with at least nominally complete tokens (cf. Fig. B.76).

terminate() Termination of a DummyAction simply consists in the deletion of the execution class (cf. Fig. B.77).

Semantics DummyActionExecutions consume all of their inputs and create empty outputs without any internal processing. Their sole purpose is to enable an at least provisional further execution of the surrounding activity.

Differences to standard UML (none)

Bibliography

- [AB99] W. v. d. Aalst and T. Basten. Inheritance of Workflows: An approach to tackling problems related to change. Computing Science Reports 99/06, Eindhoven University of Technology, 1999.
- [AD94] R. Alur and D. L. Dill. A Theory of Timed Automata. *Theoretical Computer Science*, 126(2):183–235, 1994.
- [AE96] M. Andries and G. Engels. A Hybrid Query Language for an Extended Entity-Relationship Model. *Journal of Visual Languages and Computing*, 7(3):321–352, 1996.
- [AEH⁺96] M. Andries, G. Engels, A. Habel, B. Hoffmann, H.-J. Kreowski, S. Kuske, D. Plump, A. Schürr, and G. Taentzer. Graph Transformation for Specification and Programming. Report 7/96, Univ. Bremen, 1996.
- [AES01a] J. Alvarez, A. Evans, and P. Sammut. MML and the meta-model architecture. In *Workshop on Transformations in UML (WTUML'01), associated with ETAPS'01*, 2001.
- [AES01b] J. M. Alvarez, A. Evans, and P. Sammut. Mapping between Levels in the Metamodel Architecture. In *UML '01: Proceedings of the 4th International Conference on The Unified Modeling Language, Modeling Languages, Concepts, and Tools*, pages 34–46, London, UK, 2001. Springer-Verlag.
- [AK01] C. Atkinson and T. Kühne. The Essence of Multilevel Metamodeling. In *UML '01: Proceedings of the 4th International Conference on The Unified Modeling Language, Modeling Languages, Concepts, and Tools*, pages 19–33, London, UK, 2001. Springer-Verlag.
- [AK02a] D. H. Akehurst and S. Kent. A Relational Approach to Defining Transformations in a Metamodel. In J.-M. Jézéquel, H. Hussmann, and S. Cook (eds.), *UML 2002 - The Unified Modeling Language. Model Engineering, Languages, Concepts, and Tools. 5th International Conference, Dresden, Germany, September/October 2002, Proceedings*, volume 2460 of *Lecture Notes in Computer Science (LNCS)*, pages 243–258. Springer, 2002.

- [AK02b] C. Atkinson and T. Kühne. Profiles in a strict metamodelling framework. *Science of Computer Programming*, 44(1):5–22, 2002.
- [AK02c] C. Atkinson and T. Kühne. Rearchitecting the UML infrastructure. *ACM Transactions on Modeling and Computer Simulation*, 12(4):290–321, 2002.
- [AKHS00] C. Atkinson, T. Khne, and B. Henderson-Sellers. To Meta or Not to Meta—That Is the Question. *Journal of Object-Oriented Programming*, 13, No. 8:32–35, December 2000.
- [AKP03] D. H. Akehurst, S. Kent, and O. Patrascoiu. A relational approach to defining and implementing transformations between metamodels. *Software and Systems Modeling*, 2(4):215–239, 2003.
- [Alc01] Alcatel et. al. Infrastructure of the Unified Modeling Language 2.0 Specification. OMG document ad/2001-08-11, August 2001.
- [AR02] E. Astesiano and G. Reggio. An Attempt at Analysing the Consistency Problems in the UML from a Classical Algebraic Viewpoint. In M. Wirsing, D. Pattinson, and R. Hennicker (eds.), *WADT*, volume 2755 of *Lecture Notes in Computer Science*, pages 56–81. Springer, 2002.
- [Atk99] C. Atkinson. Supporting and Applying the UML Conceptual Framework. In Bézivin and Muller [BM98], pages 21–36.
- [Baa02] T. Baar. How to ground meta-circular OCL descriptions – a set-theoretic approach –. In T. Clark, A. Evans, and K. Lano (eds.), *Proceedings, Fourth Workshop on Rigorous Object-Oriented Methods, London, 2002*, 2002.
- [BCR00] E. Börger, A. Cavarra, and E. Riccobene. An ASM semantics for UML Activity Diagrams. In T. Rus (ed.), *Proceedings of Algebraic Methodology and Software Technology (AMAST 2000)*, volume 1816 of *Lecture Notes in Computer Science*. Springer, 2000.
- [BCR03] E. Börger, A. Cavarra, and E. Riccobene. Modeling the meaning of transitions from and to concurrent states in UML state machines. In *SAC '03: Proceedings of the 2003 ACM symposium on Applied computing*, pages 1086–1091, New York, NY, USA, 2003. ACM Press.
- [BE_dL⁺03] R. Bardohl, H. Ehrig, J. de Lara, O. Runge, G. Taentzer, and I. Weinhold. Node Type Inheritance Concept for Typed Graph Transformation. Technical report, TU Berlin, Forschungsberichte des Fachbereichs Informatik, 2003.
- [Bey93] M. Beyer. *AGG An Algebraic Graph System, User Manual*. Technical University of Berlin, Department of Computer Science, 1993.

- [Bez04] J. Bezivin. In Search of a Basic Principle for Model Driven Engineering. *UPGRADE, The European Journal for the Informatics Professional*, 2:21–24, 2004.
- [BG01] J. Bzivin and O. Gerb. Towards a Precise Definition of the OMG/MDA Framework. In *ASE '01: Proceedings of the 16th IEEE International Conference on Automated Software Engineering*, page 273, Washington, DC, USA, 2001. IEEE Computer Society.
- [BG04] C. Bock and M. Gruninger. Inputs and Outputs in the Process Specification Language. Technical Report NISTIR 7152, National Institute of Standards and Technology, 2004.
- [BG05] C. Bock and M. Gruninger. PSL: A semantic domain for flow models. *Software and Systems Modeling*, 4, Issue 2:209 – 231, May 2005.
- [BH05] J. Bezivin and R. Heckel. 04101 Summary – Language Engineering for Model-driven Software Development. In J. Bezivin and R. Heckel (eds.), *Language Engineering for Model-Driven Software Development*, number 04101 in Dagstuhl Seminar Proceedings. Internationales Begegnungs- und Forschungszentrum (IBFI), Schloss Dagstuhl, Germany, 2005.
- [BKPPPT01] P. Bottoni, M. Koch, F. Parisi-Presicce, and G. Taentzer. A Visualization of OCL Using Collaborations. In M. Gogolla and C. Kobryn (eds.), *UML 2001 - The Unified Modeling Language. Modeling Languages, Concepts, and Tools. 4th International Conference, Toronto, Canada, October 2001, Proceedings*, volume 2185 of *LNCIS*, pages 257–271. Springer, 2001.
- [BM97] R. Bruni and U. Montanari. Zero-safe nets: The individual token approach. In *WADT '97: Selected papers from the 12th International Workshop on Recent Trends in Algebraic Development Techniques*, pages 122–140, London, UK, 1997. Springer-Verlag.
- [BM98] J. Bézivin and P.-A. Muller (eds.). *The Unified Modeling Language, UML'98 - Beyond the Notation. First International Workshop, Mulhouse, France, June 1998*, 1998.
- [BMS04] A. F. Blackwell, K. Marriott, and A. Shimojima (eds.). *Diagrammatic Representation and Inference, Third International Conference, Diagrams 2004, Cambridge, UK, March 22-24, 2004, Proceedings*, volume 2980 of *Lecture Notes in Computer Science*. Springer, 2004.
- [Boc03a] C. Bock. UML 2 Activity and Action Models. *Journal of Object Technology*, 2(4):43–53, 2003.
- [Boc03b] C. Bock. UML 2 Activity and Action Models, Part 2. *Journal of Object Technology*, 2(5):41–56, 2003.
- [Boc03c] C. Bock. UML 2 Activity and Action Models, Part 3: Control Nodes. *Journal of Object Technology*, 2(6):7–23, 2003.

- [Boc03d] C. Bock. UML without Pictures. *IEEE Software*, 20(5):33–35, 2003.
- [Boc04] C. Bock. UML 2 Activity and Action Models Part 4: Object Nodes. *Journal of Object Technology*, 3:27–41, 2004.
- [BP01a] L. Baresi and M. Pezzè. Improving UML with Petri nets. In *Proc. ETAPS2001 Workshop on Uniform Approaches to Graphical Process Specification Techniques (UniGra), Genova, Italy*, Electronic Notes in Theoretical Computer Science (ENTCS). Elsevier Science, 2001.
- [BP01b] L. Baresi and M. Pezzè. On formalizing UML with high-level petri nets. In *Concurrent object-oriented programming and petri nets: Advances in Petri Nets*, pages 276–304. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2001.
- [BR04] B. Böhlen and U. Ranger. Concepts for Specifying Complex Graph Transformation Systems. In H. Ehrig, G. Engels, F. Parisi-Presicce, and G. Rozenberg (eds.), *Graph Transformations, Second International Conference, ICGT 2004, Proceedings*, volume 3256 of *Lecture Notes in Computer Science (LNCS)*, pages 96–111. Springer Verlag, 2004.
- [CEK⁺00] T. Clark, A. Evans, S. Kent, S. Brodsky, and S. Cook. A Feasibility Study in Rearchitcting the UML as a Family of Languages using a Precise OO Meta-modeling Approach. Available at www.puml.org, September 2000.
- [CEK01] A. Clark, A. Evans, and S. Kent. The Meta-Modeling Language Calculus: Foundation Semantics for UML. In H. Hußmann (ed.), *Fundamental Approaches to Software Engineering, 4th International Conference, FASE 2001, Proceedings*, volume 2029 of *Lecture Notes in Computer Science*. Springer-Verlag, 2001.
- [CEL⁺96] A. Corradini, H. Ehrig, M. Löwe, U. Montanari, and J. Padberg. The Category of Typed Graph Grammars and their Adjunction with Categories of Derivations. In *Graph Grammars and their Application to Computer Science: 5th International Workshop*, pages 56–74. Springer-Verlag, 1996.
- [Cer05] M. Cerioli (ed.). *Fundamental Approaches to Software Engineering, 8th International Conference, FASE 2005, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2005, Edinburgh, UK, April 4-8, 2005, Proceedings*, volume 3442 of *Lecture Notes in Computer Science*. Springer, 2005.
- [Che76] P. P.-S. Chen. The Entity-Relationship Model - toward a unified view of data. *ACM Trans. Database Syst.*, 1(1):9–36, 1976.
- [CHM00] A. Corradini, R. Heckel, and U. Montanari. Graphical operational semantics. In *Proc. ICALP2000 Workshop on Graph Transformation and Visual Modelling Techniques*. Carleton Scientific, 2000.

- [CKM⁺99] S. Cook, A. Kleppe, R. Mitchell, B. Rumpe, J. Warmer, and A. C. Wills. Defining UML Family Members Using Prefaces. In C. Mingins (ed.), *Proceedings of TOOLS Pacific 1999*. IEEE Computer Society, 1999.
- [CMR96] A. Corradini, U. Montanari, and F. Rossi. Graph processes. *Fundamenta Informaticae*, 26(3,4):241–266, 1996.
- [Coi96] P. Cointe. Reflective languages and metalevel architectures. *ACM Comput. Surv.*, 28(4es):151, 1996.
- [Com05] Compilers.net. List of Parser Generators. <http://www.compilers.net/Dir/ParserGens.htm>, 2005.
- [CRS04] A. Cavarra, E. Riccobene, and P. Scandurra. A framework to simulate UML models: moving from a semi-formal to a formal environment. In *SAC '04: Proceedings of the 2004 ACM symposium on Applied computing*, pages 1519–1523, New York, NY, USA, 2004. ACM Press.
- [DJPV03] W. Damm, B. Josko, A. Pnueli, and A. Votintseva. Understanding UML: A Formal Semantics of Concurrency and Communication in Real-Time UML. In F. de Boer, M. Bonsangue, S. Graf, and W.-P. de Roever (eds.), *Proceedings of the 1st Symposium on Formal Methods for Components and Objects (FMCO 2002)*, volume 2852 of *LNCS Tutorials*, pages 70–98, 2003.
- [dLETE04] J. de Lara, C. Ermel, G. Taentzer, and K. Ehrig. Parallel Graph Transformation for Model Simulation applied to Timed Transition Petri Nets. *Electr. Notes Theor. Comput. Sci.*, 109:17–29, 2004.
- [DM95] F.-N. Demers and J. Malenfant. Reflection in logic, functional and object-oriented programming: a Short Comparative Study. In *Proceedings of the IJCAI'95 Workshop on Reflection and Metalevel Architectures and their Applications in AI*, pages 29–38, August 1995.
- [Dör95] H. Dörr. *Efficient Graph Rewriting and its Implementation*, volume 922 of *Lecture Notes in Computer Science*. Springer-Verlag, Berlin, 1995.
- [DP05] B. Dobing and J. Parson. UML in Practice: A Survey of UML Use. available at www.omg.org/docs/ad/05-02-08.pdf, 2005.
- [dRS84] J. des Rivieres and B. C. Smith. The implementation of procedurally reflective languages. In *LFP '84: Proceedings of the 1984 ACM Symposium on LISP and functional programming*, pages 331–347, New York, NY, USA, 1984. ACM Press.
- [EB04] C. Ermel and R. Bardohl. Scenario animation for visual behavior models: A generic approach. *Software and Systems Modeling*, 3:164 – 177, 2004.
- [EE93] H. Ehrig and G. Engels. Towards a Module Concept for Graph Transformation Systems. Technical report, Leiden University, 1993.

- [EE95] J. Ebert and G. Engels. Specialization of Object Life Cycle Definitions. Fachbericht Informatik 19/95, Universität Koblenz-Landau, Fachbereich Informatik, Koblenz, 1995.
- [EE^{dL}+05] H. Ehrig, K. Ehrig, J. de Lara, G. Taentzer, D. Varró, and S. Varró-Gyapay. Termination Criteria for Model Transformation. In Cerioli [Cer05], pages 49–63.
- [EEKR99] H. Ehrig, G. Engels, H.-J. Kreowski, and G. Rozenberg (eds.). *Handbook on Graph Grammars and Computing by Graph Transformation. Volume 2: Specifications and Programming*. World Scientific, Singapore, 1999.
- [EEPT05] H. Ehrig, K. Ehrig, U. Prange, and G. Taentzer. Formal Integration of Inheritance with Typed Attributed Graph Transformation for Efficient VL Definition and Model Manipulation. In *2005 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC'05)*, pages 71–78, 2005.
- [EFLR99] A. Evans, R. France, K. Lano, and B. Rumpe. Meta-modelling semantics of UML. In H. Kilov (ed.), *Behavioural Specifications for Businesses and Systems*. Kluwer, 1999.
- [EH00] G. Engels and R. Heckel. From Trees to Graphs: Defining the Semantics of Diagram Languages with Graph Transformation. In *ICALP Satellite Workshops*, pages 373–382, 2000.
- [EHHS00] 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 (eds.), *UML*, volume 1939 of *Lecture Notes in Computer Science*, pages 323–337. Springer, 2000.
- [EHHS02] G. Engels, J. H. Hausmann, R. Heckel, and S. Sauer. Testing the Consistency of Dynamic UML Diagrams. In *Proc. Sixth International Conference on Integrated Design and Process Technology (IDPT 2002)*, 2002.
- [EHK⁺97] H. Ehrig, R. Heckel, M. Korff, M. Löwe, L. Ribeiro, A. Wagner, and A. Corradini. Algebraic Approaches to Graph Transformation, Part II: Single Pushout Approach and Comparison with Double Pushout Approach. In Rozenberg [Roz97], pages 247–312.
- [EHK01] G. Engels, R. Heckel, and J. Küster. Rule-based Specification of Behavioral Consistency based on the UML Meta Model. In Gogolla and Kobryn [GK01], pages 272–287.
- [EHKZ05] C. Ermel, Hölscher, S. Kuske, and P. Ziemann. Animated Simulation of Integrated Behavioral Models based on Graph Transformation. In *IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC'05)*. IEEE Computer Society, 2005.
- [EHS99] G. Engels, R. Heckel, and S. Sauer. Dynamic Meta Modeling: A Graphical Approach to Operational Semantics. In *Proc. OOPSLA'99 Workshop on Rigorous Modeling and Analysis with the*

- UML: Challenges and Limitations, Denver, CO, USA, November 2 1999.*
- [EHSW99] G. Engels, R. Hüicking, S. Sauer, and A. Wagner. UML Collaboration Diagrams and their Transformation to Java. In R. France and B. Rumpe (eds.), *UML'99 - The Unified Modeling Language. Beyond the Standard. Second International Conference, Fort Collins, CO, USA, October 28-30. 1999, Proceedings*, volume 1723 of *Lecture Notes in Computer Science (LNCS)*, pages 473–488. Springer-Verlag, October 1999.
- [EK99] A. Evans and S. Kent. Core Meta-Modelling Semantics of UML: The pUML Approach. In R. B. France and B. Rumpe (eds.), *UML'99: The Unified Modeling Language - Beyond the Standard, Second International Conference, Fort Collins, CO, USA, October 28-30, 1999, Proceedings*, volume 1723 of *Lecture Notes in Computer Science*. Springer, 1999.
- [EKMR99] H. Ehrig, H.-J. Kreowski, U. Montanari, and G. Rozenberg (eds.). *Handbook on Graph Grammars and Computing by Graph Transformation. Volume 3: Concurrency, Parallelism, and Distribution*. World Scientific, Singapore, 1999.
- [EPS73] H. Ehrig, M. Pfender, and H. J. Schneider. Graph Grammars: An Algebraic Approach. In *IEEE Conf. on Automata and Switching Theory*, pages 167–180, Iowa City, 1973.
- [EPT04] H. Ehrig, U. Prange, and G. Taentzer. Fundamental Theory for Typed Attributed Graph Transformation. In H. Ehrig, G. Engels, F. Parisi-Presicce, and G. Rozenberg (eds.), *Graph Transformations, Second International Conference, ICGT 2004, Rome, Italy, September 28 - October 2, 2004, Proceedings*, volume 3256 of *Lecture Notes in Computer Science*, pages 161–177. Springer, 2004.
- [Esh02] R. Eshuis. *Semantics and Verification of UML Activity Diagrams for Workflow Modeling*. PhD thesis, University of Twente, 2002.
- [ESW⁺05] A. Evans, P. Sammut, J. S. Willans, A. Moore, and G. Maskeri. A Unified Superstructure for UML. *Journal of Object Technology*, 4(1):165–182, 2005.
- [EW01] R. Eshuis and R. Wieringa. A Real-Time Execution Semantics for UML Activity Diagrams. In *FASE '01: Proceedings of the 4th International Conference on Fundamental Approaches to Software Engineering*, pages 76–90, London, UK, 2001. Springer-Verlag.
- [EW04] R. Eshuis and R. Wieringa. Tool Support for Verifying UML Activity Diagrams. *IEEE Trans. Software Eng.*, 30(7):437–447, 2004.
- [Fav04] J.-M. Favre. Towards a Basic Theory to Model Driven Engineering. In *Proceedings of the Third Workshop in Software Model Engineering WiSME 04@UML2004 Lisboa, 2004*.

- [Fla02] R. G. Flatscher. Metamodeling in EIA/CDIF—meta-metamodel and metamodels. *ACM Trans. Model. Comput. Simul.*, 12(4):322–342, 2002.
- [FM03] S. Flake and W. Müller. Formal semantics of static and temporal state-oriented OCL constraints. *Software and Systems Modeling*, 2(3):164–186, 2003.
- [FNTZ00] T. Fischer, J. Niere, L. Torunski, and A. Zündorf. Story Diagrams: A new Graph Transformation Language based on UML and Java. In H. Ehrig, G. Engels, H.-J. Kreowski, and G. Rozenberg (eds.), *Theory and Application of Graph Transformations, 6th International Workshop, TAGT'98, Paderborn, Germany, November 16-20, 1998, Selected Papers*, volume 1764 of *Lecture Notes in Computer Science (LNCS)*. Springer-Verlag, 2000.
- [Fow05a] M. Fowler. Language Workbenches: The Killer-App for Domain Specific Languages? <http://martinfowler.com/articles/languageWorkbench.html>, June 2005.
- [Fow05b] M. Fowler. UML mode. available online at <http://www.martinfowler.com/bliki/UmlMode.html>, 2005.
- [FQL⁺03] J. M. Fuentes, V. Quintana, J. Llorens, G. Genova, and R. Prieto-Diaz. Errors in the UML metamodel? *SIGSOFT Softw. Eng. Notes*, 28(6):3–3, 2003.
- [FS00] M. Fowler and K. Scott. *UML Distilled, Second Edition*. Oldenbourg, 2000.
- [FTF] Finalization Task Force - UML 2 Superstructure Issues Database. <http://www.omg.org/issues/uml2-superstructure-fff.html>.
- [fuj] From UML to Java and Back Again: The Fujaba homepage. www.upb.de/cs/isileit.
- [GHV03] S. Gyapay, R. Heckel, and D. Varró. Graph Transformation with Time. *Fundamenta Informaticae*, 58(1):1–22, November 2003.
- [GJ79] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co., New York, NY, USA, 1979.
- [GJM91] C. Ghezzi, M. Jazayeri, and D. Mandrioli. *Fundamentals of Software Engineering*. Prentice Hall Int., 1991.
- [GK01] M. Gogolla and C. Kobryn (eds.). *UML 2001 - The Unified Modeling Language, Modeling Languages, Concepts, and Tools, 4th International Conference, Toronto, Canada, October 1-5, 2001, Proceedings*, volume 2185 of *Lecture Notes in Computer Science*. Springer, 2001.
- [GKM98] R. Geisler, M. Klar, and S. Mann. Precise UML Semantics Through Formal Metamodeling. In L. Andrade, A. Moreira, A. Deshpande, and S. Kent (eds.), *Proceedings of the OOPSLA'98 Workshop on Formalizing UML. Why? How?*, 1998.

- [GKP98] R. Geisler, M. Klar, and C. Pons. Dimensions and Dichotomy in Metamodeling. Technical Report Bericht-Nr 98-5, TU Berlin, 1998.
- [GPP98] M. Gogolla and F. Parisi-Presicc. State Diagrams in UML: A Formal Semantics using Graph Transformation. In M. Broy, D. Coleman, T. S. E. Maibaum, and B. Rumpe (eds.), *Proceedings PSMT'98 Workshop on Precise Semantics for Modeling Techniques*, pages 55–72. Technische Universität München, 1998.
- [GPTdB93] M. Gemis, J. Paredaens, I. Thyssens, and J. V. den Bussche. GOOD: a graph-oriented object database system. In P. Buneman and S. Jajodia (eds.), *Proceedings of the 1993 ACM SIGMOD International Conference on the Management of Data*, volume 22(2) of *ACM SIGMOD Record*, pages 505–510, 1993.
- [GR99] M. Gogolla and M. Richters. Transformation Rules for UML Class Diagrams. In J. Bézivin and P.-A. Muller (eds.), *The Unified Modeling Language, UML'98 - Beyond the Notation. First International Workshop, Mulhouse, France, June 1998, Selected Papers*, volume 1618 of *LNCIS*, pages 92–106. Springer, 1999.
- [GR01] M. Große-Rhode. Formal Concepts for an Integrated Internal Model of the UML. In *UNIGRA 2001, Uniform Approaches to Graphical Process Specification Techniques (a Satellite Event of ETAPS 2001)*, volume 44(4) of *Electronic Notes in Theoretical Computer Science (ENTCS)*. Elsevier, 2001.
- [GSCK04] J. Greenfield, K. Short, S. Cook, and S. Kent. *Software Factories: Assembling Applications with Patterns, Models, Frameworks, and Tools*. Wiley, 2004.
- [GZK02] M. Gogolla, P. Ziemann, and S. Kuske. Towards an Integrated Graph Based Semantics for UML. In P. Bottoni and M. Minas (eds.), *Proc. ICGT Workshop Graph Transformation and Visual Modeling Techniques (GT-VMT'2002)*, *Electronic Notes in Theoretical Computer Science (ENTCS)*. Elsevier, October 2002.
- [Hau01] J. H. Hausmann. Dynamische Metamodellierung zur Spezifikation einer operationalen Semantik von UML. Master's thesis, Universität Paderborn, 2001.
- [Hau03] J. H. Hausmann. Metamodelling Relations - Relating Metamodels. In *Proceedings of the Metamodeling for MDA Workshop 2003*, Nov. 2003.
- [HG97] D. Harel and E. Gery. Executable Object Modeling with Statecharts. *Computer*, 30(7):31–42, 1997.
- [HH04] R. Heckel and J. H. Hausmann. What's new in UML 2? Challenges and Solutions for Model-Driven Development. Tutorial at the 25th International Conference on Application and Theory of Petr Nets, Bologna, Italy, June 2004.

- [HH05] R. Heckel and J. H. Hausmann. UML 2 - Neue Chancen, neue Probleme. Tutorial at the Software Engineering 2005, Essen, Germany, March 2005.
- [HHB02] R. Hennicker, H. Hussmann, and M. Bidoit. On the Precise Meaning of OCL Constraints. In T. Clark and J. Warmer (eds.), *Object Modeling with the OCL: The Rationale behind the Object Constraint Language*, pages 69–84. Springer, 2002.
- [HHS00] J. H. Hausmann, R. Heckel, and S. Sauer. Ein Konzept zur anwendungsbezogenen UML-Semantikbeschreibung durch dynamische Metamodellierung. In H. Giese and S. Philippi (eds.), *Proc. 8th GROOM Workshop: Visuelle Verhaltensmodellierung verteilter und nebenläufiger Softwaresysteme (VVVNS 2000)*, November 13-14, 2000, Münster, Germany, pages 64–69. Fachbereich Mathematik - Informatik, Westfälische Wilhelms-Universität Münster, 2000.
- [HHS01] J. H. Hausmann, R. Heckel, and S. Sauer. Towards Dynamic Meta Modeling of UML Extensions: An Extensible Semantics for UML Sequence Diagrams. In Minas and Bottoni [MB01], pages 80–87.
- [HHS02a] J. H. Hausmann, R. Heckel, and S. Sauer. Dynamic Meta Modeling with Time: Specifying the Semantics of Multimedia Sequence Diagrams. In P. Bottoni and M. Minas (eds.), *GT-VMT'2002 Graph Transformation and Visual Modeling Techniques, Barcelona, Spain, 11-12 October 2002*, volume 72(3) of *ENTCS*. Elsevier, 2002.
- [HHS02b] J. H. Hausmann, R. Heckel, and S. Sauer. Extended Model Relations with Graphical Consistency Conditions. In *In Proceedings UML 2002 Workshop on Consistency Problems in UML-based Software Development, Bleckinge Institute of Technology, Research Report 2002:06*, pages 61–74, 2002.
- [HHS04] J. H. Hausmann, R. Heckel, and S. Sauer. Dynamic Meta Modeling with Time: Specifying the semantics of multimedia sequence diagrams. *Software and Systems Modeling*, 3(3):181–193, 2004.
- [HHT96] A. Habel, R. Heckel, and G. Taentzer. Graph Grammars with Negative Application Conditions. *Fundamenta Informaticae*, 26(3/4):287–313, 1996.
- [HK03] J. H. Hausmann and S. Kent. Visualizing Model Mappings in UML. In *Proc. of the ACM Symposium on Software Visualization 2003*, 2003.
- [HK04] D. Harel and H. Kugler. The RHAPSODY Semantics of Statecharts (or, On the Executable Core of the UML). In *Integration of Software Specification Techniques for Application in Engineering*, volume 3147 of *Lecture Notes in Computer Science*, pages 325–354. Springer-Verlag, 2004.

- [HKS01] J. H. Hausmann, J. M. Küster, and S. Sauer. Identifying Semantic Dimensions of (UML) Sequence Diagrams. In A. Evans, R. France, A. Moreira, and B. Rumpe (eds.), *Practical UML-Based Rigorous Development Methods - Countering or Integrating the eXtremists. Workshop of the pUML-Group held together with the UML 2001 October 1st, 2001 in Toronto, Canada*, volume P-7 of *LNI*, pages 142–157. German Informatics Society, 2001.
- [HKT02] R. Heckel, J. Küster, and G. Taentzer. Confluence of typed attributed graph transformation systems. In *Graph Transformation, First International Conference, ICGT 2002, Barcelona, Spain, October 7-12, 2002, Proceedings*, volume 2505 of *Lecture Notes in Computer Science*, pages 161–176. Springer-Verlag, 2002.
- [HMTW95] R. Heckel, J. Muller, G. Taentzer, and A. Wagner. Attributed graph transformations with controlled application of rules. In *Proc. Colloquium on Graph Transformation and its Application in Computer Science*, 1995.
- [Hof05] B. Hoffmann. Graph Transformation with Variables. In U. M. Hans-Jörg Kreowski, F. Orejas, G. Rozenberg, and G. Taentzer (eds.), *Formal Methods in Software and System Modeling (Festschrift for Hartmut Ehrig on the Occasion of his 60th Birthday)*, volume 3393 of *Lecture Notes in Computer Science*, pages 101 – 115. Springer-Verlag, 2005.
- [Hor99] I. Horrocks. *Constructing the User Interface with Statecharts*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.
- [HP01] A. Habel and D. Plump. Computational Completeness of Programming Languages Based on Graph Transformation. In F. Honsell and M. Miculan (eds.), *FoSSaCS*, volume 2030 of *Lecture Notes in Computer Science*, pages 230–245. Springer, 2001.
- [HR00] D. Harel and B. Rumpe. Modeling Languages: Syntax, Semantics and All That Stuff - Part I: The Basic Stuff. Technical Report MCS00-16, Faculty of Mathematics and Computer Science, The Weizmann Institute of Science, Israel, September 2000.
- [HR04] D. Harel and B. Rumpe. Meaningful Modeling: What’s the Semantics of “Semantics”? *Computer*, 37(10):64–72, 2004.
- [HS99] B. Henderson-Sellers. OML: Proposals to Enhance UML. In Bézivin and Muller [BM98], pages 349–364.
- [HS01] B. Henderson-Sellers. Some Problems with the UML V1.3 Metamodel. In R. H. Sprague, Jr. (ed.), *Proc. 34th Annual Hawaii International Conference on System Sciences (HICSS-34)*. IEEE Computer Society, 2001.
- [HW95] R. Heckel and A. Wagner. Ensuring Consistency of Conditional Graph Grammars – A constructive Approach. In *Proc. of SEGRAGRA’95 “Graph Rewriting and Computation”*, volume 2 of *Electronic Notes in Theoretical Computer Science (ENTCS)*, 1995.

- [HZ01] R. Heckel and A. Zündorf. How to specify a graph transformation approach: A meta model for FUJABA. In H. Ehrig and J. Padberg (eds.), *Uniform Approaches to Graphical Process Specification Techniques, satellite workshop of ETAPS 2001, Genova, Italy*, 2001.
- [ITU93] ITU-T, Geneva. *Recommendation Z.120: Message Sequence Chart (MSC)*, 1993.
- [Kas91] U. Kastens. Attributed Grammars as a Specification Method. In H. Alblas and B. Melichar (eds.), *Attribute Grammars, Applications and Systems*, volume 545 of *Lecture Notes in Computer Science*, pages 16–47. Springer, 1991.
- [KC99] S.-K. Kim and D. Carrington. Formalizing the UML Class Diagram Using Object-Z. In R. France and B. Rumpe (eds.), *UML'99 - The Unified Modeling Language. Beyond the Standard. Second International Conference, Fort Collins, CO, USA, October 28-30, 1999, Proceedings*, volume 1723 of *LNCS*, pages 83–98. Springer, 1999.
- [KC00a] S.-K. Kim and D. Carrington. A Formal Mapping between UML Models and Object-Z Specifications. *Lecture Notes in Computer Science*, 1878:2–21, 2000.
- [KC00b] S.-K. Kim and D. Carrington. An Integrated Framework with UML and Object-Z for Developing a Precise Specification. In N.N. (ed.), *Proceedings of APSEC 2000*. IEEE Computer Society, 2000.
- [KC00c] S.-K. Kim and D. Carrington. UML Metamodel Formalization with Object-Z: The State Machine Package. Technical Report No 00-29, University of Queensland, 2000.
- [Ken97] S. Kent. Constraint Diagrams: Visualizing Invariants in OO Modelling. In *Proceedings of OOPSLA97*, pages 327–341. ACM Press, October 1997.
- [KER99] S. Kent, A. Evans, and B. Rumpe. UML Semantics FAQ. In *ECOOP'99 Workshop Reader*. Springer Verlag, LNCS, December 1999.
- [KGKK02] S. Kuske, M. Gogolla, R. Kollmann, and H.-J. Kreowski. An Integrated Semantics for UML Class, Object and State Diagrams Based on Graph Transformation. In *IFM '02: Proceedings of the Third International Conference on Integrated Formal Methods*, pages 11–28, London, UK, 2002. Springer-Verlag.
- [KGR99] S. Kent, S. Gaito, and N. Ross. A Meta-model Semantics for Structural Constraints in UML. In Kilov et al. [KRS99], chapter 9, pages 123–141.
- [KHH⁺97] S. Kent, A. Hamie, J. Howse, F. Civello, and R. Mitchell. Semantics Through Pictures: towards a diagrammatic semantics for object-oriented modelling notations. In *Proceedings of*

- ECOOP'97 Workshop on Precise Semantics for Object-Oriented Modelling Techniques*, Technical Report TUM-I9725. University of Munich, June 1997.
- [kmf] The Kent Modeling Framework. www.cs.ukc.ac.uk/kmf.
- [KMR02] A. Knapp, S. Merz, and C. Rauh. Model Checking Timed UML State Machines and Collaborations. In W. Damm and E.-R. Olderog (eds.), *7th Intl. Symp. Formal Techniques in Real-Time and Fault Tolerant Systems (FTRTFT 2002)*, volume 2469 of *Lecture Notes in Computer Science*, pages 395–414, Oldenburg, Germany, September 2002. Springer-Verlag.
- [KN03] M. Kardos and U. Nickel. ASMs as Integration Platform towards Verification and Validation of Distributed Production Control Systems at Multiple Levels of Abstraction. In E. Börger, A. Gargantini, and E. Riccobene (eds.), *Abstract State Machines*, volume 2589 of *Lecture Notes in Computer Science*, page 416. Springer, 2003.
- [Kna99] A. Knapp. A Formal Semantics for UML Interactions. In R. France and B. Rumpe (eds.), *UML'99 - The Unified Modeling Language. Beyond the Standard. Second International Conference, Fort Collins, CO, USA, October 28-30, 1999, Proceedings*, volume 1723 of *LNCS*, pages 116–130. Springer, 1999.
- [KNS92] G. Keller, M. Nttgens, and A.-W. Scheer. Semantische Prozedurmodellierung auf der Grundlage "Ereignisgesteuerter Prozedurketten (EPK)". Arbeitsbericht Heft 89, Institut für Wirtschaftsinformatik Universität Saarbrücken, 1992.
- [Kob04] C. Kobryn. UML 3.0 and the future of modeling. *Software and Systems Modeling*, 3(1):4–8, 2004.
- [KRS99] H. Kilov, B. Rumpe, and I. Simmonds (eds.). *Behavioral Specifications of Businesses and Systems*, volume 523 of *The Kluwer International Series in Engineering and Computer Science*. Kluwer Academic Publishers, 1999.
- [KS02] U. Kastens and C. Schmidt. VL-Eli: A Generator for Visual Languages - System Demonstration. *Electronic Notes in Theoretical Computer Science*, 65(3), 2002.
- [Kue05] T. Kuehne. What is a Model? [online]. In J. Bezivin and R. Heckel (eds.), *Language Engineering for Model-Driven Software Development*, number 04101 in Dagstuhl Seminar Proceedings. Internationales Begegnungs- und Forschungszentrum (IBFI), Schloss Dagstuhl, Germany, 2005.
- [Küh05] T. Kühne. Understanding metamodeling. In G.-C. Roman, W. G. Griswold, and B. Nuseibeh (eds.), *ICSE*, pages 716–717. ACM, 2005.

- [Kus00] S. Kuske. *Transformation Units—A structuring Principle for Graph Transformation Systems*. PhD thesis, University of Bremen, 2000.
- [Kus01] S. Kuske. A Formal Semantics of UML State Machines Based on Structured Graph Transformation. In M. Gogolla and C. Kobryn (eds.), *UML 2001 - The Unified Modeling Language. Modeling Languages, Concepts, and Tools. 4th International Conference, Toronto, Canada, October 2001, Proceedings*, volume 2185 of *Lecture Notes in Computer Science (LNCS)*, pages 241–256. Springer, 2001.
- [Küs04] J. Küster. *Consistency Management of Object-Oriented Behavioral Models*. PhD thesis, Universität Paderborn, 2004.
- [KW01] A. Kleppe and J. Warmer. Unification of Static and Dynamic Semantics of UML. Klasse Objecten Whitepaper, 2001.
- [Kwo00] G. Kwon. Rewrite rules and Operational Semantics for Model Checking UML Statecharts. In A. Evans, S. Kent, and B. Selic (eds.), *UML 2000 - The Unified Modeling Language. Advancing the Standard. Third International Conference, York, UK, October 2000, Proceedings*, volume 1939 of *LNCS*, pages 528–540. Springer, 2000.
- [LB93] M. Löwe and M. Beyer. AGG — An Implementation of Algebraic Graph Rewriting. In *Proceedings of the Fifth International Conference on Rewriting Techniques and Applications, '93*, volume 690 of *Lecture Notes in Computer Science (LNCS)*, pages 451–456, 1993.
- [LB99] K. Lano and J. Bicarregui. Semantics and Transformations for UML Models. In Bézivin and Muller [BM98], pages 107–119.
- [LE90] M. Löwe and H. Ehrig. Algebraic Approach to Graph Transformation Based on Single Pushout Derivations. In R. H. Möhring (ed.), *WG*, volume 484 of *Lecture Notes in Computer Science*, pages 338–353. Springer, 1990.
- [LKW93] M. Löwe, M. Korff, and A. Wagner. An Algebraic Framework for the Transformation of Attributed Graphs. In M. R. Sleep, M. J. Plasmeijer, and M. van Eekelen (eds.), *Term Graph Rewriting: Theory and Practice*, chapter 14, pages 185–199. John Wiley & Sons Ltd, 1993.
- [LLH02] Z. Liu, X. Li, and J. He. Using Transition Systems to Unify UML Models. In *ICFEM '02: Proceedings of the 4th International Conference on Formal Engineering Methods*, pages 535–547, London, UK, 2002. Springer-Verlag.
- [LLH04] X. Li, Z. Liu, and J. He. A formal semantics of UML sequence diagrams. In *Proc. of Australian Software Engineering Conference (ASWEC'2004), 13-16 April 2004*, Melbourne, Australia, 2004. IEEE Computer Society.

- [LM93] I. Litovsky and Y. Métivier. Computing with Graph Rewriting Systems with Priorities. *Theoretical Computer Science*, 115(2):191–224, 1993.
- [LMS95] I. Litovsky, Y. Metivier, and E. Sopena. Different Local Controls for Graph Relabeling Systems. *Mathematical Systems Theory*, 28(1):41–65, 1995.
- [Löw93] M. Löwe. Algebraic approach to single-pushout graph transformation. *Theoretical Computer Science*, 109:181–224, 1993.
- [LPP99] J. Lilius and I. Porres Paltor. The Semantics of UML State Machines. Technical Report 273, TUCS - Turku Centre for Computer Science, Turku, Finland, Jun 1999.
- [Lud03] J. Ludewig. Models in Software Engineering. *Software and Systems Modeling*, 2(1):5–14, 2003.
- [MB01] M. Minas and P. Bottoni (eds.). *2002 IEEE CS International Symposium on Human-Centric Computing Languages and Environments (HCC 2001), September 5-7, 2001 Stresa, Italy*. IEEE Computer Society, 2001.
- [MB02] S. J. Mellor and M. Balcer. *Executable UML: A Foundation for Model-Driven Architectures*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.
- [Mey97] B. Meyer. UML- The Positive Spin. *American Programmer*, 1997.
- [Min04] M. Minas (ed.). *Proceedings of the International Workshop on Visual Languages and Formal Methods (VLFM 04)*, 2004.
- [MK00] M. Minas and O. Köth. Generating Diagram Editors with DIAGEN. In *Proc. Applications of Graph Transformations With Industrial Relevance (AGTIVE), Kerkrade (The Netherlands), September 1–3, 1999*, volume 1779 of *Lecture Notes in Computer Science (LNCS)*, pages 433–440. Springer-Verlag, 2000.
- [MLB76] M. Marcotty, H. Ledgard, and G. V. Bochmann. A Sampler of Formal Definitions. *ACM Comput. Surv.*, 8(2):191–276, 1976.
- [Mon70] U. Montanari. Separable Graphs, Planar Graphs and Web Grammars. *Information and Control* 16, pages 243–267, 1970.
- [Mor99] B. Morand. Modeling: Is it Turning Informal into Formal? In Bézivin and Muller [BM98], pages 37–48.
- [Mos92] P. D. Mosses. *Action semantics*. Cambridge University Press, New York, NY, USA, 1992.
- [Mos96] P. D. Mosses. Theory and Practice of Action Semantics. In *MFCS '96, Proc. 21st Int. Symp. on Mathematical Foundations of Computer Science (Cracow, Poland, Sept. 1996)*, volume 1113 of *Lecture Notes in Computer Science*, pages 37–61. Springer-Verlag, 1996.

- [Mos00] P. D. Mosses. Modularity in Meta-Languages. Report Series RS-00-50, Basic Reserach in Computer Science BRICS, Aarhus, DK, December 2000.
- [Mos01] P. D. Mosses. The Varieties of Programming Language Semantics. In D. Bjørner, M. Broy, and A. V. Zamulin (eds.), *Ershov Memorial Conference*, volume 2244 of *Lecture Notes in Computer Science*, pages 165–190. Springer, 2001.
- [Mos03] P. D. Mosses. Fundamental Concepts and Formal Semantics of Programming Languages. Lecture Notes http://wiki.daimi.au.dk/dSprogSem-02/fundamental_concepts_and_.wiki, BRICS & Department of Computer Science, University of Aarhus, Denmark, Nov 2003.
- [Mos04a] P. D. Mosses. Action Semantics an example of language description engineering . Dagstuhl Seminar 04101 on Model-Driven Language Engineering, <http://www.dagstuhl.de/files/Materials/04/04101/04101.MossesPeter.Slides.pdf>, March 2004.
- [Mos04b] P. D. Mosses. Modular Structural Operational Semantics. *Journal of Logic and Algebraic Programming*, 60–61:195–228, 2004.
- [MSW00] M. Münch, A. Schürr, and A. J. Winter. Integrity Constraints in the Multi-paradigm Language PROGRES. In *TAGT'98: Selected papers from the 6th International Workshop on Theory and Application of Graph Transformations*, pages 338–351, London, UK, 2000. Springer-Verlag.
- [Mül96] J. Müller. On Termination of Single-Pushout Graph Rewriting. Technical Report 96-38, TU Berlin, 1996.
- [MV93] M. Minas and G. Viehstaedt. Specification of Diagram Editors Providing Layout Adjustments. In E. P. Glinert and K. A. Olsen (eds.), *Proc. IEEE Symp. Visual Languages, VL*, pages 324–329. IEEE Computer Society, 24–27 1993.
- [NLS⁺02] E. D. Nitto, L. Lavazza, M. Schiavoni, E. Tracanella, and M. Trombetta. Deriving executable process descriptions from UML. In *ICSE '02: Proceedings of the 24th International Conference on Software Engineering*, pages 155–165, New York, NY, USA, 2002. ACM Press.
- [Obe03] I. Ober. An ASM semantics for UML Derived from the meta-model and incorporating actions. In *Abstract State Machines - Advances in Theory and Applications.*, volume 2589 of *LNCS*. Proceedings 10th International Workshop, ASM 2003, 2003.
- [Obj00] Object Management Group. UML 2.0 Superstructure RFP. OMG document ad/00-09-02, 2000.
- [Obj01] Object Management Group. UML Specification Version 1.4. <http://www.omg.org/cgi-bin/doc?formal/01-09-67>, September 2001.

- [Obj02a] Object Management Group. Meta Object Facility (MOF) Specification, Version 1.4, 2002.
- [Obj02b] Object Management Group. MOF 2.0 Query / Views / Transformations RfP, 2002.
- [Obj03a] Object Management Group. Meta Object Facility (MOF) 2.0 Core Specification. <http://www.omg.org/cgi-bin/doc?ptc/2003-10-04>, 10 2003.
- [Obj03b] Object Management Group. UML 2.0 Infrastructure, 2003.
- [Obj03c] Object Management Group. UML 2.0 OCL Final Adopted Specification. <http://www.omg.org/cgi-bin/doc?ptc/2003-10-14>, 10 2003.
- [Obj03d] Object Management Group. UML 2.0 Superstructure Specification -final adopted specification-, 10 2003.
- [Obj03e] Object Management Group. UML Version 1.5 Specification. <http://www.omg.org/cgi-bin/doc?formal/03-03-01>, 03 2003.
- [Obj04] Object Management Group. UML 2.0 Superstructure - FTF convenience document-. <http://www.omg.org/cgi-bin/doc?ptc/2004-10-02>, 10 2004.
- [Obj05] Object Management Group. UML 2.0 Superstructure- Public Specification-. OMG document formal/05-07-04, August 2005.
- [Ode97] J. Odell. Standardization for OO A&D? *Distributed Computing*, 1, 1997.
- [Øve98] G. Øvergaard. A Formal Approach to Relationships in The Unified Modeling Language. In M. Broy, D. Coleman, T. S. E. Maibaum, and B. Rumpe (eds.), *Proceedings PSMT'98 Workshop on Precise Semantics for Modeling Techniques*. Technische Universität München, TUM-I9803, 1998.
- [Øve99] G. Øvergaard. A Formal Approach to Collaborations in the Unified Modeling Language. In R. France and B. Rumpe (eds.), *UML'99 - The Unified Modeling Language. Beyond the Standard. Second International Conference, Fort Collins, CO, USA, October 28-30. 1999, Proceedings*, volume 1723 of *LNCIS*, pages 99–115. Springer, 1999.
- [Øve00] G. Øvergaard. Using the BOOM framework for formal specification of the UML. In *Proc. ECOOP Workshop on Defining Precise Semantics for UML*, 2000.
- [Pad82] P. Padawitz. Graph Grammars and Operational Semantics. *Theoretical Computer Science*, 19:117–141, 1982.
- [Pen03] T. Pender. *UML Bible*. Wiley & Sons, 2003.
- [Pet62] C. A. Petri. *Kommunikation mit Automaten*. PhD thesis, Bonn: Institut für Instrumentelle Mathematik, Schriften des IIM Nr. 2, 1962.

- [PL99] I. P. Paltor and J. Lilius. vUML: A Tool for Verifying UML Models. In R. J. Hall and E. Tyugu (eds.), *Proc. of the 14th IEEE International Conference on Automated Software Engineering, ASE'99*. IEEE, 1999.
- [Plo81] G. Plotkin. A structural approach to operational semantics. Technical Report DAIMI FN-19, Aarhus University, Computer Science Department, 1981.
- [Plo04] G. D. Plotkin. The Origins of Structural Operational Semantics. *Journal of Functional and Logic Programming*, 60-61:3–15, 2004.
- [Plu98] D. Plump. Termination of Graph Rewriting is Undecidable. *Fundam. Inform.*, 33(2):201–209, 1998.
- [POB02] R. Paige, J. Ostroff, and P. Brooke. Checking the Consistency of Collaboration and Class Diagrams using PVS. In *Proc. Fourth Workshop on Rigorous Object-Oriented Methods (ROOM4)*. British Computer Society, London,, 2002.
- [Por01] I. Porres. *Modeling and Analyzing Software Behavior in UML*. PhD thesis, Turku Centre for Computer Science, Nov 2001.
- [PR69] J. L. Pfaltz and A. Rosenfeld. Web Grammars. *Int. Joint Conference on Artificial Intelligence*, pages 609–619, 1969.
- [PS04] D. Plump and S. Steinert. Towards Graph Programs for Graph Algorithms. In H. Ehrig, G. Engels, F. Parisi-Presicce, and G. Rozenberg (eds.), *Graph Transformations, Second International Conference, ICGT 2004, Rome, Italy, September 28 - October 2, 2004, Proceedings*, volume 3256 of *Lecture Notes in Computer Science*, pages 128–143. Springer-Verlag, 2004.
- [QVT] QVT Partners. Submission for MOF 2.0 Query/Views/Trasformations RFP. OMG document ad/2003-08-08.
- [RA01] G. Reggio and E. Astesiano. A Proposal of a Dynamic Core for UML Metamodelling with MML. Technical Report DISI-TR-01-1, DISI, Universita di Genova, Italy, 2001.
- [RACH99] G. Reggio, E. Astesiano, C. Choppy, and H. Hussmann. A CASL Formal Definition of UML Active Classes and Associated State Machines. Technical Report DISI-TR-99-16, DISI – Universita di Genova, Italy, 1999.
- [RACH00] G. Reggio, E. Astesiano, C. Choppy, and H. Hussmann. Analysing UML Active Classes and Associated State Machines – A Lightweight Formal Approach. In T. Maibaum (ed.), *Proc. Fundamental Approaches to Software Engineering (FASE 2000), Berlin, Germany*, volume 1783 of *Lecture Notes in Computer Science (LNCS)*, pages 127–146. Springer-Verlag, March/April 2000.
- [RCA01] G. Reggio, M. Cerioli, and E. Astesiano. Towards a Rigorous Semantics of UML Supporting Its Multiview Approach. In

- H. Hussmann (ed.), *Fundamental Approaches to Software Engineering, 4th International Conference, FASE 2001, Proceedings*, volume 2029 of *Lecture Notes in Computer Science (LNCS)*, pages 171–186. Springer, 2001.
- [Reg02] G. Reggio. Metamodelling Behavioral Aspects: The case of the UML State Machines. In *Proc. of the 5th World Conference on Integrated Design and Process Technology (IDPT 2002)*. Society for Process Technology, 2002.
- [Rei85] W. Reisig. *Petri nets: an introduction*. Springer-Verlag New York, Inc., New York, NY, USA, 1985.
- [Ren03a] A. Rensink. GROOVE: A Graph Transformation Tool Set for the Simulation and Analysis of Graph Grammars. Available at <http://www.cs.utwente.nl/~groove>, 2003.
- [Ren03b] A. Rensink. Towards Model Checking Graph Grammars. In M. Leuschel, S. Gruner, and S. L. Presti (eds.), *Workshop on Automated Verification of Critical Systems (AVoCS)*, Technical Report DSSE-TR-2003-2, pages 150–160. University of Southampton, 2003.
- [Ren04a] A. Rensink. The GROOVE Simulator: A Tool for State Space Generation. In J. Pfalz, M. Nagl, and B. Böhlen (eds.), *Applications of Graph Transformations with Industrial Relevance (AGTIVE)*, volume 3062 of *Lecture Notes in Computer Science*, pages 479–485. Springer-Verlag, 2004.
- [Ren04b] A. Rensink. Representing First-Order Logic using Graphs. In H. Ehrig, G. Engels, F. Parisi-Presicce, and G. Rozenberg (eds.), *Graph Transformations, Second International Conference, ICGT 2004, Rome, Italy, September 28 - October 2, 2004, Proceedings*, volume 3256 of *Lecture Notes in Computer Science*, pages 319–335. Springer-Verlag, 2004.
- [Ren04c] A. Rensink. Time and Space Issues in the Generation of Graph Transition Systems. In *International Workshop on Graph-Based Tools (GraBaTs)*, Electronic Notes in Theoretical Computer Science. Elsevier Science Publishers, 2004.
- [RG00] M. Richters and M. Gogolla. Validating UML Models and OCL Constraints. In A. Evans, S. Kent, and B. Selic (eds.), *UML 2000 - The Unified Modeling Language. Advancing the Standard. Third International Conference, York, UK, October 2000, Proceedings*, volume 1939 of *LNCS*, pages 265–277. Springer, 2000.
- [RH04] B. Rumpe and W. Hesse (eds.). *Modellierung 2004, Proceedings zur Tagung, 23.-26. März 2004, Marburg, Proceedings*, volume 45 of *LNI*. GI, 2004.
- [Rod98] P. Rodgers. A Graph Rewriting Programming Language for Graph Drawing. In *Proceedings of the 14th IEEE Symposium on Visual Languages, Halifax, Nova Scotia, Canada*. IEEE, IEEE Computer Society Press, September 1998.

- [Rod00] P. Rodgers. Constructs for Programming with Graph Rewrites. In H. Ehrig and G. Taentzer (eds.), *GRATRA 2000: Joint APPLIED GRAPH and GETGRATS Workshop on Graph Transformation Systems*, pages 59–66, March 2000.
- [Roz97] G. Rozenberg (ed.). *Handbook of Graph Grammars and Computing by Graph Transformation. Volume 1: Foundations*. World Scientific, 1997.
- [RS97] J. Rekers and A. Schürr. Defining and parsing visual languages with layered graph grammars. *Journal of Visual Languages and Computing*, 8(1):27–55, 1997.
- [RSV04] A. Rensink, A. Schmidt, and D. Varró. Model Checking Graph Transformations: A Comparison of Two Approaches. In *Proc. ICGT 2004: Second International Conference on Graph Transformation*, Rome, Italy, September 2004. Springer.
- [RV00] P. J. Rodgers and N. Vidal. Graph Algorithm Animation with Grrr. In *Proc. Applications of Graph Transformations With Industrial Relevance (AGTIVE), Kerkrade (The Netherlands), September 1–3, 1999*, volume 1779 of *Lecture Notes in Computer Science (LNCS)*, pages 379–394. Springer-Verlag, 2000.
- [RW99] G. Reggio and R. Wieringa. Thirty one Problems in the Semantics of UML 1.3 Dynamics. In *OOPSLA '99 workshop "Rigorous Modelling and Analysis of the UML: Challenges and Limitations"*, 1999.
- [Sch77] H. J. Schneider. Graph Grammars. In M. Karpínski (ed.), *Fundamentals of Computation Theory*, volume 56 of *Lecture Notes in Computer Science*, pages 314–331, 1977.
- [Sch90] A. Schürr. Introduction to PROGRES, an Attribute Graph Grammar Based Specification Language. In M. Nagl (ed.), *Graph-Theoretic Concepts in Computer Science*, volume 411 of *Lecture Notes in Computer Science*, pages 151–165, 1990.
- [Sch91] A. Schürr. *Operational Specifications with Programmed Graph Rewriting Systems*. PhD thesis, RWTH Aachen, D-52056 Aachen, Germany, 1991.
- [Sch95] A. Schürr. PROGRES for Beginners, 1995.
- [Sch96a] D. A. Schmidt. On the need for a popular formal semantics. *ACM Computing Surveys*, 28(4es):175, 1996.
- [Sch96b] A. Schürr. Logic Based Programmed Structure Rewriting Systems. *Fundamenta Informaticae*, 26(3,4):363 – 386, 1996.
- [Sei01] E. Seidewitz. What do models mean. OMG document ad/03-03-31, March 2001.
- [Sel04] B. Selic. On the Semantic Foundations of Standard UML 2.0. In M. Bernardo and F. Corradini (eds.), *SFM*, volume 3185 of *Lecture Notes in Computer Science*, pages 181–199. Springer, 2004.

- [SG99] A. J. H. Simons and I. Graham. 30 Things That Go Wrong in Object Modelling with UML 1.3. In Kilov et al. [KRS99], pages 221–242.
- [SH05] H. Störrle and J. H. Hausmann. Towards a Formal Semantics of UML 2.0 Activities. In *Software Engineering 2005*, pages 117–128, Essen, 2005.
- [SJM04] R. V. D. Straeten, V. Jonckers, and T. Mens. Supporting Model Refactorings through Behaviour Inheritance Consistencies. In T. Baar, A. Strohmeier, A. Moreira, and S. J. Mellor (eds.), *UML 2004 - The Unified Modeling Language. Model Languages and Applications. 7th International Conference, Lisbon, Portugal, October 11-15, 2004, Proceedings*, volume 3273 of *LNCS*, pages 304–319. Springer, 2004.
- [SK02] F. Steimann and T. Kühne. A Radical Reduction of UML’s Core Semantics. In J.-M. Jézéquel, H. Hussmann, and S. Cook (eds.), *UML 2002 - The Unified Modeling Language. Model Engineering, Languages, Concepts, and Tools. 5th International Conference, Dresden, Germany, September/October 2002, Proceedings*, volume 2460 of *LNCS*, pages 34–48. Springer, 2002.
- [SLKN01] J. Sprinkle, Á. Lédeczi, G. Karsai, and G. Nordstrom. The New Metamodeling Generation. In *ECBS*, pages 275–. IEEE Computer Society, 2001.
- [SS71] D. Scott and C. Strachey. Towards a Mathematical Semantics for Computer Languages. In *Computers and Automata*, pages 19–46. Wiley, 1971.
- [Ste01] P. Stevens. On Use Cases and Their Relationships in the Unified Modelling Language. In H. Hussmann (ed.), *Fundamental Approaches to Software Engineering, 4th International Conference, FASE 2001, Proceedings*, volume 2029 of *LNCS*, pages 140–155. Springer, 2001.
- [Ste02] P. Stevens. On the interpretation of binary associations in the Unified Modelling Language. *Software and Systems Modeling*, 1(1):68–79, 2002.
- [Ste03] S. Steinert. Graph Programs for Graph Algorithms. Technical Report volume 7/03, University of Oldenburg, Fakultät II, Department für Informatik, 2003.
- [Ste04] F. Steimann. UML-A oder warum die Wissenschaft ihre eigene einheitliche Modellierungssprache haben sollte. In Rumpe and Hesse [RH04], pages 121–133.
- [Stö03] H. Störrle. Assert, Negate and Refinement in UML-2 Interactions. In *International Workshop on Critical Systems Development with UML (CSDUML03)*, pages 79–94, 2003.

- [Stö04a] H. Störrle. Semantics of Control-Flow in UML 2.0 Activities. In *IEEE Symposium on Visual Languages and Human-Centric Computing 2004*, pages 235–242, 2004.
- [Stö04b] H. Störrle. Semantics of Exceptions in UML 2.0 Activities. Technical Report 0403, University of Munich, 2004.
- [Stö04c] H. Störrle. Semantics of Structured Nodes in UML 2.0 Activities. In I. Porres (ed.), *Proceedings of NWUML'2004: The 2nd Nordic Workshop on UML, Modeling, Methods and Tools*, pages 19–32, 2004.
- [Stö05a] H. Störrle. Semantics and Verification of Data Flow in UML 2.0 Activities. In M. Minas (ed.), *Proceedings of the Workshop on Visual Languages and Formal Methods (VLFM 2004)*, volume 127(4) of *Electronic Notes in Theoretical Computer Science (ENTCS)*, pages 35–52. Elsevier, 2005.
- [Stö05b] H. Störrle. *UML 2 erfolgreich einsetzen*. Addison-Wesley, 2005.
- [Stö05c] H. Störrle. *UML 2 für Studenten*. Pearson Studium, 2005.
- [SV03] A. Schmidt and D. Varró. CheckVML: A Tool for Model Checking Visual Modeling Languages. In P. Stevens, J. Whittle, and G. Booch (eds.), *Proc. UML 2003: 6th International Conference on the Unified Modeling Language*, volume 2863 of *LNCS*, pages 92–95, San Francisco, CA, USA, October 20–24 2003. Springer.
- [SWZ95] A. Schürr, A. Winter, and A. Zündorf. Graph Grammar Engineering with PROGRES. In Botella and Schäfer (eds.), *ESEC'95 Proceedings of the 5th European Software Engineering Conference*, volume 989 of *Lecture Notes in Computer Science*, pages 219–234, Berlin, 1995. Springer-Verlag.
- [SWZ99] A. Schürr, A. Winter, and A. Zündorf. The PROGRES Approach: Language and Environment. In Engels et al. [EEKR99], pages 487–550.
- [Tae92] G. Taentzer. Parallel High-Level Replacement Systems. Technical Report 92/10, TU Berlin, 1992.
- [Tae96] G. Taentzer. *Parallel and Distributed Graph Transformation: Formal Description and Application to Communication-Based Systems*. PhD thesis, TU Berlin, 1996.
- [Tai97] A. Taivalsaari. Classes Versus Prototypes: Some Philosophical and Historical Observations. *JOOP*, 10(7):44–50, 1997.
- [Tar44] A. Tarski. The semantic conception of truth and the foundations of semantics. *Philosophy and Phenomenological Research*, 4, 1944.
- [TB94] G. Taentzer and M. Beyer. Amalgamated Graph Transformations and Their Use for Specifying AGG — an Algebraic Graph Grammar System. In H. J. Schneider and H. Ehrig (eds.), *Graph*

- Transformations in Computer Science*, volume 776 of *Lecture Notes in Computer Science*, pages 380–394, 1994.
- [Tho03] D. Thomas. UML - Unified or Universal Modeling Language? UML2, OCL, MOF, EDOC - The Emperor Has Too Many Clothes. *Journal of Object Technology*, 2:7–12, 2003.
- [TR05] G. Taentzer and A. Rensink. Ensuring Structural Constraints in Graph-Based Models with Type Inheritance. In Cerioli [Cer05], pages 64–79.
- [TS95] G. Taentzer and A. Schürr. DIEGO, Another Step Towards a Module Concept for Graph Transformation Systems. In *Proc. of SEGRAGRA '95 "Graph Rewriting and Computation"*, volume 2 of *Electronic Notes in Theoretical Computer Science (ENTCS)*, 1995.
- [U2P03] U2Partners. U2 Partners' UML 2.0: Infrastructure, 3rd revised submission. <http://www.omg.org/cgi-bin/doc?ad/03-01-01>, 2003.
- [UML97] UML Partners. Unified Modeling Language v. 1.1. OMG document ad/97-08-11, August 1997.
- [Var02] D. Varró. A Formal Semantics of UML Statecharts by Model Transition Systems. In A. Corradini, H. Ehrig, H.-J. Kreowski, and G. Rozenberg (eds.), *Graph Transformation, First International Conference, ICGT 2002, Barcelona, Spain, October 7-12, 2002, Proceedings*, volume 2505 of *Lecture Notes in Computer Science (LNCS)*, pages 378–392, Barcelona, Spain, October 7–12 2002. Springer-Verlag.
- [Var03] D. Varró. *Automated Model Transformations for the Analysis of IT Systems*. PhD thesis, Budapest University of Technology and Economics, 2003.
- [Var04] D. Varró. Automated Formal Verification of Visual Modeling Languages by Model Checking. *Journal of Software and Systems Modeling*, 3(2):85–113, May 2004.
- [vdB01] M. van der Beeck. Formalization of UML-Statecharts. In M. Gogolla and C. Kobryn (eds.), *UML 2001 - The Unified Modeling Language. Modeling Languages, Concepts, and Tools. 4th International Conference, Toronto, Canada, October 2001, Proceedings*, volume 2185 of *Lecture Notes in Computer Science (LNCS)*, pages 406–421. Springer, 2001.
- [vdB02] M. von der Beeck. A structured operational semantics for UML-statecharts. *Software and Systems Modeling*, 1:130 – 141, 2002.
- [VP03] D. Varró and A. Pataricza. VPM: A visual, precise and multilevel metamodeling framework for describing mathematical domains and UML. *Journal of Software and Systems Modeling*, 2(3):187–210, October 2003.
- [WC90] J.-P. Wu and S. T. Chanson. Translation from LOTOS and Estelle Specifications to Extended Transition System and its Verifica-

- tion. In *FORTE '89: Proceedings of the IFIP TC/WG6.1 Second International Conference on Formal Description Techniques for Distributed Systems and Communication Protocols*, pages 533–549. North-Holland, 1990.
- [Wik05] Wikipedia. Language — Wikipedia, the free encyclopedia, 2005.
- [Win01] A. Winter. Exchanging Graphs with GXL. Technical Report 9–2001, Universität Koblenz-Landau, Institut für Informatik, Universität Koblenz-Landau, Institut für Informatik, Rheinau 1, D-56075 Koblenz, 2001.
- [WS97] A. Winter and A. Schürr. Modules and Updatable Graph Views for PROgrammed Graph REwriting Systems. Technical Report 97-3, RWTH Aachen, FG Informatik, October 1997.
- [XB02] F. Xie and J. C. Browne. Integrated State Space Reduction for Model Checking Executable Object-Oriented Software System Designs. In *FASE '02: Proceedings of the 5th International Conference on Fundamental Approaches to Software Engineering*, pages 64–79, London, UK, 2002. Springer-Verlag.
- [XLB01] F. Xie, V. Levin, and J. C. Browne. Model Checking for an Executable Subset of UML. In *ASE '01: Proceedings of the 16th IEEE International Conference on Automated Software Engineering*, page 333, Washington, DC, USA, 2001. IEEE Computer Society.
- [ZHG05] P. Ziemann, K. Hölscher, and M. Gogolla. From UML Models to Graph Transformation Systems. In M. Minas (ed.), *Proceedings of the Workshop on Visual Languages and Formal Methods (VLFM 2004)*, volume 127(4) of *Electronic Notes in Theoretical Computer Science*, pages 17–33. Elsevier Science, 2005.
- [Zün95] A. Zündorf. *Programmierte Graph-Ersetzungs-Systeme: Spezifikation, Implementierung und Anwendung in einer integrierten Entwicklungs-Umgebung*. PhD thesis, Fakultät für Mathematik, Naturwissenschaften und Informatik, Rheinisch-Westfälische Technische Hochschule Aachen, 1995.
- [Zün96] A. Zündorf. Graph Pattern Matching in PROGRES. In J. E. Cuny, H. Ehrig, G. Engels, and G. Rozenberg (eds.), *Proc. 5th Int. Workshop on Graph Grammars and their Application to Computer Science*, volume 1073 of *Lecture Notes in Computer Science*, pages 454–468. Springer-Verlag, 1996.

Index

- abstract syntax, 8
- abstract transitive closure, 66
- action node, 139, 209
- Action Semantics
 - by Mosses, 40
 - in UML, 41
- Activity Diagram
 - activity graphs, 132
 - control structures, 133
 - deficiencies, 125
 - difference to Petri nets, 131
 - evaluation semantics, 135
 - fork node problem, 125
 - history of, 207
 - interpretation by GROOVE, 187
 - offer semantics, 136
 - role in UML, 208
 - token flow, 127
 - traverse-to-completion semantics, 129
- Advanced Language User, 21
- attributed graph, 67
- big-step rule
 - formalization, 100
 - formulation of, 166
 - introduction, 90
- Bremen approach, 29
- carpenter deadlock example, 191
- central buffer node, 214
- CheckVML, 176
 - comparison to GROOVE, 177
- code generation, 24
- COMMA meta model, 15
- compilation semantics, 28
- concrete invocations, 100
- concrete semantics, 23
- concrete syntax, 8
- concrete transitive closure, 66
- conservative extensions
 - discussion of, 120
 - motivation, 111
 - restrictions for, 115
- control flow, 214
- control node, 139, 210
- core semantics, 31
- dangling edges, 71
- data store node, 214
- decision node, 210
- denotational meta modeling
 - dynamic extensions, 35
 - introduction, 32
 - usage in DMM, 105
 - usage in OCL, 34
- denotational semantics, 28
- derivation, 73
- derivation sequence, 80
- discrete behavior, 13
- DMM, *see* Dynamic Meta Modeling
- DMM interpreter, 175
 - requirements for, 194
- DMM package
 - definition, 113
 - guidelines for, 165
 - import, 113
 - merge, 113
- DMM rule
 - correspondence to operations, 108
 - difference to Communication Diagram, 117
 - formulation of, 168
 - translation to GROOVE rules, 182
- DMM specification
 - efficiency of, 158
 - modularity of, 158
 - semantic correctness, 155
 - syntactic correctness, 155
 - understandability of, 156
- DMM system, 100

- Double-Pushout approach, 71
- Dynamic Meta Modeling
 - adequacy of, 119
 - analyzability of, 118
 - architecture of, 116
 - introduction, 41
 - precision of, 118
 - underspecification in, 120
 - understandability of, 117
 - universality of, 119
- dynamic semantics, 11
- early precondition checking, 169
- edge, 140
- edge-label preserving graph morphism, 65
- extensional entity, 105
 - guidelines for formulation, 160
 - Relations to, 164
- final node, 212
- final rule, 95
- flows, 214
- fork node, 211
- fork node problem, 125
- GOS, *see* Graphical Operational Semantics
- grammar, 8
- graph
 - attributes in, 67
 - consistency conditions, 73
 - correspondence to UML, 69
 - graphical representation of, 69
 - in DMM, 68
 - labeled, 64
 - overview, 64
 - typed graphs, 65
- Graph Transformation
 - overview, 63
- Graph Transformation rule
 - definition of, 72
 - in DMM, 76
 - introduction, 70
 - presentation in DMM, 78
 - priorities in, 81
 - programmed GTs, 82
 - signature, 88
 - transformation units in, 81
 - triggers in, 81
 - with invocation, 95
- Graph Transformation System, 80
- Graphical Operational Semantics, 39
- GROOVE tool set
 - comparison to CheckVML, 177
 - components of, 178
 - Graph Transformation in, 181
 - introduction to, 176
 - merge embargo, 181
 - rule notion, 181
 - translation of DMM rules, 182
- GRRR system, 81
- GT, *see* Graph Transformation
- GTR, *see* Graph Transformation rule
- heuristics, 153
 - for efficiency, 158
 - for modularity, 158
 - for understandability, 156
- incompleteness of models, 39
- Indefinability Theorem, 10
- induction in SOS, 39
- inheritance clan, 66
- initial node, 212
- instance graph, 68
- intensional entity, 105
 - guidelines for formulation, 162
 - Relations to, 164
 - replication of, 106
- interpretation states, 109
- interpreter semantics, 38
 - in DMM, 107
- invocation
 - application of, 99
 - concrete invocation, 100
 - definition, 87
 - encoding in GROOVE, 184
 - fulfillment, 100
 - introduction, 82
 - matching of, 99
 - open invocation, 100
 - rule signature, 88
- join node, 211
- L-level, 33
- Labeled Transition System
 - definition, 109
 - generation of, 175

- state of, 109
- labels, 64
- language, 7
- Language Engineer, 21
- language extension concept of UML, 13
- language level, 33
- linguistic instance, 33
- LTS, *see* Labeled Transition System
- M-level, 33
- merge embargo edge, 181
- merge node, 211
- meta modeling
 - criticism of, 15
 - in DMM, 118
 - overview, 14
 - strict meta modeling, 18
- Meta Modeling Language, 35
- Meta Object Facility
 - criticism of, 16
 - introduction to, 15
- Meta Relation, *see* Relation
- meta-circularity, 16
- methodology, 153
- MML, *see* Meta Modeling Language
- model checking
 - of Graph Transformations, 175
- model level, 33
- model validation
 - by GROOVE, 190
 - of Activity Diagrams, 190
- Modular Operational Semantics, 40
- MOF, *see* Meta Object Facility
- NAC, *see* Negative Application Condition
- Negative Application Condition
 - in DMM, 77
 - in GROOVE, 181
 - introduction, 73
 - matching, 79
- nested mapping, 47
- Object Constraint Language, 14
- object flow, 214
- object node, 139, 212
- OCL, *see* Object Constraint Language
 - visualizations of, 62
- offer, 139
- open invocation, 100
- operational semantics
 - application to VMLs, 39
 - introduction, 37
 - usage in DMM, 107
- Pair
 - concrete syntax, 54
 - definition, 58
 - introduction, 46
- parameter, 213
- Petri nets, 129
- pin, 213
- premise rules, 92
- priorities, 81
- process, 153
- programmed Graph Transformation, 82
- PROGRES system, 82
- pUML group, 32
- quality of DMM specifications, 153
- Relation, 45
 - concepts of, 51
 - concrete syntax, 53
 - definition, 55
 - instantiation of, 59
 - introduction, 46
 - nested, 47
 - scope, 51
- relation pattern, 50
- requirements for semantics descriptions, 22
- rule graph, 68
- rule matching
 - definition, 72
 - injectivity in, 72
- rule schema
 - overview, 93
 - unfolding of, 96
 - with UQS and premises, 95
- scope of a nested Relation, 51
- semantic correctness, 155
- semantic domain, 10
- semantic domain meta model
 - concept of, 32
 - inheritance in the, 113
 - of Activity Diagrams, 140
- semantic gap, 22

- semantic mapping, 10
- semantic variation point
 - definition, 13
 - support in DMM, 121
- semantics
 - dynamic, 11
 - overview, 10
 - static, 11
- semantics definition
 - form and content, 19
 - requirements for, 22
- Single-Pushout approach, 71
- small-step rule
 - formulation of, 167
 - introduction, 90
- spawnpoint, 139
- static semantics, 11
- Story Diagram, 82
- Structured Operational Semantics, 37
- SVP, *see* semantic variation point
- syntactic correctness, 155

- termination analysis, 101
- test models, 190
- token, 138
- token flow, 127
- transformation units, 81
- translation semantics, 28
- traverse-to-completion, 129
- triggers, 81
- tuple, 57
- type graph
 - definition of, 65
 - in DMM, 68
 - inheritance in, 66

- UML, *see* Unified Modeling Language
- UML profile
 - introduction, 13
 - support by DMM, 120
- Unified Modeling Language
 - achievements, 12
 - behavioral diagrams, 123
 - characteristics of, 13
 - criticism of, 16
 - dissemination of, 21
 - language extensions, 13
 - overview, 12
 - problems in the definition of, 125
 - specification documents, 14

- Universally Quantified Structure
 - encoding in GROOVE, 183
 - in DMM, 79
 - in preconditions, 170
 - introduction, 74
 - restrictions of, 76
 - unfolding of, 79
 - vs. iteration, 168
- UQS, *see* Universally Quantified Structure

- Visual Modeling Language
 - abstract syntax, 8
 - characteristics of, 12
 - concrete syntax, 8
 - overview, 11
- VML, *see* Visual Modeling Language

- wildcard label, 64