Systematic Development of Model-based Software Engineering Methods

Stefan Sauer

sauer@s-lab.upb.de

Dissertation

zur Erlangung des Grades “Doktor der Naturwissenschaften” (Dr. rer. nat.)

Fakultät für Elektrotechnik, Informatik und Mathematik

Universität Paderborn

Paderborn, Januar 2011

Diese Dissertation wurde im Novemer 2010 bei der Fakultät für Elektrotechnik, Informatik und Mathematik der Universität Paderborn eingereicht. Sie wurde auf der Grundlage der Gutachten von der Promotionskommission angenommen. Die mündliche Prüfung fand am 20. Januar 2011 in Paderborn statt.

Gutachter:

- Prof. Dr. Gregor Engels
- Prof. Dr. Wilhelm Schäfer

Promotionskommission:

- Prof. Dr. Gregor Engels (Vorsitzender)
- Prof. Dr. Wilhelm Schäfer
- Prof. Dr. Hans Kleine Büning
- Prof. Dr. Franz J. Rammig
- Dr. Matthias Meyer

Danksagung

Auch wenn eine Dissertation eine individuelle Leistung darstellt, so ist sie gemeinhin doch das Ergebnis einer Zusammenarbeit und ohne die Unterstützung anderer Menschen nicht vorstellbar. So ist es auch in diesem Fall. Deshalb gebührt mein Dank all denen, die mich bei meiner wissenschaftlichen Arbeit und der Anfertigung dieser Dissertation unterstützt haben.

Meinen besonderen Dank möchte ich meinem Mentor, Doktorvater und langjährigen Wegbereiter Prof. Dr. Gregor Engels aussprechen. Seine kollegiale Art, sein Vertrauen und seine Unterstützung waren und sind wesentliche Erfolgsfaktoren für meine Arbeit. Ich danke ihm, dass er mir die Möglichkeit gegeben hat, in seiner Fachgruppe zu forschen und zu lehren, das s-lab – Software Quality Lab mit aufzubauen und insbesondere dafür, dass er mich über die gesamte Zeit und bis zuletzt über Höhen und Tiefen in vielfältiger Weise unterstützt hat, diese Dissertation zu einem erfolgreichen Abschluss zu bringen. Ich danke Prof. Dr. Wilhelm Schäfer für die langjährige konstruktive Zusammenarbeit und dass er die Rolle des Gutachters meiner Dissertation übernommen hat. Ich danke den Mitgliedern der Promotionskommission, der neben den beiden vorgenannten die Herren Prof. Dr. Hans Kleine Büning, Prof. Dr. Franz J. Rammig und Dr. Matthias Meyer angehörten, dass Sie mir diese Ehre erwiesen haben. Matthias Meyer danke ich auch, dass er mir in der „heißen Phase“ des Aufschreibens in der Geschäftsführung des s-lab „den Rücken freigehalten hat“.

Ich danke den Kolleginnen und Kollegen, mit denen ich im s-lab, in der Fachgruppe Datenbank- und Informationssysteme von Prof. Dr. Gregor Engels und in der Universität Paderborn in den vergangenen Jahren zusammenarbeiten durfte, für die interessante, freundschaftliche und angenehme Zusammenarbeit. Ich danke auch den Kolleginnen und Kollegen der nationalen und internationalen Wissenschaft, mit denen ich zusammengetroffen bin, und deren Impulse mich immer wieder einen Schritt weiter gebracht haben. Mein Dank gilt aber auch den Partnern des s-lab und ihren Mitarbeiterinnen und Mitarbeitern, mit denen ich in vielfältigen Projekten neue Erkenntnisse und Erfahrungen insbesondere aus der Sicht der Praxis sammeln konnte. Stellvertretend seien hier die Kolleginnen und Kollegen von Capgemini sd&m Research genannt, mit denen ich an der Spezifikationsmethodik für betriebliche Informationssysteme gearbeitet habe.

Neben dem wissenschaftlichen und beruflichen Umfeld trägt aber auch mein privates Umfeld, allen voran meine Familie, einen wesentlichen Anteil am erfolgreichen Abschluss dieser Dissertation. Ich danke meinen Eltern Gerhard und Ursula Sauer, dass sie mir mein Studium und den akademischen Werdegang ermöglicht haben. Der größte Dank gebührt aber meiner Frau Kerstin und meinen Kindern Bjarne und Louisa! Für ihre Toleranz, Hilfe und so manche Entbehrung! Ich danke Kerstin, dass sie die Geduld aufgebracht hat, mich auf diesem Weg zu begleiten und mir immer wieder mit jeglicher Art von Unterstützung zur Seite gestanden hat. Bjarne und Louisa danke ich, dass sie es akzeptiert haben, dass ihr Papa auch in seiner vermeintlichen Freizeit an der Dissertation gearbeitet hat, anstatt diese Zeit mit ihnen zu verbringen.

Abstract

The development of today's software systems demands for sophisticated software engineering processes and methods. Effort that is invested once in the methods can be systematically reused in projects. We use the term method engineering for the systematic development of software engineering methods.

In the first part of this thesis, I present and characterize software engineering methods and method engineering approaches that I have developed. They are particularly concerned with model-based and model-driven development of business information systems, multimedia and advanced interactive systems, and some related domains. Two formal methods that are used in the software engineering methods are also presented: Dynamic Meta Modeling (with Time) and Visual Contracts.

The second part is devoted in more detail to the meta-method for modeling and tailoring of software engineering methods, called MetaME. It combines ideas from meta-modeling, method engineering and language engineering. The meta-method comprises a product dimension and a process dimension. Artifact types are derived from software engineering concepts to form the product dimension. In the process dimension, software development tasks are described as operations that act upon the artifacts. These tasks are performed as activities in the method's workflow.

The third part contains a selection of my most important research contributions to the field of software engineering methods and their systematic development.

Keywords: Method Engineering, Software Engineering Method, Meta-Model, Model-based Software Development, Model-driven Development, Object-oriented Modeling, Multimedia Applications, Business Information Systems, Interactive Systems, Advanced User Interfaces

Zusammenfassung

Die Entwicklung moderner Softwaresysteme erfordert den Einsatz anspruchsvoller Softwareentwicklungsprozesse und -methoden. Aufwand, der einmal in die Methodenentwicklung investiert wird, kann systematisch in Projekten ausgenutzt werden. Wir verwenden die Bezeichnung „Method Engineering“ für die systematische Entwicklung von Softwareentwicklungsmethoden.

Im ersten Teil dieser Dissertation stelle ich Softwareentwicklungsmethoden und Ansätze des Method Engineering vor, die ich entwickelt habe, und charakterisiere sie. Sie beschäftigen sich insbesondere mit der modellbasierten und modellgetriebenen Entwicklung von betrieblichen Informationssystemen, Multimediasystemen und anspruchsvollen interaktiven System sowie einigen verwandten Domänen. Zwei formale Methoden, die in den Softwareentwicklungsmethoden eingesetzt werden, werden außerdem erläutert: dynamische Metamodellierung (mit Zeit) und visuelle Kontrakte.

Der zweite Teil ist der detaillierten Beschreibung der von mir entwickelten Meta-Methode MetaME für die Modellierung und Anpassung von Softwareentwicklungsmethoden gewidmet. Sie kombiniert Ideen der Metamodellierung, des Method Engineering und der Entwicklung von (Modellierungs-) Sprachen. Die Meta-Methode umfasst eine Produkt- und eine Prozessdimension. Artefakttypen werden von Konzepten des Software Engineering abgeleitet, und sie bilden gemeinsam die Produktdimension. In der Prozessdimension werden Softwareentwicklungsaufgaben als Operationen beschrieben, die auf den Artefakten ausgeführt werden. Diese Aufgaben werden als Aktivitäten im Ablauf der Methode ausgeführt.

Der dritte Teil enthält eine Auswahl meiner wichtigsten Forschungsarbeiten, die ich zu dem Themenbereich Softwareentwicklungsmethoden und deren systematische Entwicklung publiziert habe.

Schlüsselwörter: Methodenentwicklung, Softwareentwicklungsmethoden, Meta-Modell, modellbasierte Softwareentwicklung, modellgetriebene Softwareentwicklung, objektorientierte Modellierung, Multimediaanwendungen, betriebliche Informationssysteme, interaktive Systeme, moderne Benutzungsschnittstellen

Contents

Part I:

Model-based Software Engineering Methods

1 Introduction

The development of today's software systems demands for sophisticated software engineering processes and methods. Not only the (globally) distributed development of large and complex software systems by heterogeneous teams requires precise and documented methods, but also lightweight and agile methods need to have a precise foundation. Effort that is invested once in the methods can be systematically reused in software development projects and other software development endeavors. This applies to different types of systems (also called system domains in this work) and across application domains. To build the required software engineering methods in a systematic way, engineering principles must be applied not only for the development of software systems, but also for the definition of the methods themselves. The systematic development of software engineering methods is called *method engineering* in this work.

Software engineering methods are required for all software engineering endeavors – such as software development projects or systematic software product development – regardless of their strictness or agility. Yet, the development of software engineering methods has to address a number of challenges:

(a) Software engineering methods must be precisely described.
(b) Software engineering methods must be usable and oriented towards the people that need to work with the methods or the produced results.
(c) Software engineering methods must be adequate for the tasks at hand and the domains in which they are deployed.
(d) Software engineering methods must be reusable and tailorable for distinct and dynamically changing situations, such as domains, organizations, project context or even project situations.

In this work, we concentrate on model-based (and model-driven) software development for software systems of different kind. Depending on the increasing level of importance of models in development methods, we distinguish different notions:

- *model-based*: the development method deploys models to describe certain aspects of the system, and models play an important role in the development method; they are the central artifacts, other artifacts are secondary and accompany the models;
- *model-driven*: models are the central artifacts of the development method, and model transformations accompany the models, which define how (partial) models relate with each other and how models are derived from other models.

In the following, we distinguish between model-based and model-driven from a methodological point of view. As a consequence, models are in either case the central type of artifacts that are produced in the software development process. However, in model-based and model-driven development, they are used for different development purposes and thus play different roles in the software development methods and processes. In addition, due to their use in a range of system domains, the models also differ with respect to the aspects of the software system that they capture. We will stress these aspects where the distinction is important.

From a methodological perspective, the systematic development of software engineering methods is an engineering task and has to be directed by and follow a well-defined method engineering method. Being a method itself, such a meta-method has to comprise a process and a product part. The process part altogether defines who has to do what and when. The product part defines the required work products (types of artifacts) that are expected to be used or produced by the tasks and activities of the process part. Both parts hold a set of relevant aspects that are required to characterize a software engineering method.

In the contributed software engineering methods and attached works, I present a set of method constituents, i.e., products of method engineering. Their main characteristics and, to a limited degree, the process how they have been developed will be described in the following sections.

The first part of this thesis presents and characterizes a collection of partial software engineering methods and method engineering approaches that I have worked on. We particularly look at the model-based and model-driven development of business information systems, multimedia applications and other advanced interactive systems (i.e., specifically their user interfaces), and some related domains. I also contributed to the development of two methods that are used for specifying the semantics on the level of models and meta-models and that are used in the collected software engineering methods: Dynamic Meta-Modeling (with Time) and Visual Contracts.

In the second part of this thesis, we look in more detail at the meta-method for modeling and tailoring of software engineering methods, called *MetaME*, that I have developed as a method for the systematic development of software engineering methods. It combines ideas from meta-modeling, method engineering and language engineering. The meta-method comprises a product dimension and a process dimension. Artifact types are derived from software engineering concepts to form the product dimension. This is the foundation for the development of languages for describing and modeling the software engineering methods. In the process dimension, software development tasks are described as operations that act upon the artifacts. For their formal description, we reuse the formal methods from the software engineering domain in the method engineering domain. The tasks of the method are performed as activities in the method's workflow.

The third part of this thesis contains a selection of my most important research contributions to the field of software engineering methods and their systematic development.

The remainder of this Part I is organized as follows: The general framework for integrating software engineering and method engineering and the classification of contributed software engineering methods according to a classification scheme is the subject of Section 2. In the following, one section is dedicated to each class of research contributions: formal methods for software engineering methods (Section 3), fundamental methods that deal with general approaches related to model-driven development methods and their application (Section 4), and methods for two system domains: multimedia and interactive systems (Section 5) and business information systems (Section 6). Previous work on method engineering is presented in Section 7, followed by some concluding remarks.

2 Classifying Research Contributions to Model-based Software Engineering Methods

In this section, I first introduce the common framework for method engineering and software engineering and then the classification scheme for the characterization of software engineering methods. It is based on the identification of aspects that are important for a software engineering method. A collection of partial software engineering methods that I have worked on is presented and classified according to this classification scheme. Experience from this work has led to research on method engineering which will be described in Section 7. Publications that are contributed to Part III of this thesis are indicated by citation markers in bold letters.

2.1 Engineering Framework for Methods and Software

Before we analyze the relevant aspects for the classification of software engineering methods, we have to generally distinguish between the two fundamental engineering domains that are involved in the systematic development of software engineering methods: method engineering and software engineering. We use the term *method engineering* in this work to denote the systematic development of software engineering methods.

Software engineering is the engineering domain that is concerned with the systematic development of software systems and the artifacts that belong to a software system along its lifecycle. *Software engineering method* is used in this work to denote the full set of elements needed to describe a software development endeavor (e.g. a software development project) in all relevant aspects. This does not only cover the software development process and its contained activities, but also the artifacts (containing their semantic content and their syntactic representation by appropriate languages) that are to be produced, the tasks that need to be performed for achieving the development goals, the roles in an organization that participate in the development, the tools, techniques and utilities that are employed, as well as relationships between these concepts.

A corresponding characterization can be applied to the *method engineering meta-method*: it denotes the full set of elements needed for developing and describing a software engineering method, in all relevant aspects. This covers the method engineering process and its contained activities as well as the method elements that are to be produced, the tasks that need to be performed to achieve the development goals, the roles in an organization that participate in the method development, the tools, techniques and utilities that are employed, as well as relationships between these concepts.

In our framework, each method consists fundamentally of a process model and a product model (see Figure 1). The *process model* specifies how to proceed in the execution of the method. We call the execution of a concrete process instance that conforms to the process model *enactment* of the process model. The *product model* defines the results (i.e., work products) that are expected from executing the method. We call the production of a concrete result that conforms to the product model *instantiation* of the product model. The objective of software engineering is to support the development of software artifacts (the software system being one distinguished type

of artifact) in a systematic way by enacting the process of a software engineering method. Analogously, method engineering is concerned with the systematic development of software engineering methods by enacting the process of the method engineering meta-method. The meta-method is itself a method, but is called meta-method since it is employed to produce methods as the result of its enactment. A concrete software engineering method thus is an instance of the method engineering's product model. In the same way, software artifacts are instances of the product model of software engineering methods.

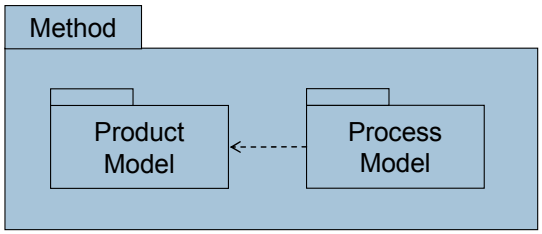


Figure 1: Each method comprises an product model and a process model, where the process model depends on the product model (dashed arrow)

Figure 2 summarizes these correspondences and relationships. Method engineering thus corresponds to the enactment of the process model and the instantiation of the product model of the method engineering meta-method. In turn, software engineering corresponds to enacting the process model and instantiating the product model of the software engineering method.

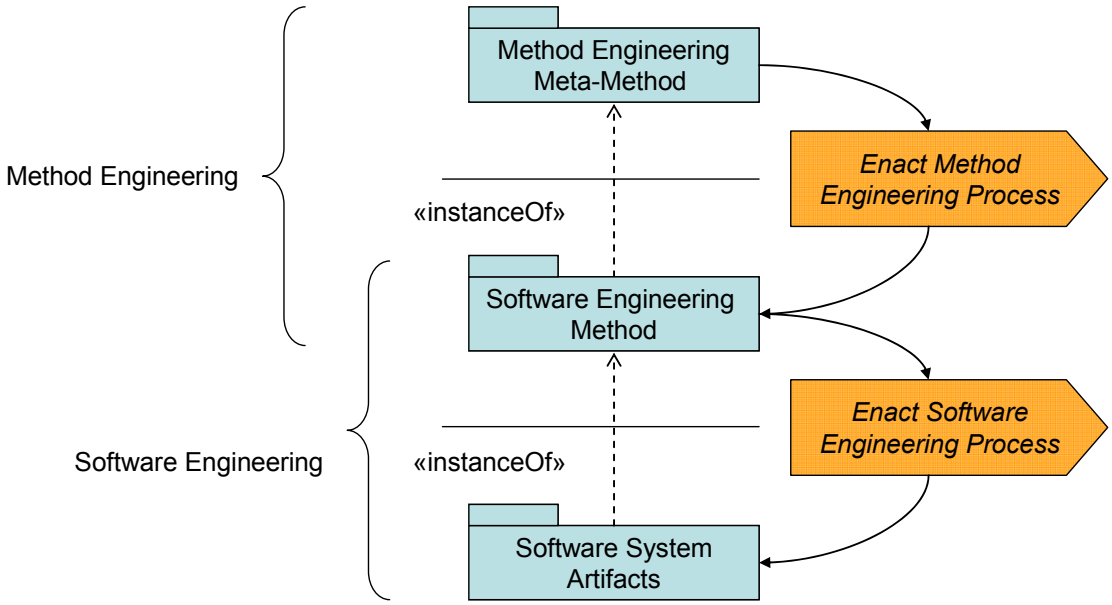


Figure 2: Method engineering and software engineering are two engineering domains that are related by the concept of methods

In our overall method and software engineering framework, we assign the different methods to different layers of a hierarchy. More generally, each method is the product of enacting and instantiating its meta-method. In such a meta-method hierarchy, we distinguish different method engineering domains. In analogy with the MOF meta-model hierarchy we find the domain 'engineering with objects' on the topmost layer.

There, we locate our foundational meta-meta-method. Enacting this meta-meta method (i.e., its process model) and instantiating its product model, we obtain our meta-method for method engineering. It is located on the next layer. Methods of this layer deal with the systematic development of methods for a particular domain, which is software engineering in our case. This domain is located on the third layer and contains the software engineering methods that are employed to develop software systems. They are developed by enacting and instantiating the meta-method for method engineering. Finally, software engineering then is concerned with enacting and instantiating the software engineering methods in order to run a software engineering endeavor and produce the software artifacts that instantiate the artifact model, i.e., the product model of the software engineering method.

A refined view of this (meta-) modeling architecture is given in the context of the MetaME meta-method for method engineering in Section 11.1.

Next we look more precisely into the domain of method engineering. If we work in the domain of method engineering, we encounter a number of more or less general disciplines (see Figure 3). Method engineering can itself be considered as an (engineering) discipline. Regardless of considering method engineering as the domain or the discipline, it comprises a set of other (engineering) (sub-) disciplines. In this work, we distinguish the disciplines requirements engineering, domain engineering, language engineering, process engineering, and software engineering.

Requirements engineering is the origin of any systematic engineering process, also for eliciting the method requirements. All other engineering disciplines and their respective products depend on good requirements engineering. Language engineering deals with the development of appropriate languages for representing concepts and artifacts of the domain, such as modeling and programming languages for software development. This concerns both syntax and semantics of languages. Concepts are the ontological foundation of the software engineering methods and are the first main product of the domain engineering discipline. The definition of artifact types, which define the kinds of products of the software engineering method, is also assigned to domain engineering. Here we match concepts and languages, such that their semantics are compliant and the syntax is appropriate for representing the concepts within the software engineering method.

On top of the artifact model, we develop methods and processes for software engineering for producing and working with the corresponding software artifacts. Development of processes (including activities, milestones, workflows, phases, etc.) is the business of the process engineering discipline. The method engineering discipline more generally deals also with tasks, roles and organizations, artifact states, techniques and practices, guidance, utilities, and so on. (Refer to Section 11.2 for more details.)

Finally, we work in the software engineering discipline ourselves when we develop a comprehensive software engineering method. This self-reference is due to the demand for the systematic development of CASE tools that support the designed software engineering method.

This model of disciplines and their products makes up the backbone of our meta-method that will be described in Part II. The model of artifact types (also called artifact model),

which combines concepts and languages for their representation, is the product model of the meta-method, while the methods and processes form the process model of the meta-method. Figure 3 depicts this model within the domain of method engineering. Dashed arrows show dependencies.

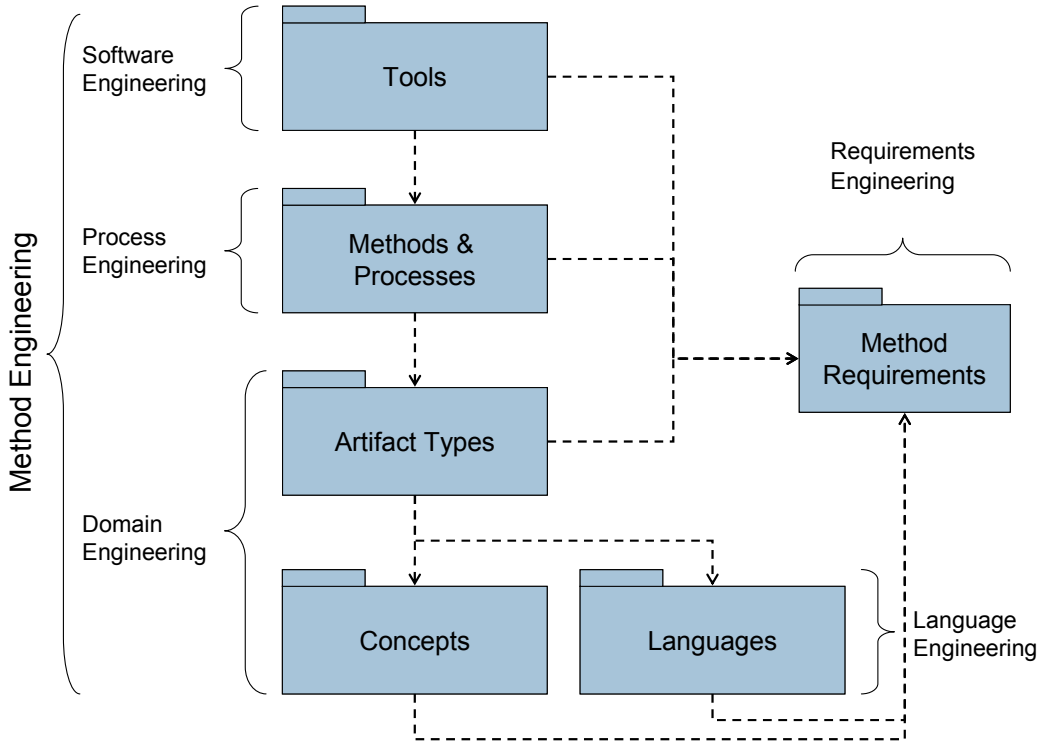


Figure 3: Method engineering and its contributing (sub-) disciplines produce the elements of a software engineering method

To illustrate the concepts and their relationships, we look at an example: We consider the system dynamics to be a relevant aspect of a software system and want to include it in our software engineering method. To address this aspect in our software engineering method, we include the concept of system dynamics in the conceptual model that is developed as part of the domain engineering discipline. We then look for candidate languages that appear appropriate for the representation of the system dynamics concept. We either use existing, ideally standard languages for specifying system dynamics such as Petri nets, automata or statecharts, or define a new language ourselves. Assume that we decide on using UML statechart diagrams for representing system dynamics. Then we define the artifact type "system dynamics model" as the representation of the concept "system dynamics" by the language UML statechart diagrams. In the software engineering method, we then also specify the pragmatics of how (method: guidelines, techniques and practices) and when (process: workflow) to build, modify and use the system dynamics model in the software engineering endeavor that applies our software engineering method.

From the presented model of method engineering disciplines and their respective products, we can derive the important aspects of software engineering methods that we consider in our classification of the methods in the next sections.

2.2 Classification of Works

The software engineering methods in the upcoming sections, with the exception of the formal methods for software engineering that are presented in Section 3, are classified according to the main aspects of software engineering methods. Not surprisingly, these aspects correspond to the main products of the method engineering process. With this classification, we can show the scope and coverage as well as the relationships of the different methods that have been developed in previous research.

In our classification we consider the aspects system domain, method domain, concepts/artifact types, process, languages, methods/techniques, and tools as the characteristic dimensions. We briefly explain them before we show the results of the characterization in tabular form.

System domain. An important aspect of software engineering methods is the domain for which software has to be developed. We distinguish between application domains and system domains (also referred to as system classes). Application domains are business domains (e.g. branches of trade like finance, sales, etc.) or engineering domains (e.g. automotive production, chemical production, etc.), but also science, education, and entertainment. In contrast, system domains in our context relate to different types of software systems such as business information systems, embedded systems, mobile systems, or multimedia and interactive systems. While the application domain predominantly affects data structures and functionality, the system domain has also important impact on the software and system architecture as well as on the software technology to be used. Most importantly, software engineering methods often have to be tailored to the specific requirements and properties of a system domain, while the application domain only has a limited effect on the design of the software engineering method. We therefore restrict our classification in general to the system domain and only note the application domain where appropriate.

Method domain. The term method domain is used to refer to the development paradigm that is used in order to build the system under consideration. Most importantly, we distinguish between model-based and model-driven development methods with this regard. But also other development styles such as design-by-contract, architecture-driven development, prototyping, or end-user development belong to this aspect.

Concepts and artifact types. This dimension discriminates software engineering methods with respect to the artifact types that are produced or used. Prominent among them, since we focus on model-based and model-driven development methods, are different kinds of models. Nevertheless, all kinds of software engineering artifacts from specification documents to code to test reports belong to this aspect. Deliverables are also regarded as a specific kind of artifact. Since the distinction between concepts and artifacts plays no significant role in our classification, we use the term artifact types in the tables; more precisely, this dimension refers to the conceptual part of the artifact types. The notational part is covered by the aspect language (see below).

Process. The process dimension refers to the software process model that belongs to the software engineering method. It collects all information regarding workflows and tasks,

but also role models or the assignment of methods to particular stages (or phases) of the software engineering process.

Language. The language aspect refers to the notation in which software artifacts are represented in the context of the software engineering method. In our research, we either use or extend the Unified Modeling Language (UML) for object-oriented modeling or develop dedicated domain-specific languages where necessary and appropriate according to the concepts of language engineering. Programming and scripting languages, and other types of specification languages also belong to this dimension.

Methods and techniques. In this classification category, we consider all methodical elements of a software engineering method that determine how the software engineering is done. Techniques and methodical guidelines are prominent members of this category, but also reference architectures, frameworks, or specific technology.

Tools. The tool aspect refers to software tools that are available to support the method or are used when enacting its process. Editors, simulation tools, code generators, consistency checkers, execution environments, analysis tools, libraries, and so on are typical instances. We also allocate other utilities such as checklists or templates to this aspect.

With these dimensions, we are now equipped to classify and discriminate the relevant research on software engineering methods. Prior to that we note a few remarks on the common grounds of the software engineering methods as regards their founding on the paradigm of object-orientation and the formal methods that we have developed to support our software engineering work. The latter is also the subject of Section 3.

All methods have in common that they contribute to the model-based development of software systems of different kinds, i.e., they belong to a given system domain. The modeling paradigm that underlies our approach is object-oriented modeling. Thus, our methods are concerned with model-based software development on the foundations of the paradigm of object-orientation. We either use or extend the Unified Modeling Language for our object-oriented modeling or develop dedicated domain-specific languages where necessary and appropriate according to the concepts of language engineering.

As foundational work towards developing object-oriented modeling languages for particular application and system domains, like multimedia applications or business information systems, we have analyzed the principle capabilities and limitations of the Unified Modeling Language (see [EHS00], [HKS01]). This analysis builds an important foundation for the decision which extensions are needed to tailor the UML for the particular set of requirements of a given domain.

If we build domain-specific extensions of the UML, we deploy the built-in extension mechanisms of meta-modeling (according to OMG's Meta Object Facility (MOF)) or profiling. In addition to using these formal concepts in our method and language engineering, we also use two formal methods that we developed ourselves as a theoretical foundation: Dynamic Meta Modeling (DMM) and Visual Contracts (VC). DMM is a graphical technique for the specification of precise semantics of modeling

languages such as the UML. It can be applied to both the meta-model level, e.g. for the specification of action semantics in behavioral models, and to the modeling level, e.g. for specifying the semantics of operations. Visual contracts use a graphical object-oriented model for specifying preconditions and post-conditions of behavioral elements in a model. Thus, we combine the application of software engineering principles and object technology for the development of specific kinds of systems such as multimedia applications, Web applications, service-based systems or business information systems. Examples are OMMMA, an extension of the UML for model-based development of multimedia software systems, and ProGUM-Web, a UML-based approach for the model-driven development of Web applications; but also the application of visual contracts for the semantic specification of operations in Java or Web services.

The following Table 1 shows the results of the analysis of previous research with respect to the aforementioned dimensions. The tables are grouped according to the sections on fundamental methods (Section 4) that generally apply and methods that have been developed in the context of a particular system domain. Among them, we distinguish between the general system classes multimedia and interactive systems (Section 5) and business information systems (Section 6). Where appropriate, the members of these domains are further specialized with respect to the system type they have been developed for. The approaches for business information systems are clustered in two groups: in the first group are the specification method SPECME for business information systems (Section 6.1) together with its accompanying method integration approaches for application landscape and application development (Section 6.2) and for quality assurance by the use of the dedicated specification quality gate (Section 6.3). The other approaches for (business) information systems build the second group.

Table 1: Characterization of conducted research on software engineering methods according to the core main of software engineering methods

Software Engineering Method	System Domain	Method Domain	Artifacts	Process	Language	Methods / Techniques	Tools
Java Code Generation **[EHSW99a]**, [EHSW99b]	general: Java programs	model-based development; conceptually model-driven: code-generation	class model, behavior model, Java code	design modeling; generation of executable code	UML collaboration diagram, UML class diagram; refined UML meta-model; Java	methodical guidelines: modeling of object interaction; two-level grammar specifies transformations; conceptual transformation algorithm	n/a
Visual Contracts [LSE05], [ELS05a], [ELS05b],[LES06], [ELSH06], [LRE+06], [EGS08]	Java programs (operations) [LSE05], [LES06]; Web Services [ELS05a], [ELS05b]; enterprise services [LRE+06]; software testing [EGS08]	design-by-contract (on model level); model-driven monitoring: validation of programms with assertion code [LSE05], [LSE06]; Web service specification and discovery [ELS05a], [ELS05b], and validation of Web services [ELS05b]; specification and management of enterprise services	class models, visual contracts; Java code (classes, behavioral code), JML assertions; binary code	contract-based development process; workflows for: design modeling, code generation: class diagram (Java class skeleton); operations annoted with assertion code; manual implementation: functional code (method bodies), additional operations & classes	UML 2 class diagram, UML 2 object diagram (pair) [LSE05], [ELS05a], [ELS05b], UML 2 composite structure diagram (pair) [LES06], [ELSH06]; UML 2 meta-model extension [ELSH06]; formalized by graph transformation theory: specifcation & matching: graph transition, generation of JML assertions: graph transformations; Java, JML (behavioral interface specification for Java)	contract-based development method: visually specifying contracts; model-driven monitoring: code generation for classes, assertion code generation, manual implementation of additional code; transformation algorithm; semantic matching of Web services: code generation for ontology classes, generation of semantic service specification	Visual Contract Workbench (Eclipse plug-in); JML Compiler, JMLUnit AGG, Jena [ELS05a]

Software Engineering Method	System Domain	Method Domain	Artifact Types	Process	Language	Methods / Techniques	Tools
OMMMA [SE99a], [SE99b], [SE99c], [SE99d], [ES02], **[SE01]**, **[ES02]**	multimedia applications, multimedia presentations; multimedia information systems [SE99c]; automotive infotainment systems [SE99b], EGS01]	model-based development; object-oriented modeling	integrated model conforming to MVC_{MM} architecture; OMMMA model types (for individual aspects): structure model, timed behavior model, dynamic behavior model, presentation model	(suited for analysis and design stage)	OMMMA-L (UML extension); one diagram type for each model view: class diagram with framework (e.g. hierarchy of media classes), timed multimedia sequence diagram, presentation diagram, statecharts; meta-model derived from UML meta-model [SE99d]	methodical guidelines for OMMMA-L modeling; tight integration of time-dynamic behavior **[SE01]**	OMMMA-Edit modeling environment: extension of Rational Rose 98: syntax-direct editors for diagrams, cross-diagram consistency checks
Model-based Development with Multimedia Authoring System (MMAS) [DEM+98], [DEM+99]	multimedia applications, multimedia presentations	model-based development with MMAS; conceptually model-driven	type and instance models: platform-independent analysis model; design model of authoring system ("programming model"); application model framework, dimensions: application logic, presentation, control, media	model-driven process model for analysis and design stage	UML class diagram, UML object diagram	modeling method; modeling framework (stages, type/instance); class framework for multimedia; conceptual model transformations	n/a
Integrated Method for Interactive Multimedia Systems [ESN03]	multimedia applications, multimedia presentations	OMMMA-based modeling; user-centered design	OMMMA models; interaction design artifacts: conceptual model, visual & design structures, storyboard, prototype, etc.	integrated process: OMMMA process (softtware engineering and media development workflows) and user-centered design process (user-driven workflows)	OMMMA-L; other, mostly informal graphical notations	n/a	OMMMA-Edit; graphics tools; prototyping tools
GuiBuilder [SDGH06], **[SE07]**	graphical and multimedia user interfaces	visual model-driven development; simulation, scripting, capture-replay; end-user development	presentation model dynamic behavior model	workflows: modeling, simulation	(limited) hierarchical UML statecharts; object-oriented presentation diagram	modeling guidelines; static and dynamic model validation; interpretive prototype simulation (with model monitoring), capture-replay, scripting	GuiBuilder: visual modeling & execution tool based on Eclipse: model editor, model validator, generator function, simulator
ProGUM-Web [LSS03]	dynamic Web applications	model-based, incremental and iterative development; prototyping; partly model-driven: code template generation, prototype generation	functional requirements (use cases); use case specification: client-server-interaction, user interaction, business logic; object flows for information exchange; data structure & site structure class models	incremental and iterative process: modeling workflow, coding workflow, prototyping workflow; developer roles: software developer, graphic designer	UML extension: UML use-case diagram, UML class diagram, UML activity diagram; extended UML meta-model; target languages: HTML, PHP (etc.) scripts	separation of design and business elements; integrated: cooperative development by graphics designer and software developer; prototyping	UML CASE tool Enterprise Architect; XMI, Java

Software Engineering Method	System Domain	Method Domain	Artifacts	Process	Language	Methods / Techniques	Tools
Specification Method (SPECME) [SSE09a], [SSE09b], [SSEB10]	business information systems	model-based system specification	system specification; artifact types (use case, entity type, functional overview etc.) defined by artifact model; main artifacts correspond to specification modules	definition of the overall processes: phases, roles, milestones, sub-disciplines; definition of tasks for creating artifacts: task-oriented specification method: steps, input artifacts, output artifacts	UML, natural language, tables, other (graphical) notations	detailed method description;strict separation of the method aspects: content, form, process and tools; methodical guidelines how to construct artifacts; structured by specification modules for main artifacts: description of artifact type, its representation, its production and use; templates, examples, hints	UML CASE tool (Enterprise Architect), Office tools, other required editors; Document Generator, Specification Validator; concept of use for standard tool set-up; specification templates
Integrated Specification Framework (SPECME plus Quality Gates (QG) [SSE09a], [SSE09b], [SSEB10]	business information systems	integrated specification and quality assurance; SPECME: model-based system specification, QG: specification quality assurance	system specification; according to the artifact model; QG check methods; reports	process synchronization SPECME and QG processes; SPECME: see above, QG: QG assessment process: phases, roles, tasks, continuous QG enhancement process	as in the system specification; QG: spreadsheets	alignment of SPECME & QG method, method integration on the foundation of the common artifact meta-model; QG: QG method guide; check methods for process and products (artifacts); checklists, user scenarios, change scenarios; concept of use for check methods	SPECME: as above; QG: checklists, templates
Integration of Application Landscaping and Development [BEH+09]	business information systems; service-oriented application landscapes; enterprise services	holistic (model-based) development of service-oriented application landscapes: integrating (model-based) application landscape development and (model based) application/service development	according to combined artifact meta-models; artifact types of both contributing methods, such as business domain, business process, business task, service, enterprise IT architecture, IT system, application landscape, component, interface, sub-system, action, operation, use case, etc.	alignment of application landscaping and software development processes based on artifact dependencies; transition between disciplines: business architecture modeling, landscape modeling, managed evolution, integration architecture management (QE); business modeling, requirements engineering, analysis & design, implementation, deployment, test, software controlling, etc. (Q)	UML models in QE and Q; Quasar Ontology: integrated meta-model of software engineering concepts, refinement links between domain concepts	integration of full-size industrial methods: Quasar (Q), Quasar Enterprise (QE); guidelines, patterns, reference architectures, scenarios, etc. (analysis, comparison and matching of concepts from both domains); complete, modular software engineering method: concept ontology, languages, artifact types, methods/processes, tools; ontology integration; language, tool, and method/process alignment	Quasar tool support, e.g. as in SPECME; QG: e.g. Integrated Architecture Framework (IAF)

Software Engineering Method	System Domain	Method Domain	Artifacts	Process	Language	Methods / Techniques	Tools
Integration of Requirements Engineering and Testing [GFJ+09], [GSW+10]	business information systems: eID systems	requirements engineering, model-based testing	textual requirements, requirements clusters; test plans, formal and informal acceptance criteria	requirements engineering, acceptance testing, part of sophisticated development process for eID systems; workflow with 3 activities: annotation, clustering, test-plan specification	natural language (requirements), tables (test plan); test-plan meta-model	multi-viewpoint requirements engineering (based on RM-ODP), acceptance testing of eID systems; linguistic analysis, requirements clustering, pattern/template-based requirements collection; detecting overlaps and temporal order; (semi-automatic) generation: test plan (pattern-based, with heuristics), acceptance criteria (formal for test steps, informal for asserts); test-plan specification	tool for capturing and managing requirements; templates for different RM-ODP viewpoints; TORC environment: parser (linguistic analysis), clustering algorithm, quality plan creation, statistics, etc.
Integrated Quality (QA) Methods in SCRUM [EGSP09]	(financial) business information systems	agile development; system test: performance testing, usability testing, user acceptance testing	non-functional requirements; product backlog (PBL), task entries in PBL; performance and load tests	extension of SCRUM process: QA days (1-3): additional QA activity with customer and user: extensive testing of functionality, non-functional properties (usability, performance); user-testing day (1): usability/performance/load tests with significantly larger group of users; defined sub-processes and tasks; dynamic process improvement	n/a	methodical extension to SCRUM for QA: early customer/user feedback; performance testing, (usability testing); usability and performance tasks in the PBL; methodical hints for managing QA and user-testing days; manual load/performance tests during user-testing day complement automated load and performance tests during QA days in customer environment	product backlog, management system
Open-Source Stacks: Architecture-driven Development Method [CS08]	(financial) business information systems	architecture-driven development; stack-based development (reuse in the large)	open-source software stack (incl. documentation, specification); software system	development process with open source stacks; activities identified for: selection, coupling, updating, replacing, testing; role model: developer, distributor, consultant, user	(specification language); (modeling and programming languages)	methodical guidelines how to develop software with open source stacks, e.g. selection, coupling, obtaining up-to-date information, update, replacement, quality assurance	open-source stacks = preconfigured assemblies of open-source components and frameworks

3 Formal Methods for Software Engineering Methods

Key to model-based and model-driven software engineering methods are precise and usable modeling languages. The Unified Modeling Language has been designed to be *the* object-oriented modeling language of today. However, it is also commonly agreed that it is not the *lingua franca* of software and systems modeling. But it comes with built-in extension mechanisms to define domain-specific variants of the modeling language. Regardless, any modeling language needs to have precise semantics in order to support rigorous modeling methods. The results of an analysis of UML's feasibility as a universal modeling language are presented in Section 3.1. Semantic dimensions of sequence diagrams and their impact on rigorous modeling are shown in Section 3.2. The remaining sections deal with approaches that colleagues and I have undertaken to formalize the semantics of UML collaboration diagrams (Section 3.3) and to use collaboration diagrams with a precise semantic interpretation by graph transformation theory as a means for specifying the semantics of modeling languages. We call this approach Dynamic Meta Modeling (with Time), see Sections 3.4 and 3.5.

The different methods, their main properties and dependencies, as well as the relevant publications, are summarized in Figure 4.

3.1 Rigorous Modeling with the Unified Modeling Language

Object-oriented modeling is the conceptual anchor point for our model-based and model-driven development methods. The Unified Modeling Language (UML) is the *de facto* industrial standard of object-oriented modeling languages. It consists of several sub-languages which are suited to model structural and behavioral aspects of software systems. The UML was developed as a general-purpose language together with intrinsic features to extend the UML towards domain-specific profiles that best fit the context of use and the problem to solve. In [EHS00], we illustrate the language features of the UML family of languages and its adaptation mechanisms. We show that the UML or an appropriate, to be defined core UML, respectively, may serve as a universal base of object-oriented modeling languages. But this core has to be adapted according to domain-specific requirements to yield an expressive and intuitive modeling language for a certain problem domain. With its built-in extension mechanisms – stereotypes, tagged values, and constraints – the UML readily supports the definition of such domain-specific profiles.

The analysis undertaken in [EHS00] reveals that the UML, on the level of abstract syntax, really is an integration of modeling languages, due to the definition of one common meta-model for all sub-languages. We call this a family of modeling languages. This finding also applies to the concrete syntax level, since an agreement on concrete notations has taken place, too.

Nevertheless, the UML is just a modeling *language* and still needs to be incorporated into software engineering methods and processes. This includes the definition of methodical guidelines for using the language features (i.e., the language's pragmatics), a software process model as well as techniques to transform UML models into other (UML) models and eventually into a corresponding implementation in a programming language.

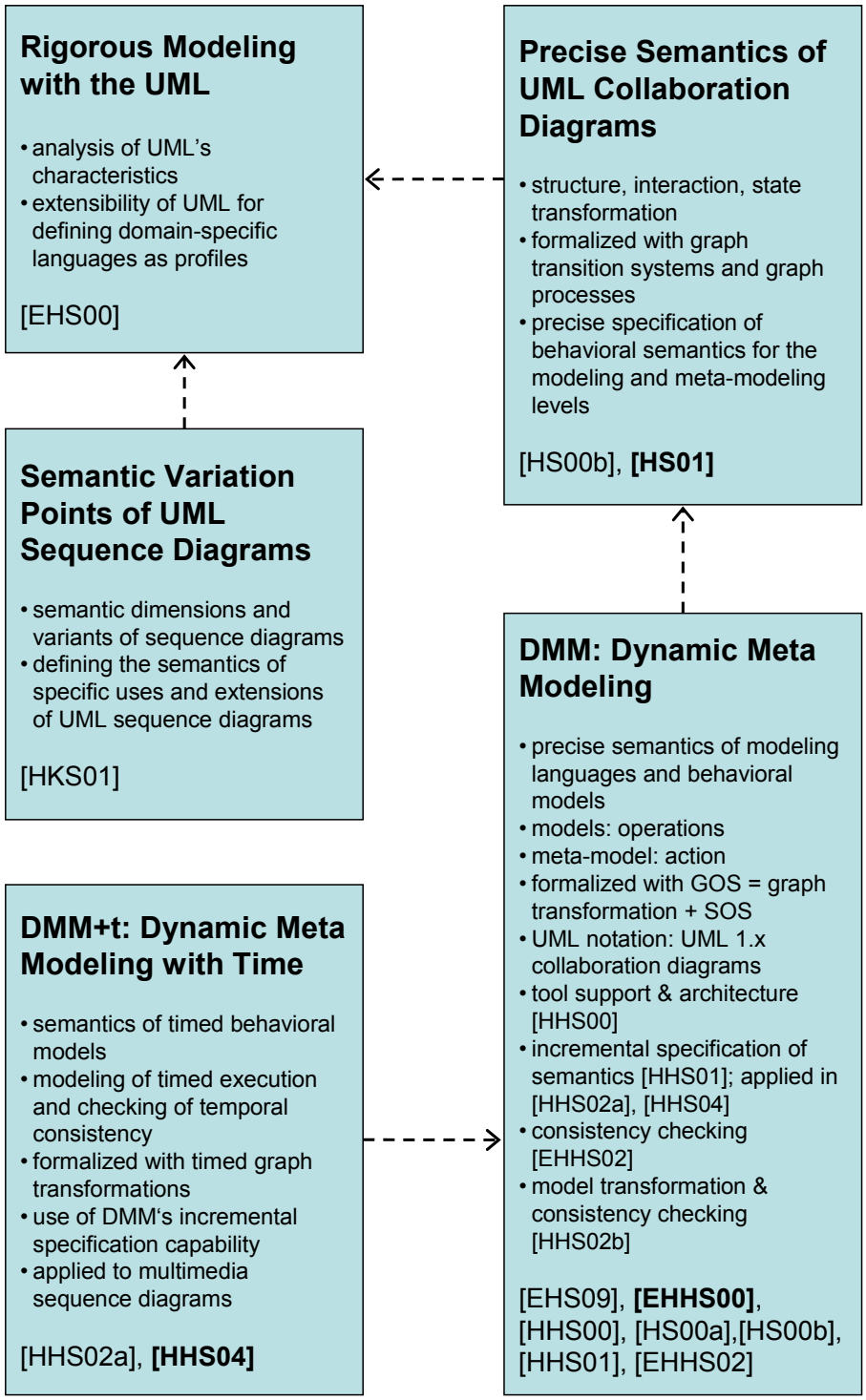


Figure 4: Overview of the contributions to precise semantics, formal methods and rigorous modeling

3.2 Semantic Dimensions of Sequence Diagrams

For a consequent and unambiguous use of a modeling language, not only its syntax, but also its semantics need to be precisely defined. In [HKS01], we survey, structure, and classify syntactic and semantic alternatives that appear in sequence diagrams, since different interpretations for sequence diagrams exist without explicit means to distinguish between them. Objective of this survey was to find semantic modeling

concepts that exist in sequence diagrams. We identify scope of interpretation, level of abstraction, composition and refinement, ordering, time, and represented function as the essential semantic dimensions of sequence diagrams (see Figure 5). Each dimension can be considered a semantic variation point of sequence diagrams. For each semantic variation point, we identify the semantic choices (i.e., the semantic variants) and we justify each choice by supplying examples for its usage in sequence diagram modeling. It is shown that UML sequence diagrams (1) do only support a subspace of the semantic concepts and (2) have ambiguous semantics with respect to several of the semantic variations identified (marked with borders and flashes in Figure 5, respectively).The spanned semantic space is suited as a basis for discussing and proposing extensions of UML sequence diagrams to precisely determine the semantic interpretation of modeled sequence diagrams (as in the case of multimedia sequence diagrams, see Sections 5.2 and 3.5).

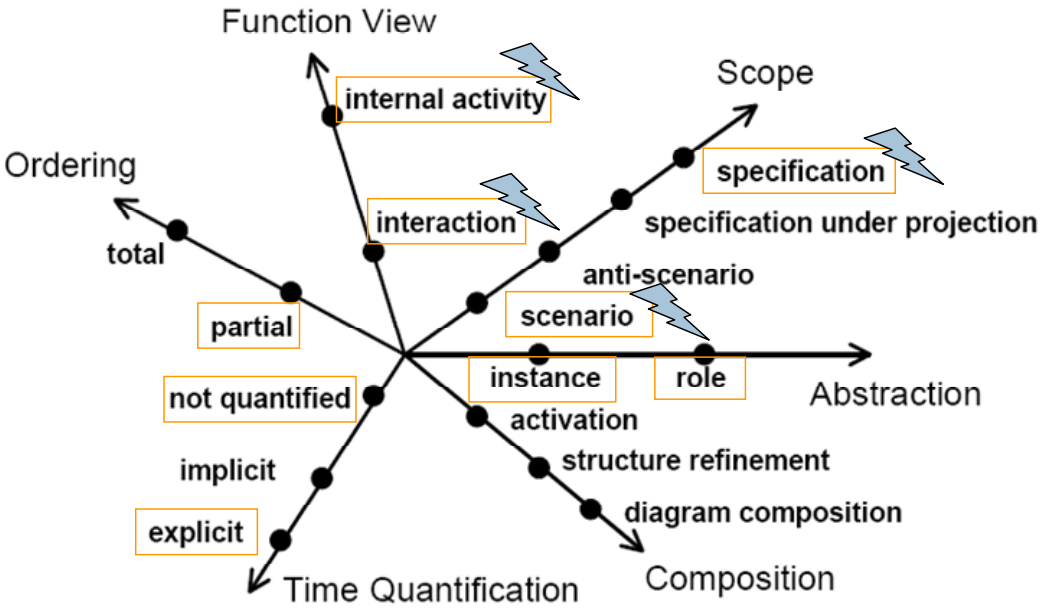


Figure 5: Semantic dimensions of sequence diagrams and their coverage by UML 1.4 (based on [HKS01])

We propose a method and two-step process for indicating the semantics of sequence diagrams with respect to semantic variations in order to define precise semantics. First, one has to define the semantic framework by introducing stereotypes for each of these semantic variations. Second, when using a sequence diagram, the modeler has to supply one stereotype for each dimension, thereby fixing the semantics. Each combination of stereotypes defines new semantics for sequence diagrams. For each combination, specific well-formedness rules may be supplied. As a consequence, defining stereotypes for each variation is an important task and should be done with great care. Essentially, this task corresponds to defining a new specialized sub-language. OCL constraints can be used to specify invalid combinations and exclude them from the set of all valid sequence diagrams.

The use of stereotypes for fixing sequence diagram semantics can easily be integrated in CASE tools. Thus it can be ensured that the semantics of each sequence diagram can be fixed by the modeler.

Within a rigorous development process, it is of importance that modeling activities can be assigned with sequence diagrams with specific semantics. For example, it may be possible to use scenarios in early stages (e.g. requirements engineering) and then proceed to specifications in later stages (e.g. analysis and design). Using our method to fix the semantics of sequence diagrams, a process model can now precisely define the form of sequence diagram to be used, e.g. by restricting or prescribing stereotypes for a development activity. Due to the fixed semantics, precise consistency relations and checks can be formulated that have to hold between different (partial) models within a development process. Thus it can be ensured that sequence diagrams are used so that no contradictions occur.

3.3 Precise Semantics of UML Collaboration Diagrams

We first proposed precise semantics for UML 1.x collaboration diagrams (they are called interaction diagrams in UML 2.x; formally, with respect to the UML 2.x meta-model, the interaction represents the behavioral part, while their structural part is still called collaboration) based on graph transformation rules and graph processes in [HS00b]. This semantics definition makes collaboration diagrams a powerful tool for the precise specification of operations and actions. More precisely, we provide means for specifying the semantics of operations in class diagrams and the interpretation of actions on statechart diagrams. Such specifications are able to describe the pre- and post-conditions of operations and actions, their effect on the current state, as well as the calls or signals that are sent during their execution.

Precise semantics of UML collaboration diagrams impacts their use on both the model and the meta-model level. The semantics of actions, like call or send actions, has to be the same in all models, that is, it should be specified once and for all on the meta-level. Action itself is a concept on the meta-model level. As operations and their interpretation differ in every model, they have to be specified on the level of individual models. In both cases, it is desirable to use UML to specify the semantics: In the case of operations, the specification has to be given by software developers, i.e., users of the UML, who should not be forced to learn yet another notation. And by specifying the semantics of actions by UML collaboration diagrams on the meta-level, people without a strong background in formal methods, like tool developers and advanced users, can benefit from the semantics specification given in UML notation, too.

In **[HS01]** we provide the semantics definition for collaboration diagrams based on concepts from the theory of graph transformation. According to the official UML documents, collaboration diagrams specify patterns of system structure and interaction. We propose to use them, in addition, for specifying, pre and post-conditions and state transformations of operations and scenarios. We formalize the three different aspects of a system model that can be expressed by collaboration diagrams – structure, state transformation, and interaction – by means of graph transformation systems and graph processes. We thus integrate the state transformation with the structural and the interaction aspect. Orthogonally, we distinguish three levels of abstraction: type, specification, and instance level. In particular, the idea of collaboration diagrams as state transformations provides new expressive power which had so far been disregarded by the UML standard. The relationships between the different abstraction levels and aspects are described in terms of homomorphisms between graphs, rules, and graph transformation systems.

Graph processes provide truly concurrent semantics to collaboration diagrams, which can be helpful for analyzing the concurrency properties of operations in terms of the associated causal dependencies. In particular, the semantics of signals (implemented as a specific kind of objects) is compatible with that of asynchronous messages in a message sequence chart, given as partially ordered send and receive actions.

The synchronous semantics of operation calls is captured by the substitution of the called diagram for the call in the calling diagram, which is formally described as the composition of two deduction rules over graph transformations. The proof-theoretic interpretation of collaboration diagrams provides a basis for implementation, e.g., in a theorem-prover, logic programming or (conditional) rewriting system. This is particularly important if collaboration diagrams are used for dynamic meta-modeling (DMM, see Section 3.4) to analyze, test, and verify the semantics specification. DMM is used e.g. in [EHHS00] as a meta-modeling approach to the semantics of call actions in statechart diagrams.

3.4 Dynamic Meta Modeling (DMM)

In addition to using meta-modeling according to the four layer meta-modeling architecture of the OMG's Meta Object Facility (MOF) for specifying the ontological and linguistic meta-models in this work, we have developed a formal approach for modeling behavior on the foundation of such meta-models. *Dynamic Meta Modeling* (DMM) [EHS99], **[EHHS00]**, [HS00a], [HHS00] is a method for the formal specification of precise semantics for modeling languages like the UML. It has been introduced as an approach to formalize the *operational semantics* of behavioral UML diagrams in [EHS99], since the UML meta-model captures the abstract syntax and static semantics of UML models by means of (meta-) class diagrams and expressions in the Object Constraint Language (OCL), but it does not cover the dynamic (operational) semantics of its behavioral diagrams.

The approach is founded on a graph-theoretic interpretation of meta-modeling. The meta-model (of UML) is interpreted as a type graph. The abstract syntax of a corresponding (UML) model is interpreted as an instance graph that conforms to the given type graph.

We deploy graph transformation theory for the definition of DMM and combine it with concepts from structured operational semantics (SOS). DMM rules are defined as graph transformation rules typed over the type graph given by the meta-model. The rules are represented in the notation of UML collaborations (i.e., UML 1.x collaboration diagrams or the structural part of UML 2.x interactions). In this way, it is possible to define the behavior of UML diagrams within UML notation. The collaborations are formally interpreted as graph transformation rules. We use their extended interpretation from [HS00b], **[HS01]** that covers pre- and post-conditions as well as state transformations of operations and scenarios (see Section 3.3). By this construction, we use collaboration diagrams that are formalized as graph transformation rules for specifying the operational semantics of graphical modeling languages [EHHS00].

The conceptual idea of the semantics specification is inherited from Plotkin's structured operational semantics (SOS) paradigm, a style of semantics specification for concurrent programming languages and process calculi. It provides powerful techniques and well-

established methodology. The basic idea of SOS is to represent the abstract syntax of program states as terms (abstract syntax trees) and to specify a transition relation on these terms by structural induction. The deduction rules for each syntactic construct define an abstract interpreter for the language.

In DMM, SOS program terms are replaced with abstract syntax graphs of UML diagrams which are augmented with a representation of the state. UML collaboration diagrams are used as deduction rules to specify a goal-oriented interpreter for the modeling language that is defined by the meta-model. The execution of operations is described with simple UML collaboration diagrams (see below) which are formally interpreted as graph transformation rules [EHHS00]. Each syntactic construct is defined separately by a set of deduction rules of a graphical operational semantics. These rules can be interpreted with predicate logics [HHS00]. Since we use graph transformation techniques for specifying the operational semantics, the approach is called *graphical operational semantics* (GOS). GOS can be used for defining the unambiguous execution semantics of graph-based modeling languages, the generation of compilers and interpreters, code generation and model simulation, or the formal analysis of behavioral models [HS00a].

With the use of DMM, dynamic semantics of UML can be both mathematically rigorous so as to enable formal specifications and proofs and, due to the use of UML notation, understandable without prior knowledge of heavy mathematic machinery. Thus, it can be used as a reference by tool developers and advanced users. The use of UML diagrams for defining the semantics of behavioral UML models is also the origin of the name *Dynamic Meta Modeling*. Figure 6 summarizes the semantic and syntactic dependencies of DMM (DMM with Time will be presented in the next section).

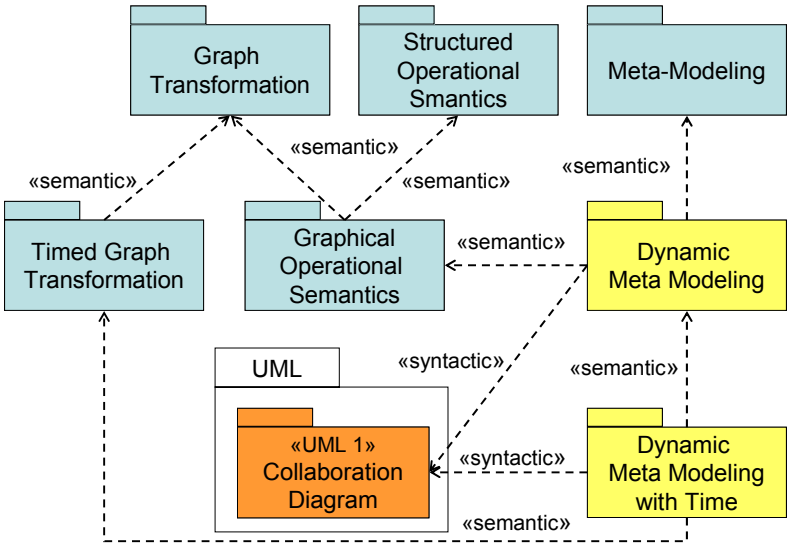


Figure 6: Semantic and syntactic dependencies of Dynamic Meta Modeling (with Time)

An important prerequisite of dynamic meta-modeling is the precise semantic foundation of UML collaboration diagrams. To resolve the generally problematic self-reference in the specification of a specification language, we define the semantics of a core part of language elements of collaboration diagrams (i.e., *simple* UML collaboration diagrams) by a formal interpretation based on GOS. This distinguished set of language constructs acts as a meta-modeling language for the dynamic aspects. This is analogous to the use

of a subset of the UML class diagrams in the Meta Object Facility for specifying the static meta-model of UML. The precise semantics of the subset of UML collaboration diagrams can then be utilized to specify the semantics of UML behavioral diagrams, i.e., specifically for the complete UML collaboration diagrams in the UML itself [HHS00]. DMM has also been applied for specifying a fragment of UML statecharts and object diagrams in [EHHS00].

Altogether, Dynamic Meta Modeling fulfills important requirements for a semantics specification of modeling languages (see [HHS00]):

- DMM is precise and formal due to its formal foundation in graphical operational semantics which combines the theory of structured operational semantics and graph transformation,
- DMM is open since it is based on formal logics,
- DMM is flexible due to its rule-based definition,
- DMM is comprehensible as it is defined as operational semantics and uses meta-modeling concepts,
- DMM is based on graphs with its combined use of meta-modeling and graph transformation.

Tool support and a possible tool architecture based on concepts of logic programming are sketched in [HHS00] as well. The deduction rules are translated into Prolog clauses, and the semantics of graph transformations is described by predicate logics. A Prolog system can then be used as the execution engine that interoperates with a UML modeling tool via XML, using XSL transformations. Theorem provers may alternatively be used – on the model level for analysis and execution of models and on the meta-model level for analyzing the semantics specification itself, for instance with respect to the validity of meta-model invariants.

Two more properties of Dynamic Meta Modeling are worth mentioning here: the ability to specify semantics incrementally and the integrated specification of model transformations and consistency constraints.

Incremental semantics specification. DMM is not only suited for specifying the semantics of the UML (and other modeling languages), but it also accounts for UML's built-in semantic variation points (e.g. compare Section 3.2) and extension mechanisms (outlined in Section 3.1). As a consequence, DMM supports the definition of *user-defined semantics*, e.g., in the context of domain-specific profiles which extend the UML standard by stereotypes, tagged values, and constraints. The semantics specification of such extensions must be formally integrated and consistent with the standard UML semantics without changing the latter. Feasible semantics approaches must allow advanced UML modelers to define domain-specific language extensions in a precise, yet usable manner.

With DMM, it is possible to incrementally specify the semantics of UML extensions. This incremental specification capability is presented in [HHS01]. It is an important property for the specification of the semantics of the multimedia-specific extensions of UML sequence diagrams that we have developed in [HHS02a], see Section 3.5. There we specify the operational semantics of UML sequence diagrams and extend this specification to include features for the modeling of multimedia applications. Our

multimedia sequence diagrams constitute a conservative extension of UML sequence diagrams for the modeling of multimedia presentations.

Consistency checking. DMM can be utilized as a conceptional platform for consistency checking [EHHS02]. We can use DMM rules for testing consistency constraints between the specifications of (partial) models given in different member languages of the UML. The consistency conditions depend on the languages involved, the development process employed, and the current stage of the development. DMM rules provide a formal and precise, yet understandable means for denoting consistency conditions; and they can be easily adapted to new requirements. Due to their syntactic similarity with UML collaboration diagrams, an advanced UML user should be able to understand the notation. For the automatic validation of models according to the consistency conditions, we conceptualize an automated testing environment in [EHHS01] that uses these rules. With this environment, it is possible to automatically check whether two diagrams conform to given consistency rules, thus enabling model improvements.

Consistency of models and model transformations are strongly interrelated topics. It is thus desirable to have a single notation for expressing model properties concerning both aspects. When using meta-modeling techniques, graph transformations are a natural candidate to express model transformations. In [HHS02b], we use graph transformations for denoting both model transformations and consistency conditions between models. This combined use benefits different types of interrelation between transformation and consistency. A special focus is the generation of automatic consistency-establishing transformations. We use relationship patterns to express related elements in models and a graphical specification of consistency conditions with respect to these relationships. The graphical consistency conditions are intuitively understandable and especially suited to express complex object structures. They can, in addition, be equipped with mechanisms that facilitate the automatic correction of certain inconsistencies. This property that consistency and transformations can be studied in a single language benefits all scenarios in which consistency and transformation interact.

In summary, graph transformation is used in this work as the theoretical foundation for model-based software engineering. Models are represented as graphs, and graph transformations are used to specify model transformations as well as their pre- and post-conditions. The transformations can be interpreted as defining precise operational semantics of the models. They can also be used to translate between different partial models, as in the case of automatic code generation from design models (compare Section 4). From a methodological point of view, graph transformation can furthermore be deployed for graphically defining relationships between model elements and consistency constraints between partial models, for checking these conditions and to correct inconsistencies if required [HHS02b]. Thus, transformations and consistency management can be handled in a common formalism.

3.5 Dynamic Meta Modeling with Time (DMM+t)

In system domains like embedded real-time systems and multimedia applications, it is important to include specifications of time in the behavior models, since their correct execution depends on the fulfillment of time constraints in addition to functional requirements. UML already incorporates (syntactic) language features to model time

and temporal constraints. Obviously, such model elements must have an equivalent in the semantic domain.

Together with Hausmann and Heckel, I have extended the DMM approach to DMM+t, *Dynamic Meta Modeling with Time* [HHS02a], **[HHS04]**. We make use of DMM's extensibility as explained in Section 3.4. It allows us to define the operational semantics of UML diagrams with time specifications incrementally. As the semantic domain, we now use timed graph transformations. They extend the formalism of attributed graph transformations by distinguished attributes for time. Transformations get a time stamp and additionally can not only modify the graph structure, but also the value of time attributes. By this means, we are able to define the operational semantics of UML diagrams with time.

We have applied DMM+t in multimedia application modeling in [HHS02a], **[HHS04]**. We describe the semantics of time aspects in (extended) UML models (now with timed execution semantics) and check temporal consistency of different partial models. For this purpose, we have defined multimedia sequence diagrams, which are a multimedia-specific variant of UML sequence diagrams. They allow us to model the control of multimedia presentations. The DMM rules with time then specify an interpreter that can be used to analyze or test a model of multimedia sequence diagrams. Timed automata have been investigated an as alternative semantic domain.

Integrating these works with the object-oriented modeling method OMMMA (see Section 5.2), we obtain a comprehensive method for the model-based development of multimedia applications. OMMMA, on the syntactic level, defines the required extensions of the UML for the modeling of multimedia applications, using UML's built-in extension mechanisms for defining UML profiles. This defines the graphical syntax of the visual modeling language. Dynamic Meta Modeling (DMM, DMM+t) provides the formal means for defining the semantics. Using this formal method, we obtain precisely defined semantics of the multimedia-specific extensions of UML sequence diagrams.

4 Fundamental Methods for Software Engineering

In this section, we look at two fundamental approaches towards model-based and model-driven software development: automatic code generation and design-by-contract with models. In Section 4.1 we present a method for automatic generation of Java code from UML 1.x collaboration diagrams. The remaining sections present the idea of visual contracts and their application in different software development domains. Visual contracts apply concepts of design-by-contract on the level of models. Their formal foundation is given by graph transformation theory. We have successfully applied visual contracts as a method for the model-driven monitoring of programs (Section 4.3), Web service discovery and validation (Section 4.4), the specification of enterprise services (Section 4.5), and for behavior specifications in model-based testing (Section 4.6). The characteristics of the presented methods and the relevant publications are summarized in Figure 7.

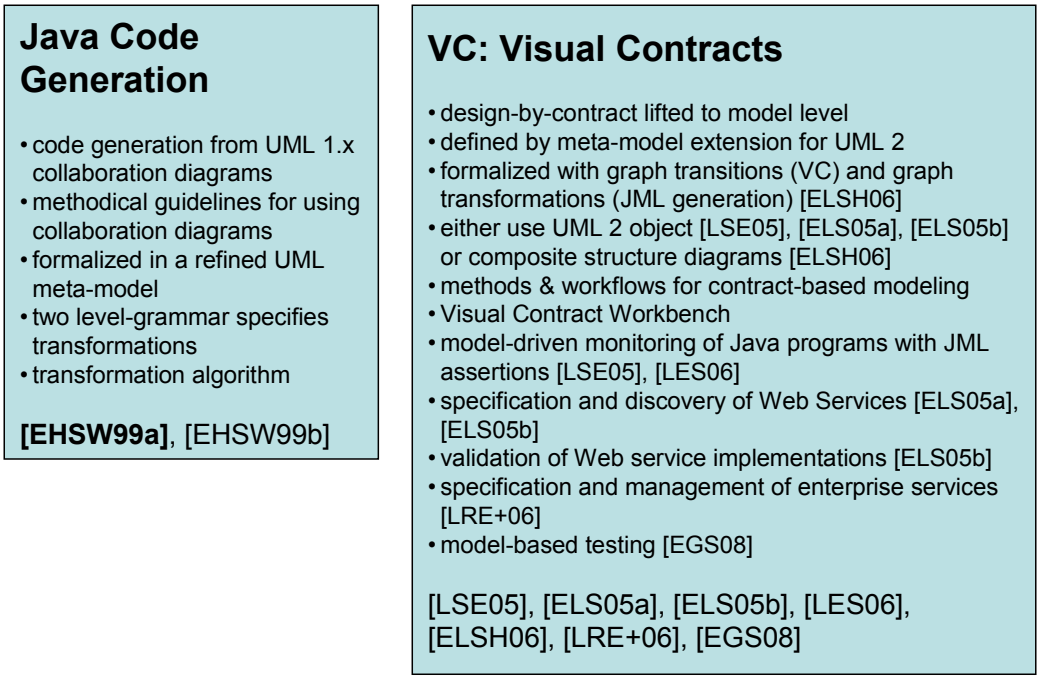


Figure 7: Overview of the contributed fundamental methods

4.1 Java Code Generation from UML Behavioral Models

A different line of research, but starting point for the precise UML work is the rule-based transformation of UML models in executable program code. The definition of such transformation rules is the basis for automatic code generation and an important aspect of rigorous and model-driven software development. In **[EHSW99a]** and [EHSW99b], we provide a translation of UML 1.x collaboration diagrams in Java code. The translation process is conceptualized and formalized. Graph transformation rules are used that operate on a representation of a model as an instance of the UML meta-model.

UML comprises a family of diagram types for specifying structure and behavior of software systems. Models specified in these diagram types have to be transformed into

corresponding code during the software development process. While previously mainly class diagrams and statechart diagrams were considered for automatic code generation, we focus on collaboration diagrams in **[EHSW99a]** and [EHSW99b]. We investigate the modeling of sequential behavior by UML collaboration diagrams and their automatic transformation into Java code. The code generation from the collaboration diagrams is closely related to the generation of code from class diagrams as explained in [EHSW99b].

Objective of our transformation is to preserve the modeled information during the transition from a model to its implementation. The automatically generated Java code fragments implement a substantial part of the system functionality. However, we do not use UML collaboration diagrams as a visual programming language for specifying the behavior of a system completely. Instead we concentrate on the modeling of object interactions, while computations on data values are neglected, and thus have to be manually added to the generated Java code.

According to this perception of code generation, we first provide methodical guidelines how to deploy collaboration diagrams in a structured way for modeling functional behavior. This is an important prerequisite for a consistent transformation into Java code. This methodical understanding is formally reflected in a refined meta-model. On this basis, we then formulate the transformation algorithm.

We build our transformation on a two-level grammar approach. It uses a kind of *meta rules* consisting of a rule scheme and an additional pattern **[EHSW99a]**. The *rule scheme* describes the generation of syntactically correct Java code. It has the form of a context-free rule expression, but it is still independent of a concrete collaboration diagram. The *pattern* is a part of an instance diagram of the meta-model. It is used to represent those parts of a concrete diagram which shall be actually transformed. Hence, the occurrence of the pattern in the instance diagram for the example application for which code shall be generated serves as an application condition for the whole meta-rule to be applied. Moreover, the concrete occurrence links together the general code generation as described by the rule scheme, and the actual elements of the concrete collaboration diagram that has to be transformed. The parameters of the rule scheme occur in the pattern and can hence be replaced by actual values in order to instantiate the rule scheme.

4.2 Visual Contracts (VC): Design-by-Contract with Models

Visual contracts are a fundamental software engineering method and provide a formalism for model-based and model-driven development. They apply concepts of design-by-contract on the level of models. The formal foundation of visual contracts is given by graph transformation theory [ELSH06].

We use *executable* visual contracts at runtime for monitoring the execution of programs. We call this approach *model-driven monitoring*. For this purpose, visual contracts are translated into assertion code that is executed together with the functional code for its monitoring [LSE05], [LES06]. We use this fundamental software engineering method for the semantic specification of Web services, the matching of service descriptions and service requests [ELS05a], [ELS05b], and the model-based development and model-driven monitoring of Web services [ELS05b]. We have applied visual contracts in an

industrial case study in order to evaluate its applicability to the development and description of enterprise services [LRE+06]. Additionally, we employ visual contracts for formalizing functional requirements in order to use them for software testing [EGS08]. All these uses will be explained in the next sections.

4.3 Executable Visual Contracts for Model-driven Monitoring

Design-by-contract is widely acknowledged to be a powerful technique for creating reliable software. It allows developers to specify the behavior of an operation precisely by pre- and post-conditions. We have developed an approach to lift the design-by-contract idea, which is usually used at the code level, to the model level. For this purpose, v*isual contracts* [LSE05] are introduced as a specification technique. They are used to graphically specify data state transformations with pre- and post-conditions. The pre- and post-conditions of behavioral elements, e.g. operations or services, are modeled by pairs of UML object diagrams [LSE05] or pairs of UML composite structure diagrams [ELSH06].

In [LSE05], we define a mapping of visual contracts into Java classes that are annotated with behavioral interface specifications in the Java Modeling Language (JML). The mapping supports testing the correctness of the implementation against the specification using JML tools, which include a runtime assertion checker. Thus we make the visual contracts executable.

By using the UML, we build on a well-known standard that is predominantly used in today's model-based and model-driven development methods. Hence, our visual contracts are understandable by software developers and can be easily integrated in model-driven software engineering methods and processes. Visual contracts also qualify for adoption in agile methods as they can be used as the origin for programmers' coding and test generation. Furthermore, our visual contracts are more intuitive and easier to understand than logic formulae (normally used for design-by-contract) and, in consequence, more efficient for information exchange among different development team members.

To formalize visual contracts, we have defined a UML 2 meta-model extension for visual contracts. The visual contracts integrate with the UML 2 meta-model. We mainly use elements from the UML 2 meta-model packages `InternalStructures` and `Collaborations`. The operational specification of the transformation from visual contracts to JML code is based upon this UML 2 meta-model extension for visual contracts. The meta-model represents the source language of the model transformation and provides the type graph on which the graph transformation rules operate, i.e., the graph transformation rules are specified on the meta-model level, and the concrete models are viewed as meta-model instances when they are transformed. Figure 8 summarizes the semantics and semantic dependencies of visual contracts.

The conceptual idea and modeling language definition for visual contracts is accompanied by a model-driven monitoring method that specifies how to work with visual contracts (see Figure 9). Since visual contracts are used on the model level, contract-based development in this case is a specific form of model-based software development. Furthermore, as visual contracts are used as an input for generative software development, too, modeling with visual contracts can even be considered a

model-driven software development method. Thus, it supports model-driven development (MDD) of software systems by lifting the design-by-contract idea from the code level to the model level.

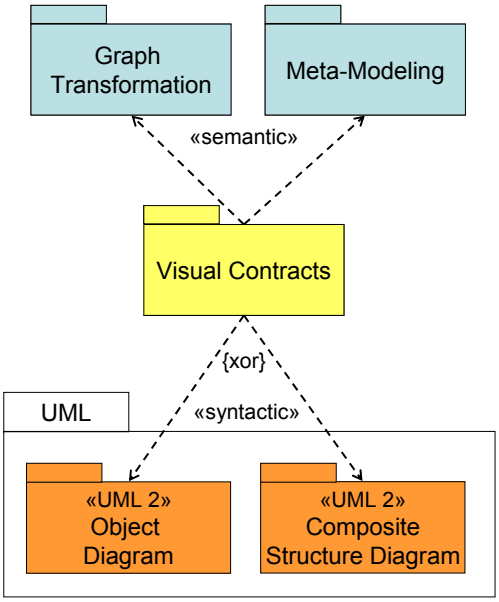


Figure 8: Semantic and syntactic dependencies of the visual contract formalism

When applying visual contracts in a design-by-contract method, a designer models the design class diagram and specifies the behavior of operations (or services) by visual contracts. Program classes with assertions are then generated from the class model and the visual contracts. More precisely, the design class diagram is translated into class templates in the target programming language, and assertion code (i.e., executable contracts) is generated for the operations' behavior. A programmer uses the visual contracts as a specification and fills the missing behavioral code in the class skeletons (this activity can be regarded as model-based). To validate that the programmer's code is a correct implementation of the visual contract, the assertion code is used for runtime monitoring of the system's behavior. Altogether, the code generation facilitates automatic monitoring of the correctness of the programmers' implementation.

The feasibility of the approach is shown in [LSE05] by a translation of our models into Java. The visual contracts are translated into JML, a design-by-contract extension for Java. Then we use the JML compiler, which translates the JML annotations into executable bytecode. The compiled bytecode contains checks to test the pre- and post-conditions at runtime. A translation of our visual contracts to OCL or Microsoft's Spec# is also possible.

For an efficient deployment of our model-driven development method, we have built a comprehensive tool support in the *Visual Contract Workbench* [LES06]. The tool is implemented as an Eclipse plug-in and uses the Eclipse Modeling Framework. An editor allows developers to coherently model class diagrams and visual contracts [LSE05]. The editor is complemented by code generation facilities that translate the model in Java classes with assertions for their operations. The visual contracts are translated into assertions in the Java Modeling Language (JML).

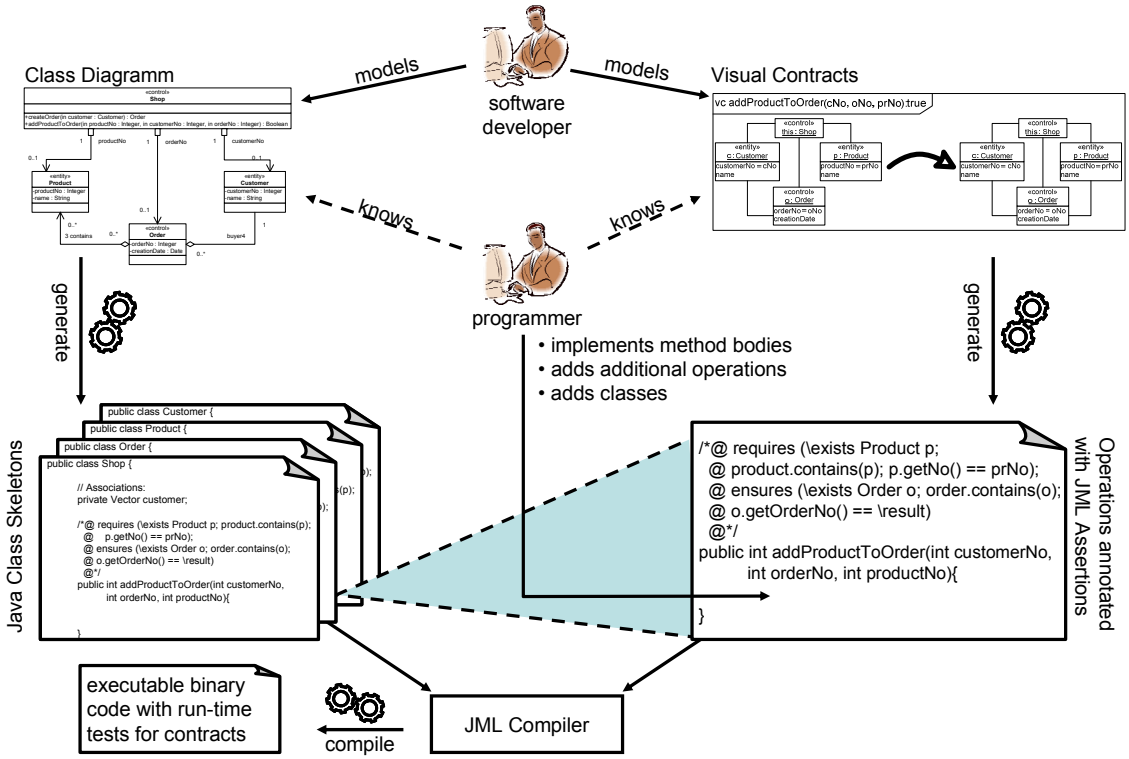


Figure 9: Overview of the design-by-contract method with visual contracts (from [LSE05])

The presented model-driven development (MDD) approach for constructing software systems advocates a stepwise refinement and transformation process starting from high-level models to concrete program code. In contrast to numerous research efforts that try to generate executable function code from models, we propose a novel approach termed *model-driven monitoring* (MDM) [ELSH06]. Models are used to specify minimal requirements and are transformed into assertions on the code level for monitoring hand-coded functional code during execution.

We deploy graph transformation theory for supporting our model-driven monitoring approach. In particular, models in the form of visual contracts are defined by graph transitions with loose semantics, while the automatic transformation from models to JML assertions on the code level is defined by strict graph transformation rules. Since the pre- and post-conditions of the visual contract only specify minimal requirements towards the implementation of an operation, we use the loose semantics of graph transitions of the double-pullback approach. For the model-to-code transformations, we use compound graph transformation rules to define a transformation of our visual contracts to JML. To automate this model transformation, we need the strict semantic interpretation of graph transformation rules as formalized by the double-pushout approach. Both aspects are supported and realized by the dedicated Eclipse plug-in.

Altogether, model-driven monitoring is a practically useful amalgamation of graph transformation and design-by-contract concepts. In contrast to the automatic generation of function code, we generate assertions from contracts that are monitored and automatically checked while the actual and manually implemented function code is executed.

4.4 Web-Service Discovery and Validation with Visual Contracts

The quality of service-oriented software systems depends substantially on linking the proper services. Two fundamental aspects are relevant: (1) Do the requirements of a service requestor and the service description of a service provider fit together? (2) Is the service implementation correct with respect to the service description?

If a service requestor wants to find a Web service that is offered by a service provider, the requestor's requirements and the description of the service must be compared. Syntactic descriptions are not sufficient for this. In [ELS05a] we apply design-by-contract on the modeling level for the semantic specification of Web services. We employ visual contracts for the semantic description of both Web services and service requests, and we introduce a matching mechanism for the comparison of requestor and provider contracts in [ELS05a], [ELS05b]. This enables an automated semantic search for Web services.

Pre- and post-conditions of a visual contract are each specified in the form of a UML object diagram. The object diagrams are typed over a class diagram. The class diagram acts as an ontological model that defines the terminology. Based on this specification of service behavior, we define the notion of compatibility. Compatibility of provided services and service requests is traced back to checking of sub-graph relationships [ELS05a]. The precondition of the provider contract has to be a sub-graph of the requestor contract's precondition, and vice versa for the post-conditions. The notion of compatibility enables the matching of service descriptions and service queries at the discovery service. The notion of compatibility accounts for structural changes such as the deletion or creation of an object. Focusing on structural changes allows us to effectively compute the matching between descriptions of service requests and Web services.

The approach may be extended e.g. with logic expressions to restrict the values of object attributes. However, this will make the computation of the matching more complicated. From our perspective, this is not required for the compatibility checks at the discovery service, since the detected services are only candidates. A service requestor has to take further steps to make a selection from the candidate set. Because only a small number of candidate services then need to be further examined, additional criteria may be considered for the selection.

Additionally, we have developed a method for generating a matching that allows us to map the contracts that are typed over the class diagrams of service requestor and service provider, respectively, to a semantic description that is typed over a domain ontology [ELS05b]. Therefore, Web services can be offered and used across domains.

A model-based method and process for the contract-based development and management of Web services in presented in [ELS05b]. In addition to the matching process, we generate analyzable semantic descriptions and runtime assertions from Web service models (i.e., visual contracts), and use them for checking the correctness of a Web service implementation against its specification at execution time. The process is depicted in Figure 10.

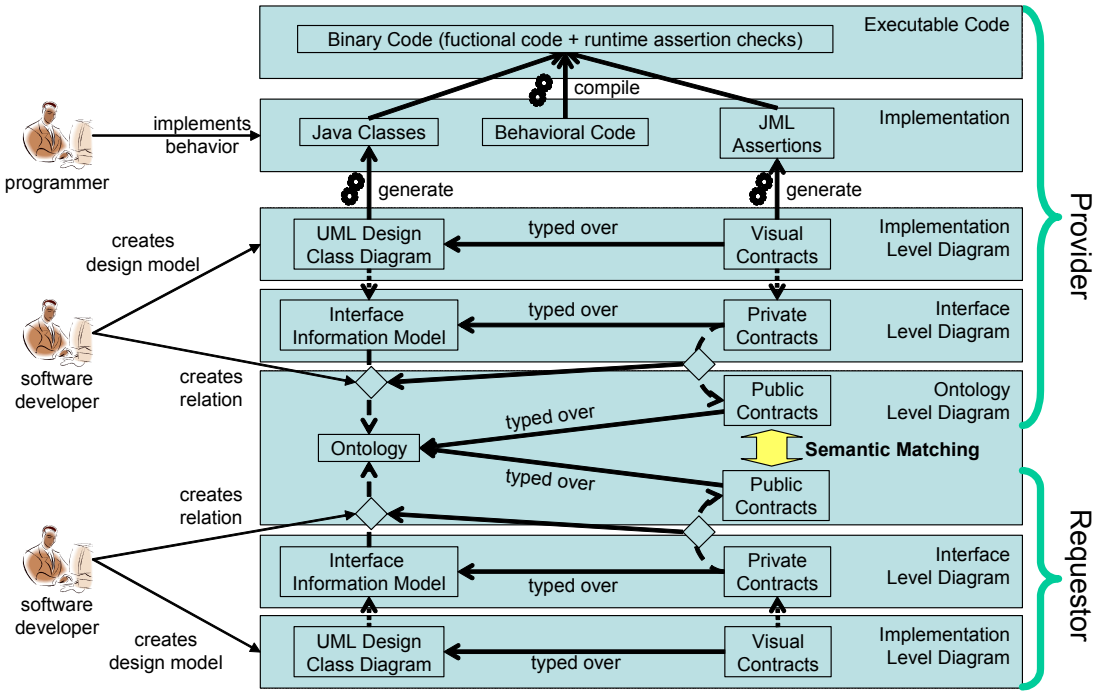


Figure 10: Specification, search and validation of Web services with visual contracts (from [ELS05b)]

The conceptual approach is complemented by implemented software tools for comparing the contracts and the generation of Java and JML code. The tool chain for the contract matching contains The Attributed Graph Grammar System (AGG) as the modeling editor for visual contracts. AGG supports typed graph transformation. The type graph is an abstract representation of the used ontology (class diagram) in our case. The ontology is represented by the semantic Web languages DAML+OIL. Visual contracts are represented as AGG typed graph transformation rules. A Java application translates between DAML+OIL and AGG's file format. The DAML+OIL representations of the ontology and the pre- and post-conditions of the visual contract are RDF graphs. Thus, we can map our matching concept for contracts to the matching of RDF graphs. The semantic Web framework Jena is used for computing the sub-graph relation.

Altogether, we have developed a model-based and partly model-driven approach for the semantic description, comparison and validation of Web services. Our solution does not only support the semantic comparison of requirements and service descriptions, but also the model-based validation of service implementations. Hence, the method supports constructive and analytical quality assurance which has the potential to reasonably improve the quality of service-oriented applications.

4.5 Specification of Enterprise Services with Visual Contracts

Conceptually, service-oriented architectures (SOA) allow for fast and cost-effective appropriation of functionality to support the business processes of a company. Required business functions are provided as enterprise services and can be flexibly deployed in service-oriented business information systems. However, the large count of such services demands for appropriate semantic descriptions for their efficient management.

We have evaluated the practical applicability of visual contracts for service management in an industrial case study together with a software company [LRE+06], using a realistic scenario from the insurance business. Visual contracts are employed as a UML-based technique for the semantic description of enterprise services and the formulation of service queries on the modeling level.

The evaluation shows that visual contracts can be used practically and economically on the business level. On the technical level, extensions are required if visual contracts are to be used for the specification of service interfaces beyond a certain degree of complexity. However, the increased expressiveness competes with the intuitive understandability of the contracts. If the visual contract formalism is extended, then the matching mechanism must also be adapted.

4.6 Model-based Testing with Visual Contracts

Visual contracts can also be applied in the methodical domain of model-based testing. In order to effectively employ use-case descriptions for software testing, the pre- and post-conditions of the use cases that are given in natural language are formalized by visual contracts [EGS08]. The visual contracts specify the modification of business objects as a result of executing the use case. The visual contracts can then be used for generating test inputs during test-case specification and for checking test outputs during test execution. Tool support is provided for linking visual contracts into the development and test processes. Particularly, we have extended the Visual Contract Workbench with a plug-in for the generation of test cases and the execution of tests.

After having considered fundamental, widely applicable approaches to model-driven development, namely automatic code generation and design-by-contract, in this section, we look at software engineering methods that are targeted towards specific types of systems, i.e., system domains, in the following sections. We start with the model-based development of multimedia applications and then look at software engineering methods for other system classes that I have developed or contributed to.

5 Multimedia and Interactive Systems

In this section we look at the system domain of multimedia and advanced interactive systems. We consider multimedia applications and software systems with advanced user interfaces to be members of this system domain. Characteristic is the importance of the user interface, and thus its integrated modeling in model-based development methods; furthermore, the tendency towards integrated, interdisciplinary development methods that let designers and end-users participate in the development process.

In this domain, work on multimedia software engineering methods and predominantly model-based development methods and processes for multimedia applications (Sections 5.1, 5.2, 5.3), the integration of informal methods with software engineering methods (Section 5.4), as well as the model-driven development of interactive multimedia systems and Web applications (Sections 5.5 and 5.6) will be presented. An overview of the contributions and their dependencies is given in Figure 11.

5.1 Multimedia Software Engineering Methods

Multimedia applications are interactive software systems that use, manipulate or produce media of different types in an integrated way. According to this definition, they are a specific type of software systems. The complexity of modern multimedia systems and their distributed development by heterogeneous teams demands for systematic development methods that account at the same time for their specific characteristics. We call this systematic development *multimedia software engineering*. From a software engineering perspective, it is necessary to apply and adapt software engineering techniques in this system domain in order to account for the multitude of relevant aspects and to keep multimedia software development projects manageable (see [ES04a]).

My work on multimedia software engineering methods started in the late 1990s from the observation, that a variety of techniques, services, systems technology and standard document formats for multimedia computing and communication had been developed in the 1990s. Despite the technological progress, the discipline of multimedia software engineering did not receive comparable attention. Multimedia application development typically followed an implement-and-test process: specialized multimedia authoring systems, multimedia frameworks, toolkits or system software were directly used for implementation. Among them, multimedia authoring tools were the dominant means for prototyping and directly creating interactive multimedia and hypermedia applications. The brisk development of multimedia frameworks and toolkits of the early 1990s had widely ebbed away, but had fostered the provision of multimedia APIs such as Microsoft's DirectX or Sun's Java Media API to simplify multimedia programming. An analysis of the state of application development based on multimedia authoring systems and object-oriented programming can be found in **[ES02]**.

However, while authoring and programming of hypermedia and multimedia applications were successfully adopted in practice, modeling and adequate specification techniques for multimedia systems needed to further evolve. No preceding modeling activities for requirements specification, analysis, or design of the system prior to the implementation were included in the multimedia software development process. Formal models without direct relation to development methods and implementation

technologies did not gain much practical relevance; yet they provided the conceptual basis for later model-based development approaches. Sophisticated multimedia process models and established, usable graphical notations tailored to the specification of multimedia systems were still lacking.

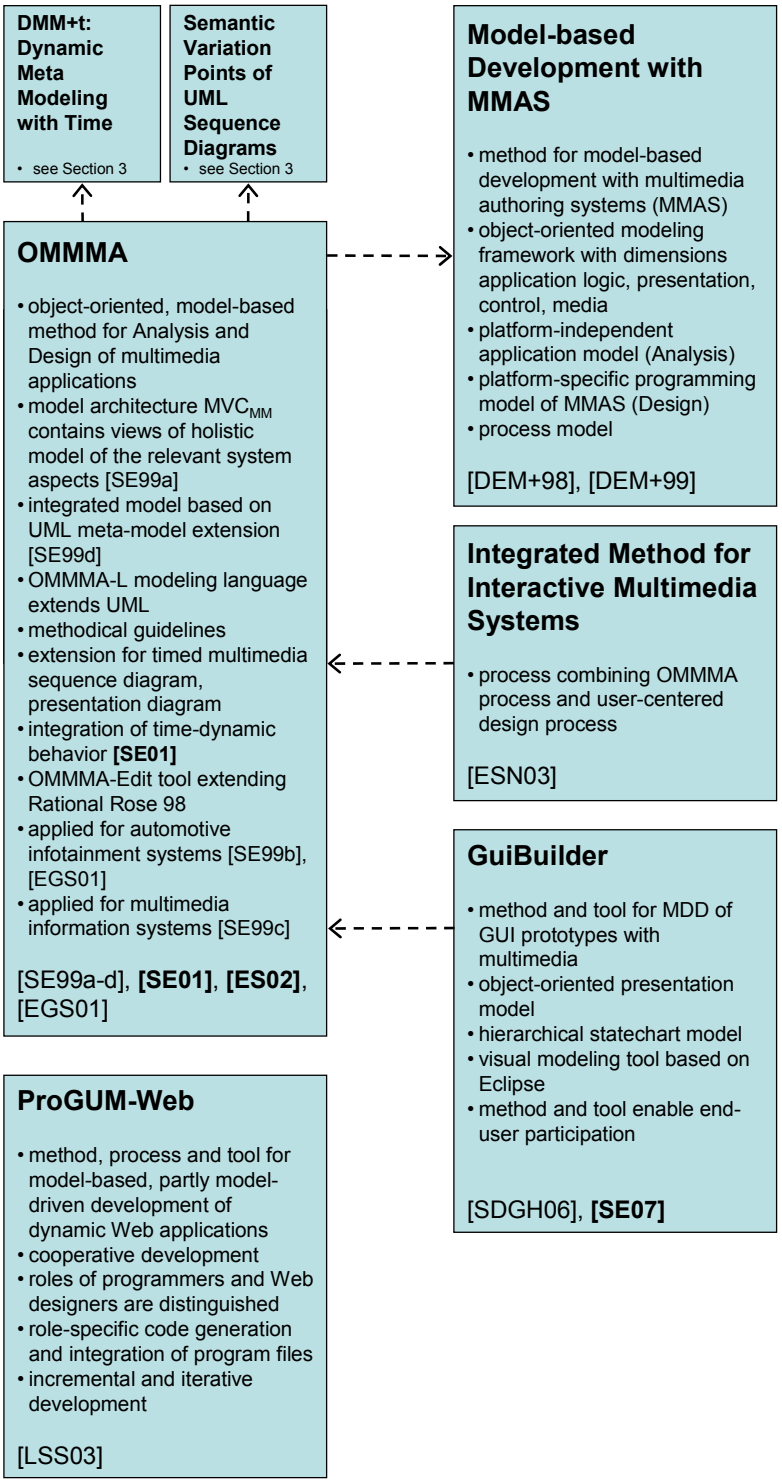


Figure 11: Overview of the work contributed to model-based and model-driven development in the domain of multimedia and interactive systems

The main focus of research on multimedia software engineering at that time was the development of hypermedia applications [ES04b], being a practically important subset of multimedia applications. Some authors worked on the model-based development of hypermedia applications. They proposed concepts and developed tools that support the automatic generation of prototype hypermedia applications using XML technology. Achievements in object-oriented modeling of multimedia applications were surveyed in [ES02]. In our analysis, we considered approaches for the object-oriented modeling of hypermedia applications and extensions of the Unified Modeling Language (UML) for hypermedia and interactive systems.

Under these circumstances, I started research on model-based development of multimedia applications. The most striking result of that work is the OMMMA method and language for object-oriented modeling of multimedia applications that is described in the next section. Addressing the general need for multimedia software engineering, we proposed to make modeling a central technique and activity in the multimedia software development process, even when multimedia authoring tools are used for implementation. An according method and process is summarized in Section 5.3.

5.2 Object-oriented Modeling of Multimedia Applications

As our main contribution to model-based development in the system domain of multimedia applications, we have developed the model-based development method for multimedia applications OMMMA [SE99a][1], [SE99c], [SE99d]. OMMMA is an acronym for object-oriented modeling of multimedia applications. At the core of the OMMMA method is an integrated graphical model that coherently captures the important aspects of multimedia applications. In [SE99a] and [SE99d] we investigate on the methodical level, how far the modeling of multimedia applications, their structure and behavior, goes beyond the modeling of conventional software systems. The analysis yields that aspects of the user interface and the time-dynamic behavior ought to be integrated parts of the multimedia application model. As a solution, we propose the OMMMA method. OMMMA is based on a fundamental modeling architecture that extends the model-view-controller architecture pattern towards multimedia. We call this modeling architecture MVC_{MM} [SE99a].

The integrated model combines a set of different partial models (i.e., modeling views) according to MVC_{MM}. Each of them is dedicated to modeling a particular aspect. The collection of aspects covers application structure, application behavior, media and the user interface. Furthermore, we have identified and incorporated the relationships and inter-dependencies among the individual partial models in the integration model **[ES02]**, given by the common meta-model. The different partial models and their interrelationships are depicted in Figure 12.

The meta-model is also the linguistic foundation for the corresponding modeling language. Used as a linguistic meta-model, the OMMMA meta-model defines the interplay between the model elements that possibly appear in different diagram types (i.e., sub-languages). We defined the visual modeling language OMMMA-L as part of

[1] Recognized with Most Influential Paper Award of IEEE Symposium on Visual Languages and Human-Centric Computing 2010

the OMMMA development method for the object-oriented modeling of multimedia applications **[ES02]**. OMMMA-L is an extension of the UML. With this extension, it is possible to model the important aspects of a multimedia application in a coherent graphical model.

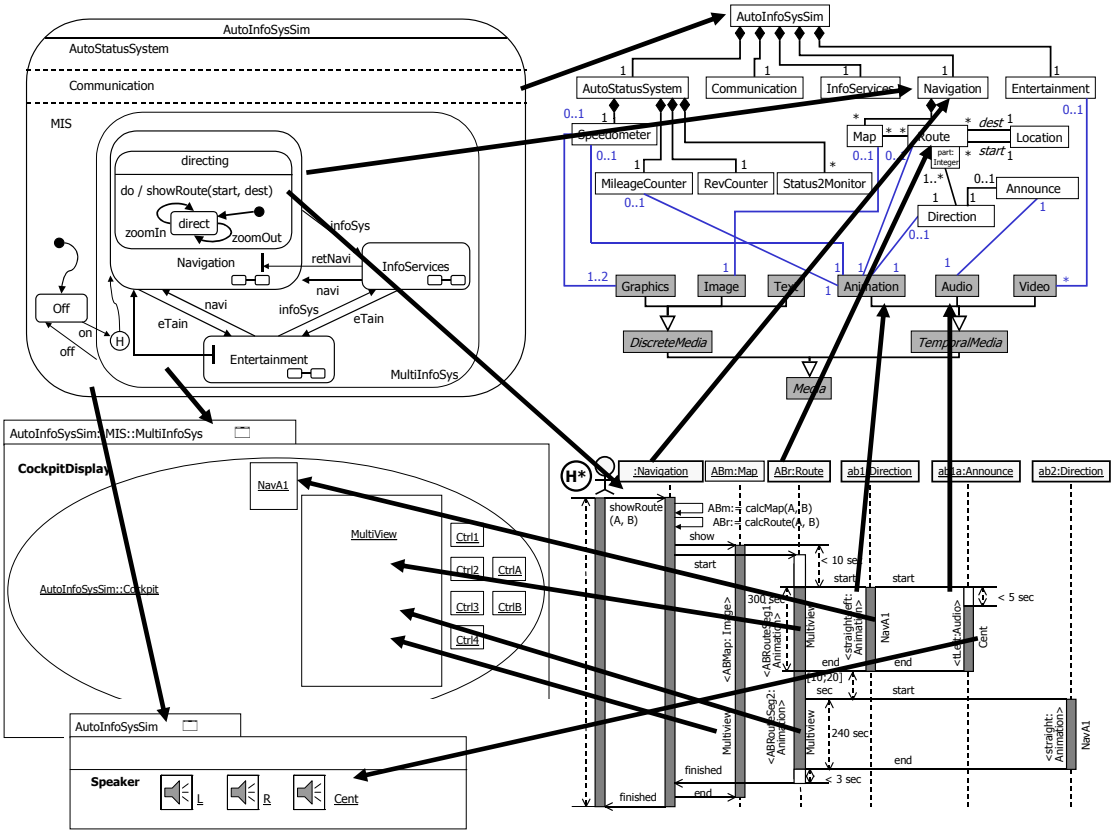


Figure 12: Integrated view of an OMMMA model showing the relationships between dynamic behavior model (top left), timed sequential behavior model (bottom right), class model (top right) and presentation model (bottom left) (from **[ES02]**)

Modeling in OMMMA is based on a separation of the aspects structure, timed behavior, dynamic behavior, and presentation. Timed behavior refers to predetermined temporal behavior of temporal media or media presentation schedules. Dynamic behavior is induced by interactive control of users or the occurrence of other kinds of events. The presentation aspect covers structure and layout of the (graphical) user interface and its elements. Each modeling view is modeled with a dedicated diagram type. A core feature is the integration of the two behavior aspects time and interactivity. This is especially important in the presence of temporal media which have a predefined temporal behavior. In a multimedia presentation, they may be reproduced without change, or they may be altered in response to dynamically occurring events at runtime like user interactions. Thus, predefined temporal behavior has to be combined with user interaction which may happen non-deterministically in an integrated and consistent model. To fulfill this requirement, we couple timed sequence diagrams that have been

extended for multimedia (compare Sect. 3.5) with UML statechart diagrams via a model interface defined in the modeling language **[SE01]**[2].

OMMMA-L is based for the most part on the standard modeling language UML. This especially holds for the notation and the graphical representation of the diagram types. For the definition of OMMMA-L as an extension of UML, we have analyzed the structural and behavioral diagram types of the UML (compare [EHS00], [HKS01]) according to the requirements of the system domain multimedia applications. Where necessary, we have adapted or extended the UML and its diagram types on the level of abstract and concrete syntax, respectively. New language elements have been incorporated into OMMMA-L in order to provide appropriate language features for the specific properties of multimedia applications and to allow for integrated modeling of all relevant aspects. Among these extensions are:

- extensions of UML sequence diagrams for modeling presentation and synchronization of media objects in a multimedia application, and
- the presentation diagram, which supports the spatial representation of media and presentation objects.

Since integration of co-existing timed procedural and interactive behavior is at the heart of multimedia systems, we specifically focus on UML-based specification of behavior in **[SE01]**. We define in detail the extensions to UML behavior diagrams for the modeling of interactive multimedia applications. In addition, we outline from the language and method perspective how these behavioral aspects are to be integrated with media, presentation, and software architecture modeling to achieve a coherent and consistent model. We show that the integration of predefined temporal behavior and interactive control can be easily mapped to object-oriented implementation techniques and frameworks and the development paradigms of most multimedia authoring systems.

OMMMA models can be used on the analysis and design stages of a model-based development process for multimedia applications. To specify its intended use, the modeling language OMMMA-L is accompanied by a method description and methodical guidelines how to deploy the language in a model-based multimedia software development process. Methodical guidelines also state how to integrate the different diagram types for the holistic modeling of all aspects of a multimedia application in a comprehensive model. For example, the class diagram is used to model media types in a hierarchical media type model as well as the logical structure of the multimedia application. Behavior is modeled with specialized multimedia sequence diagrams and statechart diagrams. A new diagram type is added, the presentation diagram, for modeling the (visual) presentation and interactive behavior elements in an integrated and demonstrative way. It contains the elements of a multimedia application user interface together with its spatial properties. In [SE99d], we describe the individual diagram types and present a meta-model for OMMMA-L that is derived from the UML meta-model. The formal semantics of UML behavior diagrams and their extensions for the multimedia domain is defined based on the concept of Dynamic Meta Modeling (see Section 3). In particular, the language extensions of OMMMA-L are formalized by a

[2] Recognized with Best Paper Award of HCC'01 Symposium on Visual/Multimedia Approaches to Programming and Software Engineering

precise semantics specification based on graphical operational semantics [EHHS00], Section 3.5).

In addition to method and language engineering, we have developed the modeling environment OMMMA-Edit as a research prototype. OMMMA-Edit is an extension of the commercial UML modeling tool Rational Rose 98. OMMMA-Edit includes additional visual, syntax-directed editors for the new, resp. extended diagram types of OMMMA-L (see Figure 13) according to the modeling language specification and the defined methodical guidelines. The modeling tool especially assures the consistency between the different diagram types within a model as defined in the integration model. Rational Rose 98 is also deployed as the repository for managing model data.

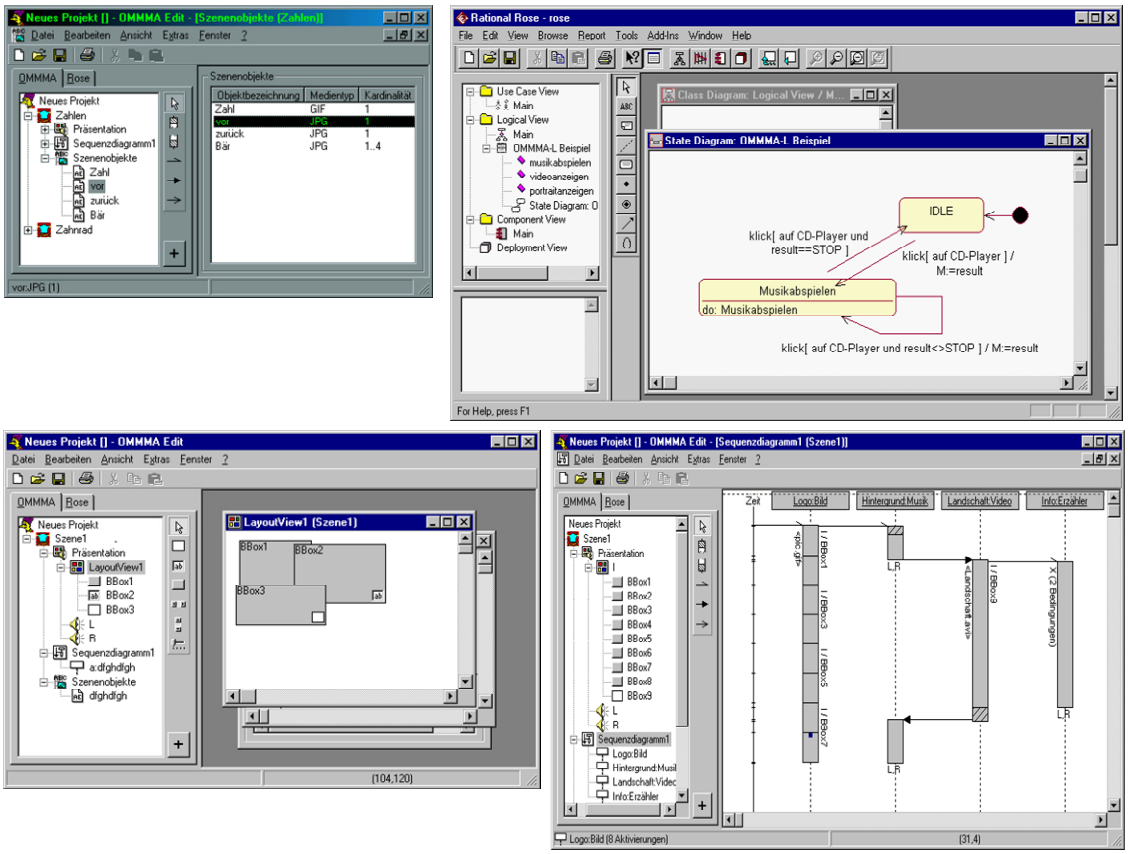


Figure 13: Screenshots of the OMMMA-Edit tool showing the project and diagram type editors

The OMMMA method and language have been applied, evaluated and specialized in different scenarios, e.g., in the domains of multimedia information systems and automotive infotainment systems.

Automotive infotainment systems. OMMMA-L has been evaluated in an industrial joint project for a domain-specific modeling language for automotive software systems. The integrated language was intended to cover aspects of interactive systems, multimedia information (infotainment) and embedded systems [SE99b]. One specific requirement was the modeling of the interactive graphical user interface of the multimedia infotainment system within automotive cockpits [EGS01]. From the identified set of relevant aspects to be modeled, we derived a modeling architecture, defined adequate diagrammatic modeling (sub-) languages extending the UML, and

guidelines for using the modeling language elements. We compared two well-known architecture patterns (MVC and PAC) [SE99b] and derived the Model-View-Controller-Communication (MVCC), an advanced architecture paradigm applicable to real-time, embedded multimedia systems. We specifically reused the OMMMA-L presentation diagram.

Multimedia information systems. We also applied OMMMA for the modeling of structure and dynamic behavior of multimedia information systems in [SE99c]. There, we relate OMMMA-L to basic concepts of multimedia information systems and services.

In summary, OMMMA-L is a precise, yet usable modeling language for the integrated specification of multimedia applications based on software engineering principles and methods. It is an extension of the Unified Modeling Language (UML) based on an object-oriented development method. We conclude in **[SE01]** that the OMMMA approach meets the following requirements:

- the diagrammatic notation is understandable even by non-technical members of a development team or users,
- even large scenarios are still manageable due to the inherent structuring on the structural and behavioral levels by modularization and nested state-machines that are coupled with sequence diagrams,
- models are easily mapped to an object-oriented implementation or the concepts of many multimedia authoring tools, but still no programming language knowledge is needed for the task of modeling,
- temporal (as well as spatial) constraints can be intuitively expressed,
- although temporal and spatial constraints are contained in different diagrams, they are parts of a common integrated model,
- the object-oriented classification of different stereotypes offers an adequate media abstraction and supports architectural decomposition,
- dedicated diagram types enable separation of concern for the different aspects of multimedia applications,
- language (and methods) are easily extensible and customizable by applying UML's built-in extension mechanisms for stereotyping, tagged-values, and constraints,
- the modeling process can thus be very flexible.

More details on OMMMA can be found in the two attached publications **[SE01]** and **[ES02]** in Part III of this thesis.

5.3 Model-based development with Multimedia Authoring Systems

In addition to method and language engineering for object-oriented modeling, my colleagues and I have also dealt with the issue of process engineering for multimedia software development. Outcome of that research is a process model for the model-based development of multimedia applications with authoring systems [DEM+98], [DEM+99].

Authoring systems allow multimedia developers ad-hoc development of applications in a sophisticated end-user programming style, using dedicated visual languages and typical metaphors. Due to the use of these tools and the lack of established software engineering processes and methods for multimedia applications, the multimedia software development process typically used to be limited to the implementation stage. But the lack of conceptualization and documentation leads to well-known software development and maintenance problems.

Based on this observation, we have developed a process model for the model-based development of multimedia applications. It is centered upon analysis and design phases that are directed towards an implementation with multimedia authoring systems. For multimedia applications, object-oriented models are produced on the type and instance level (see Figure 14). The development method is model-based. We distinguish between the platform-independent application model and the platform-specific model for a particular authoring system [DEM+98].

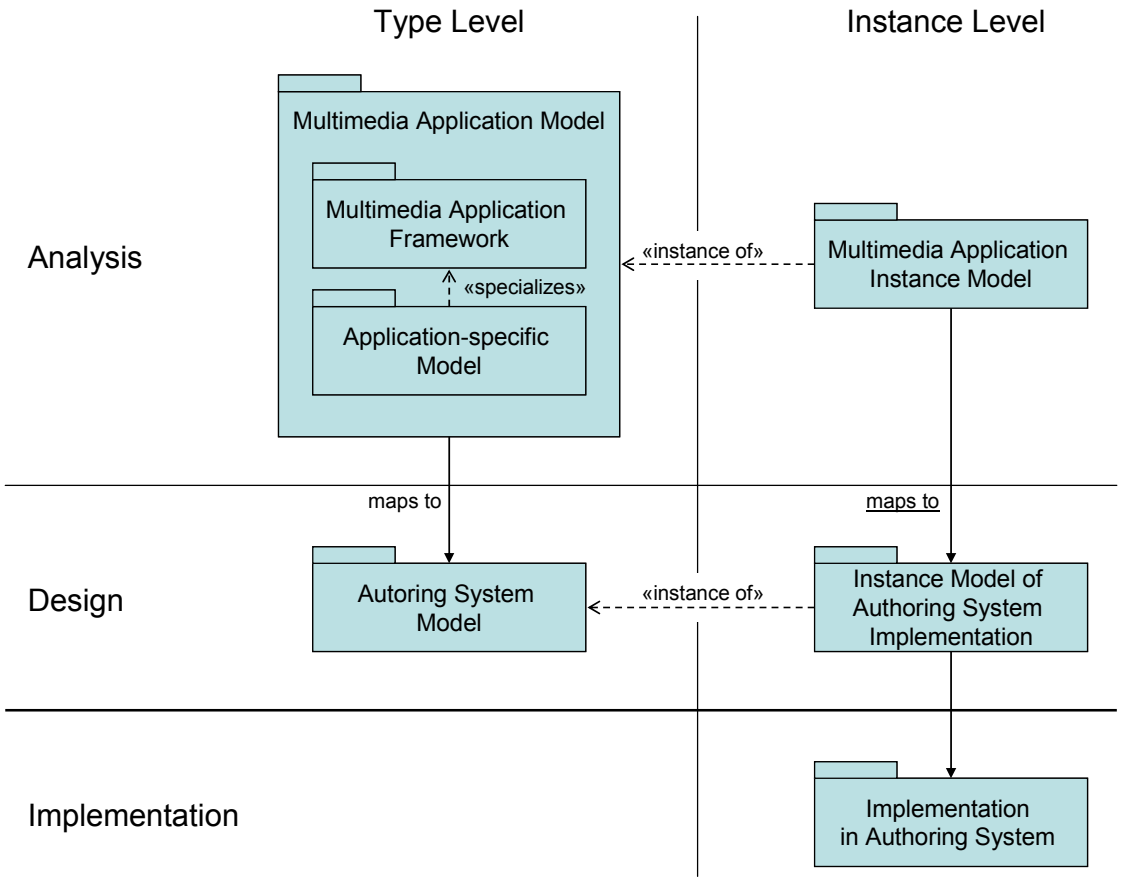


Figure 14: Model architecture for the model-based multimedia software development process with multimedia authoring systems (adapted from [DEM+99])

In our approach, the analysis model of the multimedia application is based on a framework that conforms to the typical architecture of multimedia applications. The framework consists of four dimensions: application logic, presentation, control, and media. Application-specific classes are specialized from the framework classes. Framework and application-specific classes together build the platform-independent application model of the analysis phase that is independent of the authoring system and

implementation technology. In addition, we employ a capability model of the selected authoring system on the design stage. It models the implementation concepts that are supported by the tool. We define a transformation mapping between the application's analysis model and the authoring system model that is part of the design model. This mapping enables the transformation of instance models on the analysis level to instance models on the design level.

From these two models, we systematically derive an implementation model on instance level which is used as input for the authoring system. Basic functionality can be visually programmed using the direct-manipulative graphical user interface of the authoring system. More complex functionality can be coded in the scripting or programming language of the authoring system. Code generation techniques and tools may also be applied to (partially) automate the transformation from models to code, making this development method not only model-based, but model-driven.

In [DEM+99], we have deployed this process model for the application and system domain of multimedia e-learning using the authoring system Director. With our process model, we improved the development process of multimedia applications according to software engineering criteria.

Altogether, the process model and development method support the model-based development of multimedia applications which are eventually implemented with a customary multimedia authoring system. The central idea is to use a programming model of the authoring system (i.e., a platform model) on the design stage to bridge between the platform-independent analysis model of the application and its implementation with the authoring system. The process model and method are not bound to a particular authoring system, but can be transferred and reused by defining a new authoring system model and a new transformation mapping from the application model to the authoring system model.

Since temporal behavior is not sufficiently represented in the model of [DEM+99], we extended the UML for the modeling of timed and dynamic behavior of multimedia applications in the modeling language OMMMA-L (see Section 5.2).

5.4 Integrated Methods for Interactive Multimedia Systems

In an attempt to link the work on multimedia software engineering methods and processes with design methods for interactive systems, we have commenced research on integrated software engineering methods for interactive multimedia systems. The development of many modern software systems requires joint work of experts from different domains due to the systems' complexity and the demand for a wide variety of qualities that need to be achieved. This does not only apply to technical systems like embedded and mechatronic systems – where mechanical and electrical engineering knowledge and engineering methods need to be combined with software engineering methods and expertise – but also for the system domains of interactive software systems in general and interactive multimedia systems in particular. For this purpose, we have worked on the integration of development methods and techniques from different areas of expertise.

The integration of software engineering methods and methods of user-centered design for the development of interactive multimedia systems can be assigned to the domain of multimedia software engineering. In cooperation with usability engineering experts, we have identified the characteristic similarities and differences between the OMMMA method (see Section 5.2) and user-centered design methods and techniques. Based on this analysis, we have conceptualized the integration of these approaches in a comprehensive development process for interactive multimedia applications [ESN03].

The object-oriented modeling of interactive multimedia applications in the OMMMA approach is designed to enable multimedia software developers to create comprehensive analysis and design models of multimedia software. For development of highly usable multimedia applications, this approach must be embedded in a comprehensive development process that takes a user-oriented perspective on multimedia software development. In [ESN03], we elaborate on the differences between user-centered design activities and object-oriented software design activities and outline their integration in a comprehensive development process. A high-level view of this process integration is given in Figure 15.

The two domains mainly differ in their perspective on the system under development. Furthermore, the user-centered methods are mostly targeted to the early phases of a development process where an involvement of end-users is of great importance. We argue that an integration of a user-centered design approach with an object-oriented software design approach is an important step for the development of interactive multimedia systems which are accepted by end-users. On this basis, an investigation of conceptual and design models within a user-centered approach and their possible transformation into software design models is outlined.

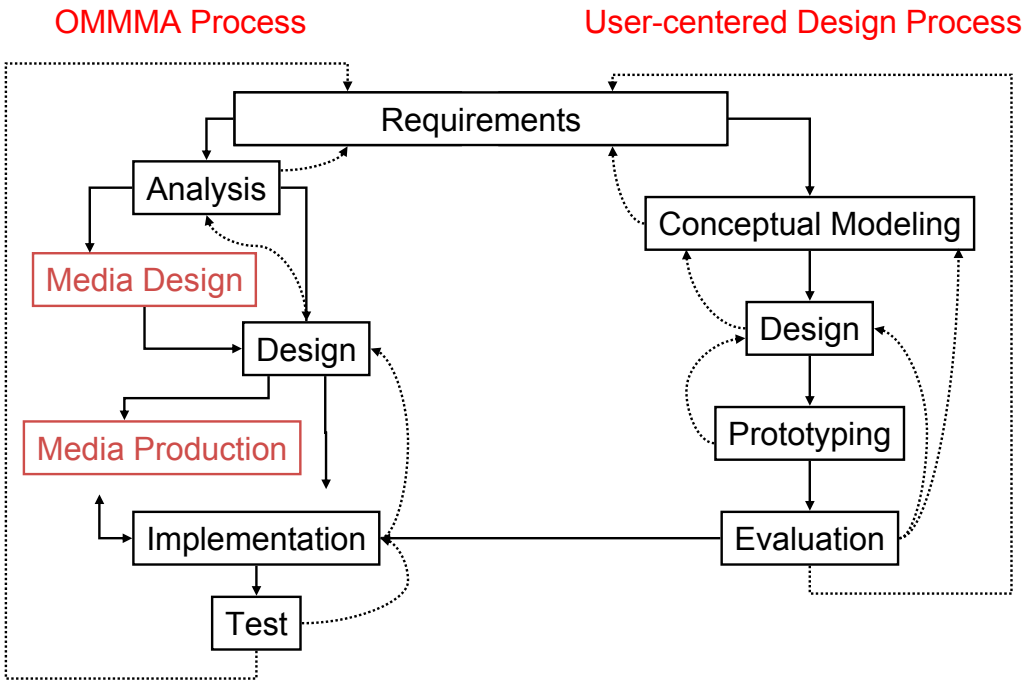


Figure 15: High-level schematic view of process integration of OMMMA process and user-centered design process

5.5 Model-driven Development of Interactive Multimedia Systems

Another continuation of the work on modeling multimedia applications goes in the direction of rapid, model-driven development and simulation of multimedia user interface prototypes. For this purpose, parts of the OMMMA approach have been taken up and extended for the development of the GuiBuilder method and tool **[SE07]**, [SDGH06]. Classic GUI builder tools are widely used in practice for developing the user interface of software systems. Typically they are visual programming tools that support direct-manipulative assembling of the user interface components.

GuiBuilder integrates the model-driven development paradigm with the GUI builder tool concept. It facilitates model-driven development of graphical and multimedia user interfaces [SDGH06]. User interface developers model the structure and appearance of the user interface with object-oriented presentation diagrams and its behavior with hierarchical statecharts.

The GuiBuilder tool provides integrated editors for the compositional presentation diagrams and the hierarchical statecharts. GuiBuilder then supports the transformation of the model to Java. Working user interface prototypes are generated. When they are executed, the modeled behavior is simulated. The execution can be controlled dynamically by users or scripts. Interactive sessions with the user interface can be recorded (as scripts) and replayed. The GuiBuilder tool combines the modeling and execution environment. It has been implemented using the Eclipse platform. The general architecture of the GuiBuilder tool is shown in Figure 16.

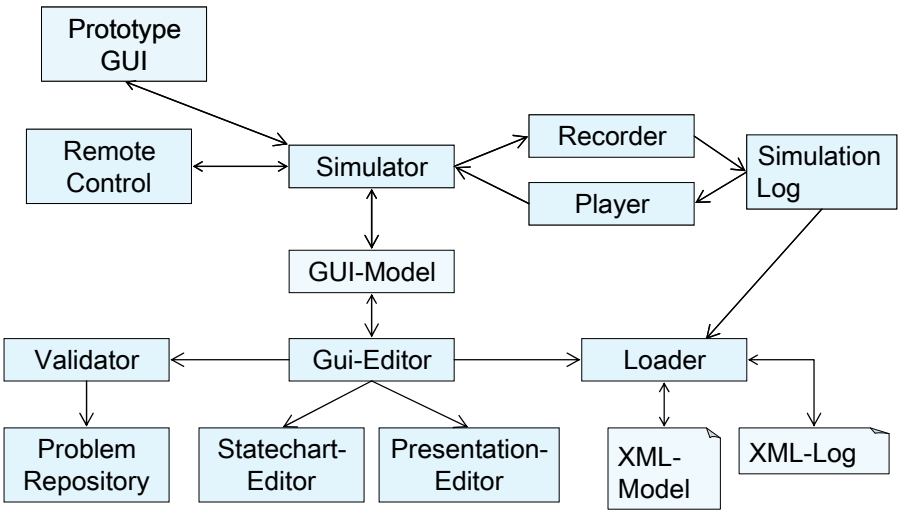


Figure 16: The general architecture showing the components and their communication relationships of the GuiBuilder tool

As to open software development towards end-user development, the GuiBuilder method and tool enable a participatory design approach where user interface developers as well as prospective end-users of the system can contribute to the design process. They are supported in modeling the desired functionality of the GUI on a high level of abstraction that is easy to understand for all involved stakeholders. The integration of the model-driven development paradigm with the GUI-builder tool concept provides

them with a usable tool for prototyping graphical (multimedia) user interfaces in practice.

We have evaluated GuiBuilder in several workshops with high-school students and people who are interested in software development, but not professional software developers or programmers **[SE07]**. After a presentation of the tool of about half an hour they were capable of using the tool for constructing, changing and simulating simple applications like a traffic-light control with only very limited support by our tutors. Thus, the tool has shown its capability to support end users with little programming skills in building and simulating interactive graphical user interfaces.

Model-driven development of graphical and multimedia user interfaces belongs to the domain of model-driven development of advanced user interfaces (MDDAUI). MDDAUI has been the topic of a series of workshops in the last years [VMB+10], [VMS10], [VSB+10], [MGB+09a], [MGB+09b], [PVS+08], [PVH+07], [PVS+07], [PVSH06], [PVS+06], [PVHS05]. MDDAUI applies generative software engineering methods to the development of advanced user interfaces, just like GuiBuilder.

GuiBuilder has another property that is worth mentioning in this context: It supports the cooperative development of design and functionality. This combination happens on a low level of abstraction, the platform-specific design model of the user interface. But it nevertheless points towards integrated software engineering methods specifically for advanced interactive systems.

As we have seen for the model-based development of multimedia applications in Section 5.4, software engineering methods will eventually have to be integrated with informal, user-oriented design techniques in a holistic development method for such user interfaces in order to appropriately account for usability requirements. Recently, we have applied our meta-method for method engineering for designing model-driven development methods for advanced user interfaces **[Sau11]**, see Section 7. There we particularly address the integration of software engineering methods with less formal user-interface design methods.

5.6 Generation of Web Application Prototypes

Related to the work on GuiBuilder, although not for multimedia applications, is the research conducted on model-driven development of Web application prototypes ProGUM-Web. It is in so far similar to GuiBuilder as it accounts for the cooperative development of Web applications by programmers and Web designers. ProGUM-Web offers a method and a tool for the generation of prototypes of dynamic Web sites from UML models [LSS03]. The development process consists of three stages which are executed iteratively and incrementally: modeling, coding, prototype generation. In the modeling stage, we use an extension of the UML that covers specific characteristics of Web applications and their development process. From the models, we generate dedicated code templates for software developers and graphics designers. The code templates can iteratively and independently be edited by them and are then re-integrated within the ProGUM-Web tool. In the third stage, the tool automatically generates an executable prototype of the Web application from the integrated files. The generation facility can be used throughout the development cycle.

With ProGUM-Web, dynamic Web sites can be independently developed by graphic designers and software developers. This functionality is based on the generation of role-specific code from UML-based models. Developers can check-in the respective code modules they worked on, and changes to a prototype can be fed back into the repository. Key to this cooperative model is the architectural separation of functionality and design.

After coming from multimedia applications and user interfaces of interactive systems to Web applications, we next look at a very different class of software systems: large business information systems.

6 Business Information Systems

Research in the area of software engineering methods for business information systems (BIS) will be presented in this section. The specification method that I have developed together with a large company (Section 6.1) is the most important contribution with respect to method engineering, since its development followed a serious method engineering method and process. This also applies to the work on the integrated method for application landscape and application development (Section 6.2) and the integrated specification framework (Section 6.3) that accompany the specification method. Other work on the integration of engineering methods and quality assurance that is presented concerns requirements engineering and software test (Section 6.4.1) and quality assurance in agile methods like SCRUM (Section 6.4.2). Finally, architecture-driven development with open-source stacks is presented as one method for efficient development of large business information systems (Section 6.5). Figure 17 summarizes these approaches.

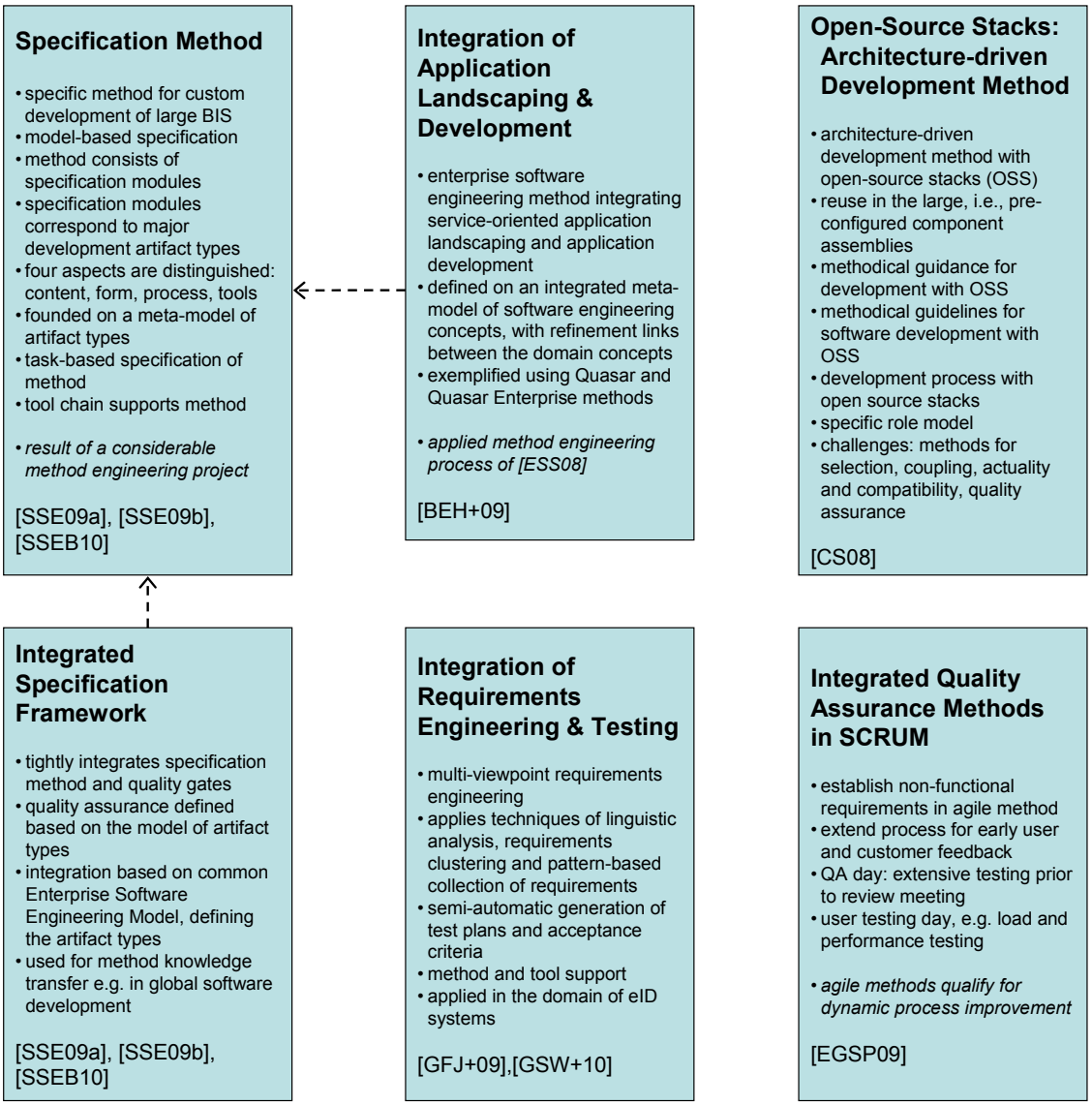


Figure 17: Research contributed to software engineering methods in the business information systems (BIS) domain

6.1 Specification Method for Business Information Systems

In a cooperative effort with the research department and multiple business units of a software company, I have development a specification method for business information systems (BIS); see e.g. [SSE09a], [SSE09b], [SSEB10].

The objective of the specification method has been defined as follows:

> *The Specification Method is the sum of systematic processes and utilities for the preparation of precise, unambiguous functional system specifications which we refer to as system specifications. Adapted to the context of the respective project, it defines the concepts and artifacts of the specification, offers specification languages and resources such as templates and examples, provides instructions regarding the process within the specification, establishes result types (work product types) for the specification and supplies suitable supporting tools. It structures the specification into disciplines, activities and tasks, and establishes the relationships between various parts of the specification.*

The specification method is particularly designed as a unified method for custom-development projects of large-scale BIS. A company-standard tool support is provided together with predefined specification templates. Method and templates can be tailored to support specific needs of the project.

The specification method is concerned with describing a software system from a conceptual viewpoint. It is assigned to the discipline *analysis* of the development process. It distinguishes the content from its form of representation and from the process how the artifacts of the software specification are produced. It also covers the support of CASE tools. These four aspects are defined on a common basis: the ontological method engineering meta-model of the relevant concepts and their relationships. While the terminology used in that specification method is slightly different from the one in this work, the four aspects resemble the notions of concept, notation (as part of artifact type), method, and tool of the method engineering framework in this thesis.

The four fundamental aspects of the specification method are further characterized as follows:

Content of a specification: Method engineers define what content a specification needs independently of the form. The content defines the parts of a system specification, according to the given business goals and requirements. Typically, the system specification includes a description of the information system's conceptual model, use cases and dialogs, and other contents. The content structure is based on a meta-model of all elements (i.e., the conceptual artifact model of the analysis discipline), including their properties and relationships to each other.

Form of a specification: The form of a specification defines how the (previously defined) content of a system specification is structured and, using a concrete notation, how it would be represented best. For example: Having defined the use case as a basic artifact, the form specifies how a use case will be specified with the UML and/or natural language.

Process for creating a specification: This aspect describes the process of creating a system specification. The steps from the problem to the solution idea to the conceptual design of an application make up the core of the process during specification. The specification method covers content production, quality assurance, and specification management. It also describes the relationships to activities of other disciplines of the company's overall software engineering methodology where adequate, especially the design. Details about the procedure for creating the separate parts of the specification are given in the corresponding specification modules.

Tools which support the specification: The specification has to be done with some kind of tool; ideally with the method-conformant use of a specification tool in full extent, using it as the specification information repository. Although the specification method is independent of a particular tool, Sparx Systems' Enterprise Architect is proposed as the basic specification tool. This aspect thus explains the collaboration between the tool and the specification method.

The separation of these four aspects creates the necessary flexibility to maneuver in the creation process and to tailor the method in the unavoidable trade-off between standardization and individuality: On the one hand, the method should be deployable across the entire company and make system specification safer, more efficient and more uniform. On the other hand, projects that develop custom software systems may significantly differ in their contextual requirements, and so they have different requirements for a specification. The specification method nevertheless assumes that the necessary content forms the predominantly "stable core" of the method; form and process are customized to a greater extent to the project in question, yet it is recommended to use the same form and process as in the method's specification. The structure of the specification method and the four aspects will now be briefly explained.

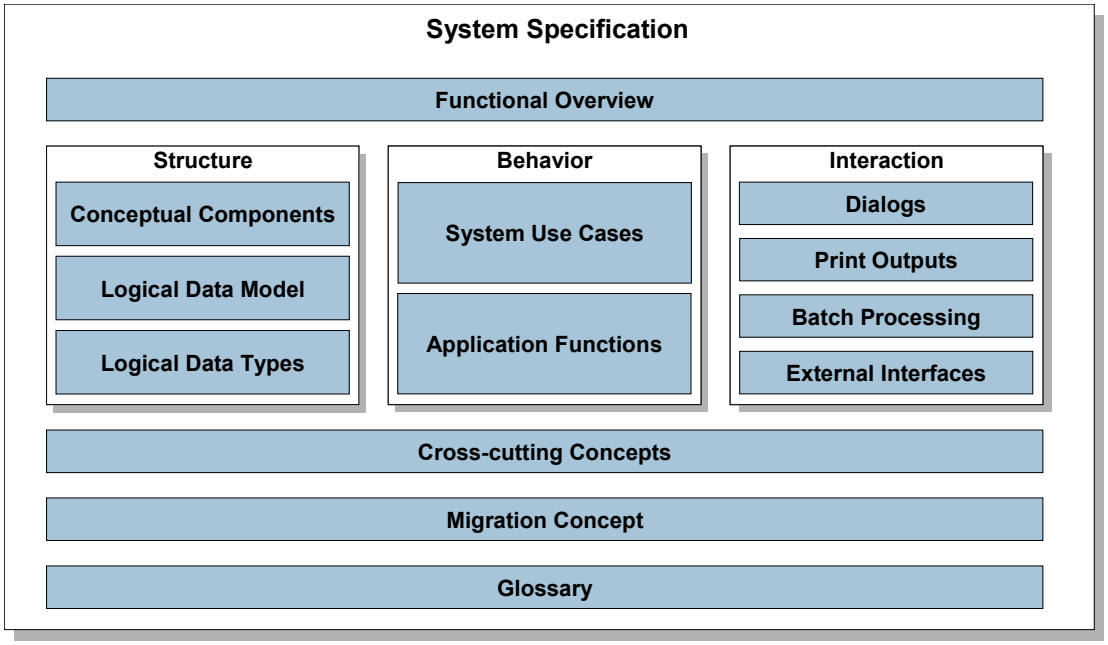


Figure 18: The specification method is organized in specification modules that correspond to the main artifact types of the system specification

Specification modules. The specification method is organized in method modules, so-called *specification modules* (see Figure 18) .Specification modules contain the detailed information regarding content and resources for producing system specifications of business information systems. They complement the general part (Part 1) of the specification method which defines the methodical superstructure and the specification process within the discipline *analysis*. Specification modules have a largely uniform structure. Within the specification modules, content and form, i.e., structure and notations, of the related artifact types are described. The description of the process and method is task-oriented. Methodical guidelines (i.e., "good practices"), specification techniques, examples, available tool support, and recommendations and warnings are also included.

The specification modules correspond to the main artifact types of a system specification. The functional overview provides a high-level specification of the BIS under development. Conceptual components are the fundamental concept of structuring the software specification and grouping its artifacts. They are derived from the topics of the problem domain. Behavior is derived from business processes (part of the *business modeling* discipline, another discipline of the overall software engineering methodology) and specified on two levels: use cases and application functions. While system functionality is generally specified with use cases, application functions refine system actions of use case specifications, e.g. if they provide complex algorithms or they are used in multiple use cases. Interaction of the software system is specified in terms of dialogues with users, interfaces with external systems, print outputs and batch processing.

Cross-cutting concepts, such as authorization or logging, migration concept and glossary complement the software specification modules in the groups structure, behavior and interaction, and the functional overview.

The general methodical principles (Part 1 of the specification method's documentation) and the specification modules (Part 2) are mapped into UML profiles which are implemented in the CASE tool (currently Enterprise Architect) to provide the required tool support.

Process. As the specification method is embedded in the company's overall software engineering methodology, it expects artifacts from the two other disciplines *business modeling* and *requirements engineering* as required input. Conversely, it is responsible for producing those artifacts that are required for the *design*, *implementation*, and *test* disciplines.

The discipline *analysis* comprises three general sub-disciplines that group different tasks: content production, specification management, and quality assurance (see Figure 19). Orthogonally, the process of this discipline is divided into three phases: *initialization*, *coarse-grained analysis* and *detailed analysis*.

Content production is the actual engineering discipline and concerned with creating and clarifying the content of the system specification. Within the tasks of this sub-discipline, the artifacts of the system specification are actually developed. *Specification management* is responsible for planning, monitoring, and controlling the course of specification. Its tasks are dedicated to organizational issues. The objective of the

q*uality assurance* sub-discipline is checking and assuring the required quality of the system specification, both the product quality of the specification artifacts and documents as well as the process quality of their development. Its tasks comprise checking plans, reviewing deliverables, and conducting audits.

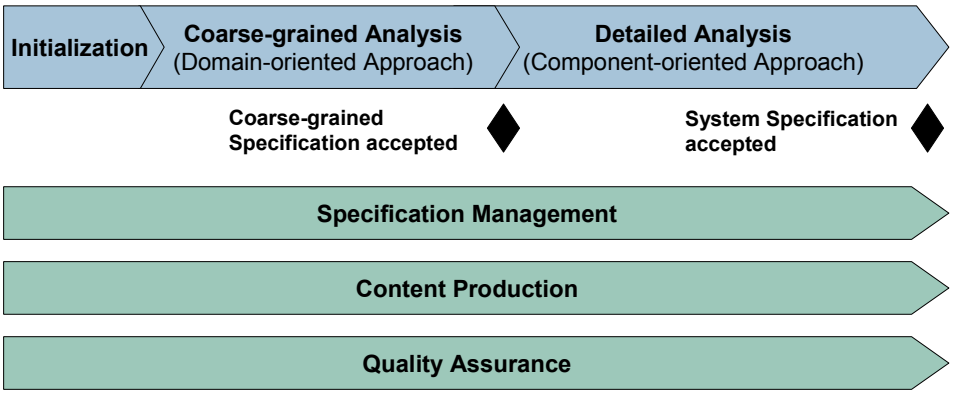


Figure 19: The discipline *analysis* comprises three phases and three sub-disciplines, and defines two milestones

A system specification is produced in two important steps, corresponding to the two phases *coarse-grained analysis* and *detailed analysis* with their respective milestones in Figure 19:

- First, an overview of the system and its functionality is developed. The most important product of this phase is the *functional overview*. In addition, artifacts that are crucial for further planning are already specified to more detail in this phase. In this phase, the specification process is organized according to the business domains (domain-oriented approach).
- Secondly, a detailed specification is made, producing the individual artifacts of the *system specification* according to the specification modules, thus building the complete specification of the system. The specification process is organized according to the previously identified conceptual components (component-oriented approach).

Including the *initialization*, the tasks of the discipline *analysis* are thus executed according to three basic phases. We have defined two milestones accordingly: (1) “coarse-grained specification accepted” and (2) “system specification accepted”.

As has been stated above, the specification method is task-oriented. Figure 20 shows the tasks and subtasks of discipline *analysis* with respect to the basic procedure in the three phases and also reflects the two-way split of producing the system specification. The description of the tasks is directly combined with the characterization of the work products, i.e., the artifacts that are produced when executing the tasks.

Content and form. The artifact types combine the relevant specification concepts with their recommended form of representation in the system specification, as defined by the specification method.

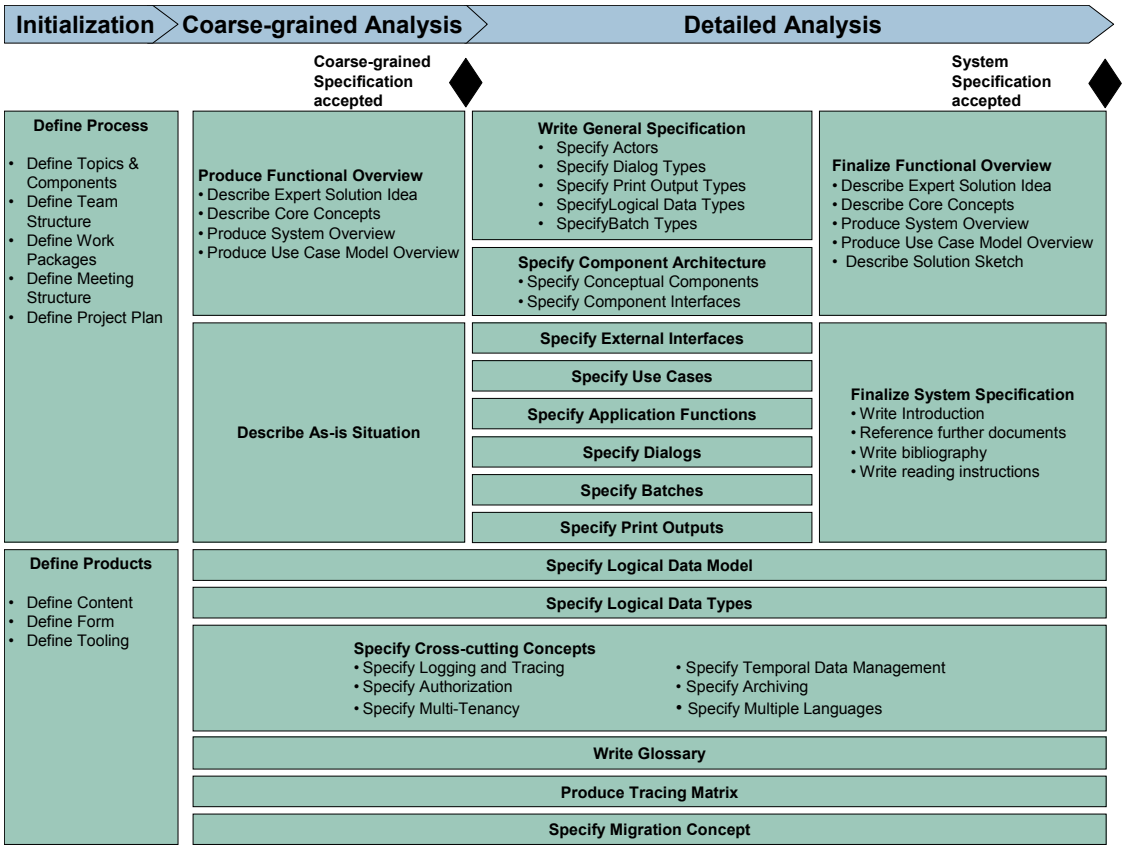


Figure 20: Tasks of discipline a*nalysis* with respect to the three phases

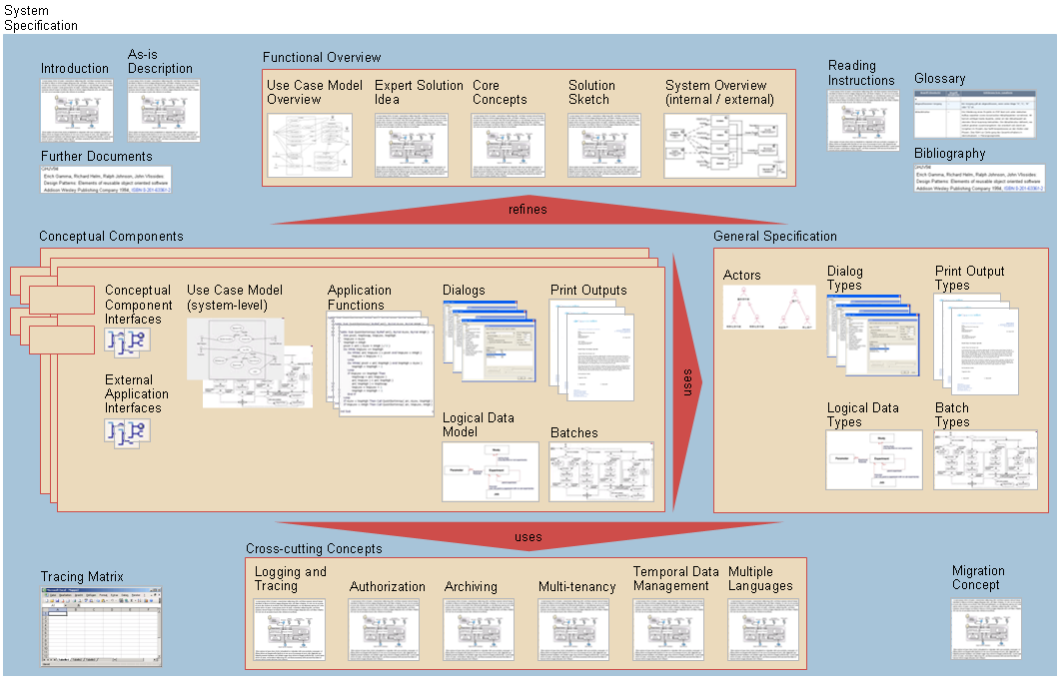


Figure 21: Artifacts of the *system specification* in bird's eye view

Figure 21 gives a schematic overview of the artifacts that belong to the *system specification*. It shows that the conceptual components are the central concept in the system specification. They refine the functional overview and use the general

specification and the cross-cutting concepts. The concepts in the upper left corner induct the reader into the system specification and the concepts in the upper right corner support reading the system specification. The tracing matrix in the lower left corner gives an overview of the relevant relationships in the system specification and to the *requirements specification*. The migration concept in the lower right corner holds the relevant information concerning the migration aspects.

The specification method is defined with the meta-model of system specification artifact types as its backbone. The meta-model builds also the foundation for quality assurance by so-called Quality Gates (see Section 6.3) as well as the company's standard software testing method. Of course, also the tools that support the specification method conform to the model of artifact types.

Tools. A standard set of tools goes along with the specification method. Figure 22 shows all parts of the tool set-up (compare [SSEB10]). The provided setting fits the underlying meta-model of specification artifacts and furthermore leads to a very efficient ramp-up of the teams.

(1) First, we define UML profiles that contain the stereotypes according to the artifact types in the meta-model.

(2) We use the MDG technology of Enterprise Architect for defining custom diagram types with connected toolboxes. In the toolboxes only those stereotypes are listed that shall be used to specify an artifact.

(3) The next step is to actually specify the network of interrelated artifacts. For this task, the user applies the process, practices, and methods described in the specification modules in connection with a specific concept of use for the Enterprise Architect.

(4) The specification can be checked for its conformance with the meta-model and its internal consistency with the Specification Validator tool. It is available as an Enterprise Architect plug-in that supports the editing and management of a rule base as well as running and reporting the configured checks on the meta-model instance in the Enterprise Architect repository.

All the previous steps are important for building a good system specification as regards its content artifacts. The next steps present another, yet very important aspect: the production of documents, since customers and partners often do not work with the modeling tool.

(5) Again, for efficient ramp-up, we provide adaptable templates. Specialized templates may be selected for specific purposes, such as e.g. global software development.

(6) In the Document Generator, the user may select and deselect artifacts in the Enterprise Architect repository. For example, it is possible to select only some of the components if the document is for a specific specialist division. We thus support the generation of deliverables specific for different target groups such as the customer's domain experts, IT experts, or our software design teams.

(7) Finally, the document generator produces word documents based on the templates, the repository and the selection.

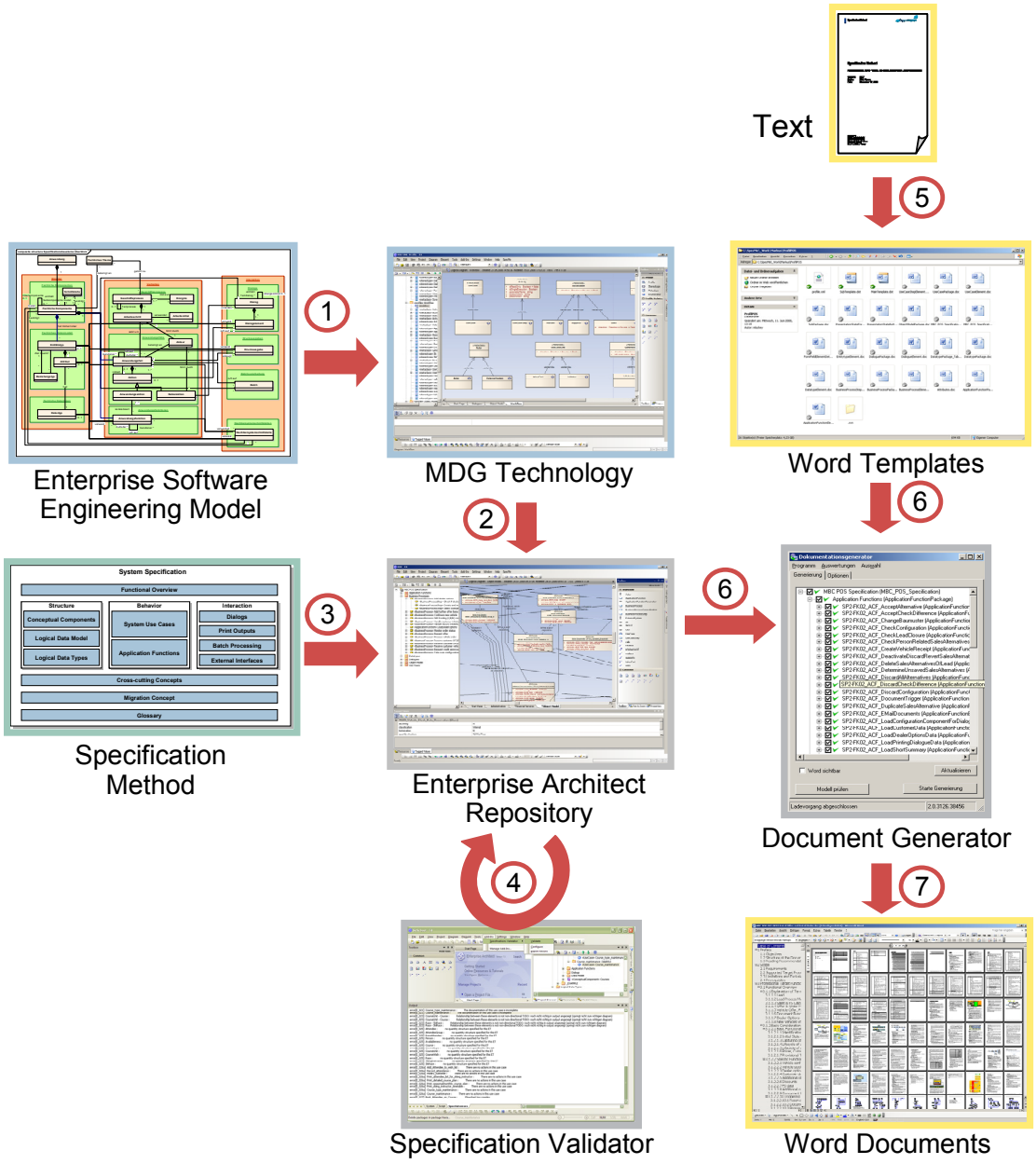


Figure 22: Tool support for the specification method for business information systems [adapted from SSEB10]

Method engineering. The specification method has been developed in a dedicated *method engineering project* of considerable size. Therefore, applying a controlled method engineering process ensured the quality of the developed specification method as a "tool" for model-based software development of large business information systems. The method engineering process that we defined with the company is depicted in Figure 23. It distinguishes three sub-processes: method development (top row), continuous quality assurance and improvement (second row), and accompanying activities for roll-out and dissemination. (To improve readability, loops for revisions

and the method improvement cycle have been omitted.) The process activities can be further characterized as follows:

- **Requirements**: elicit and specify the requirements for the specification method
- **Analysis**: analyze the state-of-the-art in theory and practice and extract good practices
- **Construction**: create the specification method's artifacts (documents, templates, tools, examples, etc.)
- **Pilot**: test the specification method in pilot projects
- **Productive Use**: use the specification method in projects throughout the company
- **Quality Assurance**: continuously assure quality during construction by expert reviews
- **Evaluation & Evolution**: continuously evaluate and improve specification method during use
- **Rollout & Training**: rollout the method in the company and train the software engineers
- **Coaching**: coach project teams that apply the specification method in their software development projects

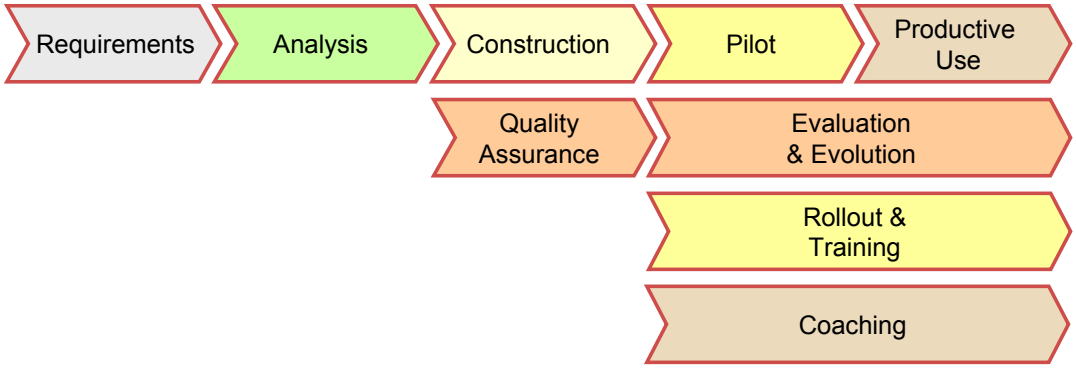


Figure 23: Method engineering process for the specification method

6.2 Integration of Application Development and Landscaping

In [BEH+09] we follow a method integration approach and present a holistic software engineering method for service-oriented application landscape development. It combines application landscaping and application development based on a common meta-model of software engineering artifacts.

We outline that the development and management of an application landscape as part of a service-oriented enterprise architecture requires a holistic approach, which reaches from business modeling and global application landscaping down to the local development of individual software components and services. We illustrate how such a holistic software engineering method for enterprises can be systematically composed and integrated from existing methods. Based on our previous research results in the area of method engineering [ESS08], we show how two concrete existing methods can be integrated. In particular, we use a common ontology of software engineering concepts, where refinement links interrelate the concepts of both methods. By this, a smooth transition between both methods can be defined.

The approach is exemplified in [BEH+09] with two existing methods, which are developed and used within the software company Capgemini. Those are Quasar Enterprise for developing application landscapes and Quasar (Quality Software Architecture) for developing single applications. Both methods contain a set of disciplines for their respective development goals. They are depicted in grey shaded boxes in Figure 24. To combine both methods, the common Quasar Ontology defines the integrated meta-model of software engineering concepts. Application landscaping and software development can then be aligned based on dependencies between the meta-model classes from the respective domains application and application landscape.

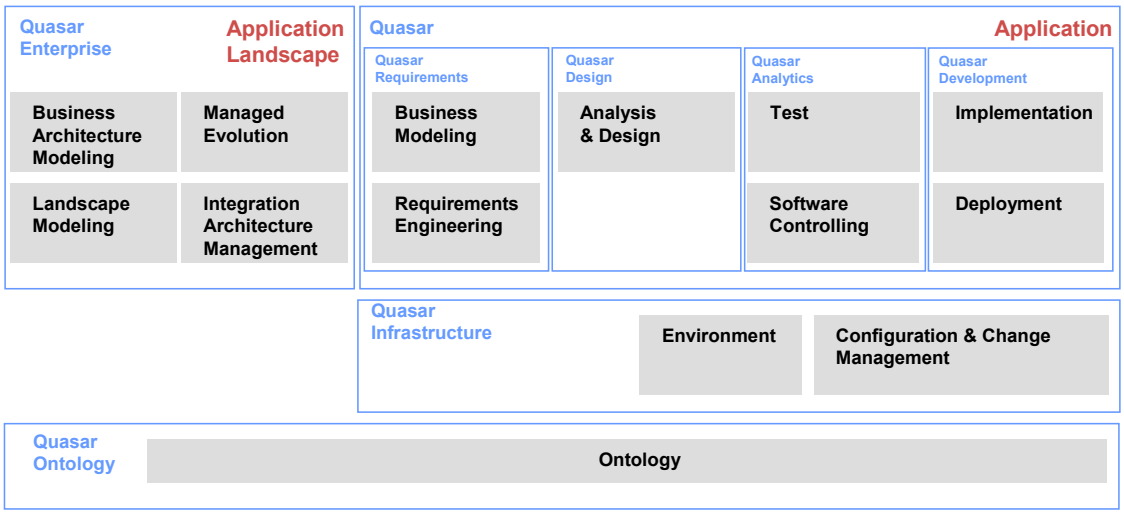


Figure 24: Disciplines of the Quasar and Quasar Enterprise methods (adapted from [BEH+09])

Both methods are integrated within a holistic software engineering method to seamlessly cover the full development cycle of service-oriented application landscapes, from business modeling and service design to actual software development. Figure 25 demonstrates how the integrated ontology serves as the key feature for integrating both methods. The integration is expressed by the 'refines' edges between the meta-model classes of Quasar (right) and Quasar Enterprise (left). The example shows the relationship between Quasar Enterprise's method for decomposing Business Services into Elementary Business Services (which are realized by Logical AL Components via Application Services), with Quasar's method for decomposing Business Processes down to Use Cases which are realized by Logical Application Components (in gestalt of Logical Operations that relate to Use Case Actions). Comparing these two sides and interrelating the notions of both sides enable a smooth transition from global application landscaping down to local application development. As an effect, Business Service Activities can be interpreted as the external view of an Activity (i.e., Business Process or BusinessTask). More details can be found in [BEH+09].

6.3 Integrated Specification Framework: Method and Quality Gates

The industrial-strength specification method that we presented in Section 6.1 is part of an overall system specification framework. This *specification framework* is introduced in [SSE09a] and presented in more detail in [SSE09b]. It consists of the specification method for business information systems and the *specification quality gate*, named QG

Specification. Actually, QG Specification assesses the quality of the then not yet ready *system specification* in order to detect erroneous developments early and have them to be corrected in a timely fashion. Another quality gate, QG Architecture, is later also involved in the quality assurance process of the system specification. It checks, among other artifacts, the completed system specification, but with fewer rigors and less methods. Both parts, engineering and quality assurance, are tightly integrated. Since this framework is tailored to the specification of large business information systems, it also facilitates a quick ramp-up phase for software engineering projects without the need for extensive tailoring or extension.

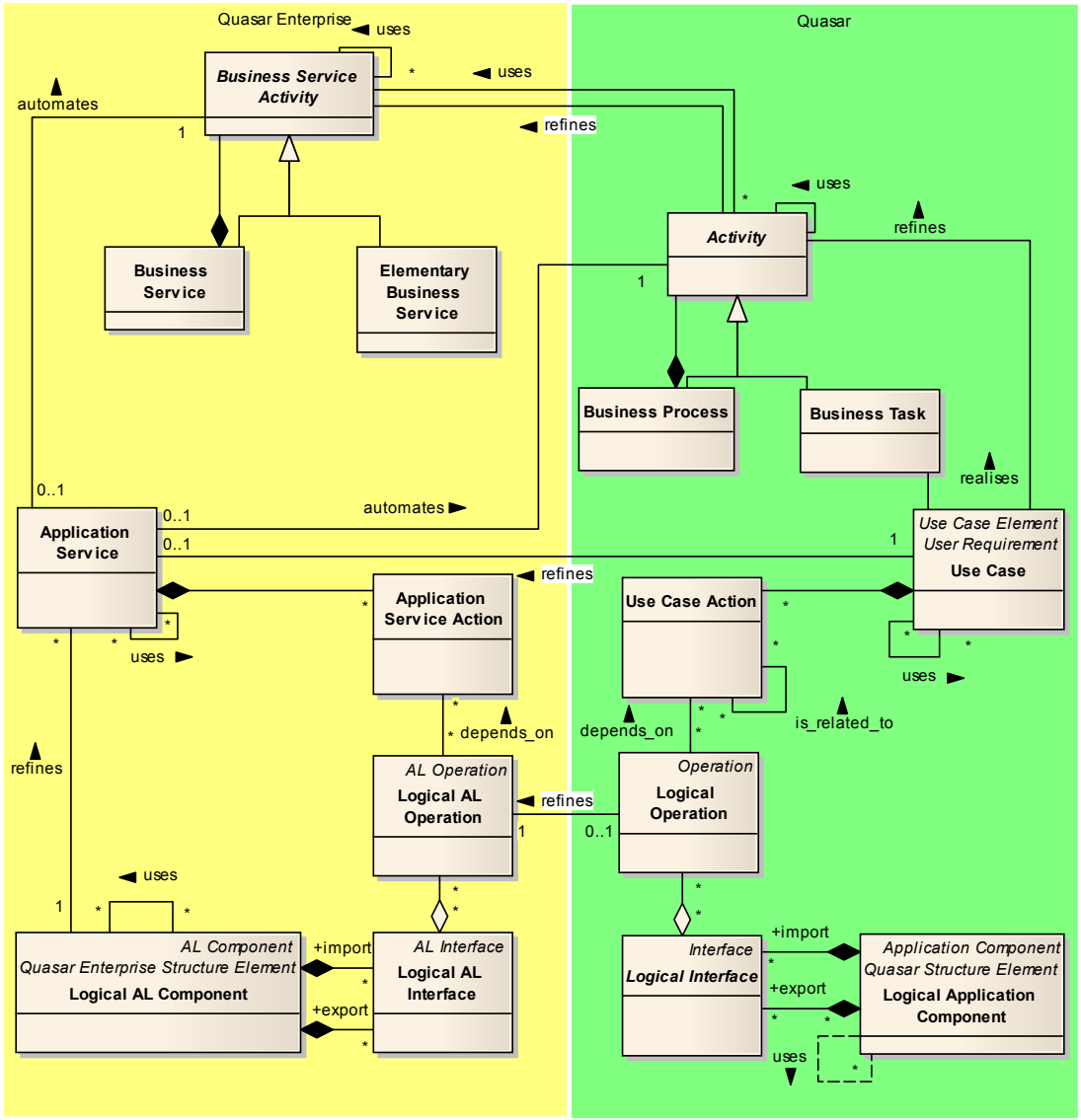


Figure 25: Excerpt from the integrated Quasar Ontology (from [BEH+09])

The *software specification* acts as a bridge between customers, architects, software developers and testers of business information systems. The *specification method* consolidates the company's engineering knowledge and best practices, enhanced by the practical application of recent developments in theory and practice, on how to specify business information systems. It serves as the constructive basis for creating complex system specifications. The *specification quality gate* ensures that the software

specification has been produced in high quality. It acts as the specification method's concerted analytical counterpart for software quality assurance on the level of specification artifacts. Quality gate assessments are executed by auditors in order to evaluate the maturity and quality of the produced software artifacts. The quality gate uses precisely defined check methods together with checklists, templates and scenarios (e.g., user scenarios, change scenarios) for assessing not only the produced artifacts, but also the actual and planned process.

The *specification framework* is the result of an effort to standardize system specification throughout the company. In [SSEB10], we report on re-aligning and unifying the varied software engineering methods that existed before within the software company, and on the standardization of quality assurance procedures. By this activity, we arrived at a comprehensive company-wide *Enterprise Software Engineering Model* (aka. "Quasar Ontology") that effectively build a common body of methodical knowledge and supports knowledge transfer between stakeholders and teams. Furthermore, it is the foundation for rigorous method integration.

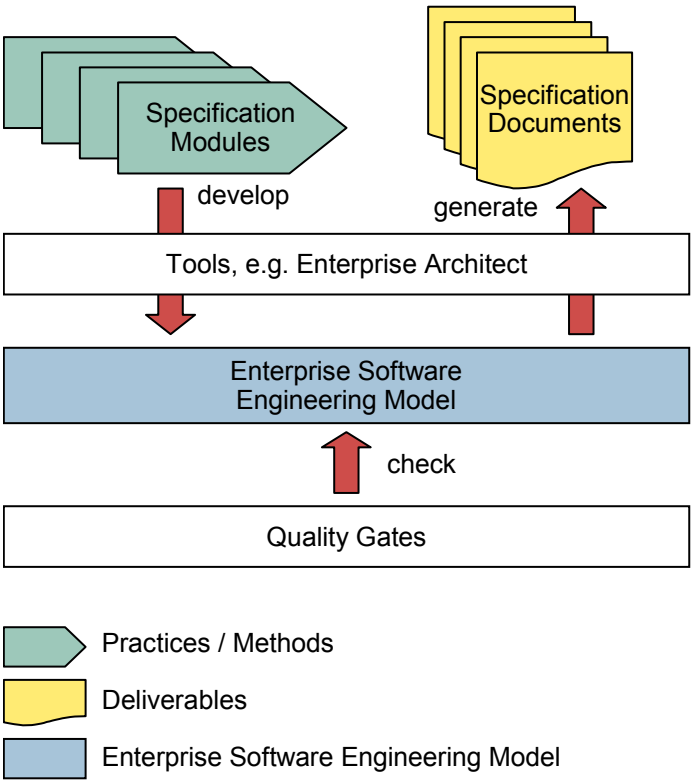


Figure 26: A schematic overview of the integrated framework for system specifications and their quality assurance based on the Enterprise Software Engineering Model (ESEM)

The Enterprise Software Engineering Model (ESEM) is the central part of the integrated methodology, see Figure 26. It defines the unified terminology, the artifact types and the relationships between them. The software development methods operate on the ESEM, creating, using and modifying artifacts, which embody instances of the ESEM, by the use of tools. Tools are also used to generate the deliverables as views on the project's instance of the ESEM. These views have to consider the background of the particular target audience and the current state of the developed artifacts. Views on the specification artifacts are e.g. customer-specific or developer-specific views.

The specification method is part of a holistic set of enterprise software engineering methods that cover all disciplines of enterprise application landscaping and application development ([BEH+09], see previous section), based on the ESEM. The *specification method* already uses the ESEM as its backbone, i.e., the specification method's artifact meta-model has become part of the ESEM. Artifacts of the system specification are thus produced by instantiating artifact types of the ESEM. To unambiguously determine the deliverables and the maturity of the deliverables, they are also defined with respect to the ESEM. *Quality gates* are then used as defined milestones. They assure a balanced growth of the developed artefacts. Quality gate assessment complies to the ESEM as well. We apply them to check the quality of the produced artifact.

With the integrated methodology, constructive and analytical methods stand no longer isolated next to each other, but are directly coupled based on the ESEM, supporting the quality assessment of the developed artifacts.

The integrated specification framework has been utilized as a methodical means for knowledge transfer in global software development (GSD) to share development knowledge among the onshore and offshore stakeholders and developers, and to assure the quality of the exchanged information [SSE10]. Availability of a unified and integrated set of methods fosters the exchange of knowledge (and the migration of people between projects) and the dependability of the methods for offshore development teams. They do no longer have to repeatedly re-adjust to method variants used by the different business units, avoiding misinterpretation of information and risks for project success.

6.4 Integration of Software Engineering and Software Quality Assurance Methods

The integration of software engineering and software quality assurance methods is attracting increasing interest recently. This challenge is addressed by our research on software testing and the tight integration of constructive software engineering methods and analytical software quality assurance methods. In addition to the integrated specification framework that has been presented in the previous section, we have investigated the integration of requirements engineering and model-based testing and the integration of software quality assurance methods in agile development methods.

6.4.1 Bridging Requirements Specification and Test

The linkage between requirements engineering and system specification with acceptance testing is the topic of [GFJ+09] and [GSW+10]. In acceptance testing, customer requirements as specified in system specifications have to be tested for their successful implementation. This is a time-consuming task due to inherent system complexity and thus a large number of requirements.

In order to reduce effort in acceptance testing, we exploit redundancies and implicit relations in requirements specifications. We use requirements specifications that are based on the multi-viewpoint technique of the reference model for open distributed processing (RM-ODP). Specifying with the RM-ODP model inherently yields redundant requirements. This redundancy supports the readers of the requirements documents in their understanding; but it may cause unnecessary effort in acceptance

testing where the requirements are checked for their fulfillment by the developed system. For that reason we look for a way how the redundancies and dependencies between requirements can be identified systematically. Once we have identified the redundancies and dependencies, we aim at reducing the testing efforts for acceptance testing.

As a solution, we deploy linguistic analysis and requirements clustering techniques [GFJ+09] as well as pattern-based requirements collection [GSW+10] for reducing the total number of test cases that are derived from the requirements specification. In particular, we provide capabilities for automatically deriving semi-formal test plans and acceptance criteria from the clustered informal textual requirements.

We apply requirements clustering techniques for identifying redundancies between requirements. First, we annotate requirements with semantic attributes. They allow us to differentiate between requirements types and enable the computation of the similarity between each pair of requirements. By defining a threshold value we can group related requirements into fine-grained clusters. For each cluster, efficient test plans can be designed by ordering related requirements. In [GSW+10], capabilities for automatically deriving semi-formal test plans and acceptance criteria from the clustered informal textual requirements are added.

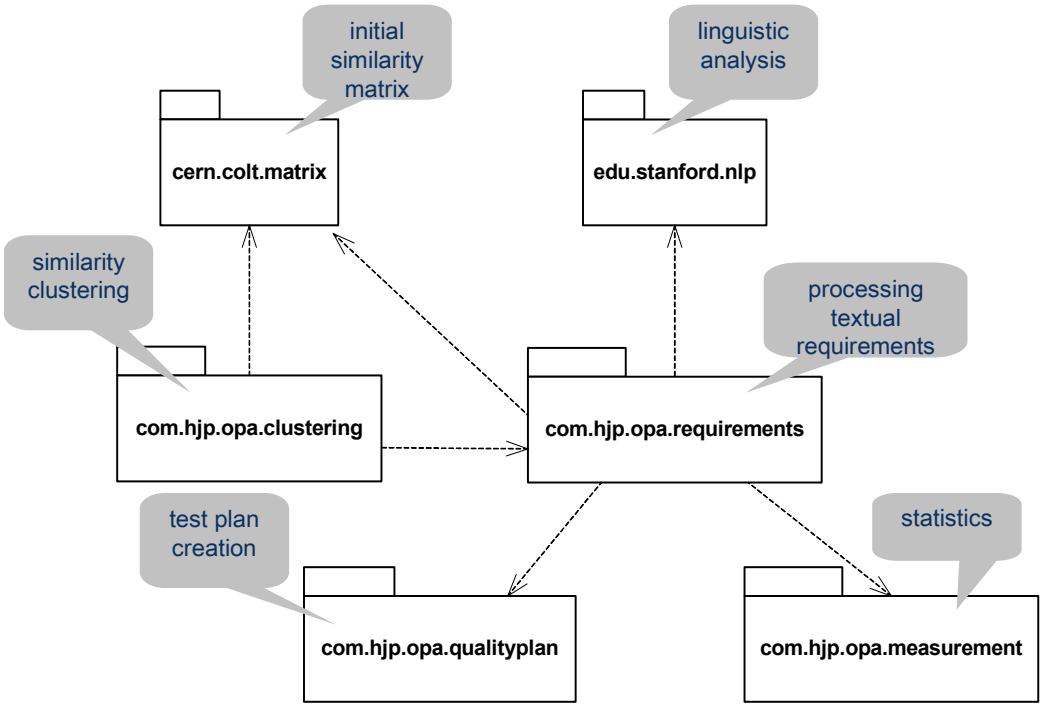


Figure 27: Architectural overview of the TORC (Test Plan Optimization by Requirements Clustering) tool environment

Most of the activities in our process are automated. This makes our approach less error-prone than manual activities. Also reproducibility is granted by automation. The process is iterative and thus enables us to review the results of each step and make improvements. Tool support (see Figure 27) is provided for automated detection of the redundancies and implicit relations in requirements [GFJ+09], but also for measurement and the generation of quality plans [GSW+10].

We have applied this approach in a joint research project with an industrial partner, an international consulting company, who are specialized in planning, procurement and acceptance testing of national electronic identification (e-ID) systems. The results show that linguistic analysis and clustering techniques can help testers in understanding the relations between requirements and for improving test planning.

6.4.2 Integrating Quality Methods in Agile Processes

Agile methods like SCRUM have gained much recognition in recent years. Although they are successfully applied many times, some problems have appeared in practice. Specifically, non-functional requirements such as performance or usability are not or only insufficiently considered. We have developed a process model how customer and user feedback can improve the quality assurance in agile development processes [EGSP09]. We focus on the improved consideration and observation of non-functional requirements. We identify five problems of the agile SCRUM process model as regards quality assurance that are solved by our approach. To improve this situation and to firmly establish early feedback from customers and users during a sprint cycle, we have extended the SCRUM process by an additional activity for quality assurance. Customer, user and their timely feedback to the development team are incorporated into the process such that the feedback can be considered for further development, future decisions and planning.

In particular, we included an additional activity for quality assurance together with the customer at the end of each iterative cycle (sprint). Customer and users test the software for a period of one to three days prior to delivery of new software functions at the end of a sprint to obtain qualified feedback. Based on the incorporation of this activity, we can validate the new functions; and involving customers, users and their timely feedback to the development team become a fixed part of the process. Based on the feedback, additional entries in the product backlog with non-functional requirements can be considered and prioritized during the planning of a new cycle. In addition to the new activity, we add a user-testing day to the process. Many application users are invited to test the current state of the software. With the user-testing day, we can execute performance and load tests with a large number of users in the usage environment, in addition to automated tests.

This work also considers the issue of software process improvement. We have identified a number of problems that limit agile development processes in general and SCRUM in particular in the areas of quality assurance and non-functional requirements. The new quality assurance (QA) activity that is done by the customer in the form of QA days helps to solve these problems. Customer and users are directly involved in each sprint by this activity. Due to the incremental process model, composed from sprints, it was possible to introduce, implement and evaluate the extension of the process model incrementally while the development process was running.

The practical implementation of the new QA activity in a real project showed that the feedback has increased attention towards the fulfillment of the requirements. Among the non-functional requirements, the main focus was on performance. Performance was not only considered, but occasionally attained central attention. This can be observed from the high priority it achieved in the product backlog. Time for performance and usability

testing is firmly scheduled and reserved due to the additional QA activity and the user-testing day, while the duration of a sprint is not significantly prolonged.

We also discussed how the initial consideration of non-functional requirements in SCRUM processes can be improved. As one possible solution we postulated the introduction of refactoring sprints. Developers get the time for extensive performance and usability testing and for revising and improving the already developed functions.

Due to the incremental character of agile processes and their division in iterative sprints, not only the expected functionality of the final project is produced stepwise in agreement of development team and customer. Also the process model is incrementally evaluated and changed. In this sense, agile process models are qualified for live adaptation while the process is executed.

6.5 Architecture-driven Development: Software Stacks

The successful and efficient development of business information systems increasingly depends on the systematic reuse of software. In addition to reusable software components, pre-assembled frameworks and, recently, so-called *software stacks* have been gaining more and more importance. Many of them are available as open source software. Open-source stacks (OSS) are a pre-configured assembly of open-source components (or frameworks). They build a sophisticated basis for the development of new software systems. The open-source components of the stack interoperate such that the open-source stack can be used as an integrated unit. In [CS08], we characterize open-source stacks and describe typical occurrences. We also describe the software development process with open-source stacks. We distinguish four roles that companies can play in the market of open-source stacks: OSS developer, OSS distributor, OSS consultant, OSS user. This shows that in a software lifecycle model of large and distributed – with respect to development and use – software systems, the concept of role can go far beyond development teams, with respect to both the organizational (beyond individuals) and the lifecycle (beyond development) dimension. This can also be observed in global software engineering as well as in the development of eID systems, where roles like component supplier, system integrator, etc. are played by different organizations.

The development of open-source stacks is, with respect to reuse of standard solutions, the consequent next step of software engineering following software components and frameworks. The reuse of open-source software components becomes easier and more effective by the provision of pre-assembled software stacks with accompanying services. Open source stacks support the effective reuse of freely available software. This helps to reduce cost and development time for the development of new software. The use of pre-assembled building blocks also increases the quality of the applications that are developed. The Interactive Knowedge Stack that we are currently developing in a joint European research project follows this paradigm. The semantic technology stack is intended to be used by providers of content management systems.

The development of open-source stacks has to address a number of challenges: selection, coupling, actuality and compatibility of components, and quality assurance of open-source stacks. Business models are manifold according to the aforementioned

roles. The tendency that companies increasingly employ open-source solutions is an indicator for the business potential of open-source software.

Another trend for new software engineering methods is software lifecycle management beyond the software development cycle. This is especially important in the presence of long-living software systems, where evolution and maintainability play an important role to prevent software aging and software erosion [GKM+10], [ERMS09]. Important methods in this domain are model-based and model-driven development methods. Yet, future-proof architectures, such as particular architectural styles or the consequent use of open-source software stacks have the potential to contribute to this domain.

After having completed the review of previous works on software engineering methods, we now step forward to the domain of method engineering as a means for the systematic development of software engineering methods.

7 Method Engineering

Answering the demand for the systematic development of software engineering methods and based on the insights and experience gained from different method development projects, I have developed MetaME, a meta-method for the engineering of software engineering methods [ES10]. It goes back to previous work where we defined a method engineering process that is centered upon the definition of a domain model of software engineering concepts and artifact types [ESS08]. The meta-method is described in detail in Part II of this thesis. The meta-method is adapted and applied in **[Sau11]** for the domain of model-driven development of advanced user interfaces.

The objective of [ESS08] is to provide a unified understanding of software engineering concepts and software artifacts, and their interrelationships. This understanding is the key to successful software development. As a solution, we define a company-wide and comprehensive ontological domain model of software engineering concepts and artifact types. On this foundation, we select or define modeling languages, the process model, and adequate tools.

In several research projects of s-lab – Software Quality Lab, we have analyzed how an appropriate company-wide software engineering method can be determined. We found that modeling languages like the UML and corresponding modeling tools are already used in software companies. But a coherent, commonly understood and accepted software engineering method is typically missing. This often has the following causes:

(1) lack of common understanding of the terminology for development artifacts,
(2) lack of common understanding how the development artifacts are interrelated, and which dependencies and refinement relationships exist between them,
(3) belief that simply by selecting UML diagram types for modeling, their purpose in the software development process is already determined,
(4) belief that simply by employing a commercial UML modeling tool, a software engineering method is determined.

Based on these observations, we have developed a process model for the systematic development of a company-specific software engineering method. The basic idea is to develop a domain model of software engineering concepts first. This provides the common understanding among the software developers (1). Adequate modeling and implementation languages are selected, which are then assigned to the software engineering concepts. As a result, we get a domain model of the software engineering building blocks (called artifact types) that are to be used in the company (2). Thus, the use of the modeling language corresponds to the identified software engineering concepts (3). On this foundation, the process is defined as a roadmap through the network of artifact types. Eventually, tools that fit the method are selected and provided to the software developers together with a method-conformant concept of use (4).

The presented method engineering process has been repeatedly applied in joint research projects with industry, for example in the development and methodical alignment of the Quasar and Quasar Enterprise methods [BEH+09] for the systematic development of large business information systems (see also Section 6) and application landscape design, respectively.

The method engineering process is one ingredient for a method engineering method. It must be complemented by an explicit model of method engineering concepts, methodical guidelines for the process steps, and method description languages – candidate languages are e.g. SPEM (see Section 10.5) or ISO 24744 (see Section 10.6) – and eventually tools. Based on the process of [ESS08], MetaME, the proposed meta-method for modeling and tailoring of software engineering methods, has been developed and introduced in [SE10].

MetaME is a meta-method for method engineering of software engineering methods. It builds on a four layer meta-model hierarchy which combines the two domains method engineering and software engineering. It combines a meta-model as a general product model of method engineering and a method engineering process model for developing software engineering methods. The process model consists of 5+1 steps. In addition, we propose to specify the tasks of a software engineering method as transformation rules that are typed over the software artifact model. Together these models cover the product and the process dimension of the meta-method. MetaME is described in detail in Part II of this thesis.

The meta-method for method engineering has recently been specialized and applied for the design of model-driven development methods in the system domain of advanced user interfaces (MDDAUI) **[Sau11]**. This work was motivated by the increasing complexity of user interfaces of interactive systems due to new interaction paradigms, required adaptability, use of innovative technologies, multimedia, and interaction modalities. Their development thus demands for sophisticated processes and methods. I propose to adopt and adapt software engineering principles to succeed. In addition to well-defined development methods, particularly model-driven development is identified as a promising candidate for mastering the complex development task in a systematic, precise and appropriately formal way. Although diverse models of advanced user interfaces are deployed in a development process to specify, design and implement the user interface, it is not standardized which models to use, how to combine them, and how to proceed in the course of development. Rather, this has to be defined by methods in the context of organizations, domains, projects. To cope with the definition of model-driven development methods for advanced user interfaces, we propose to use the meta-method for method engineering. It builds on the concept of object-oriented meta-modeling based on the 4-layer MOF architecture, yet extends it to account not only for the product model, but also for the work definitions and workflows that form the process model. The meta-method can be used for modeling and tailoring such development methods. **[Sau11]** demonstrates how to apply this meta-method for designing development methods in the domain of advanced user interfaces.

Based on the analysis of requirements for a MDDAUI development method – which originate from both the system domain of (advanced) user interfaces and the method domain of model-driven development – we adapt the general method engineering meta-method of [ES10] to cover models and model transformations as first-class citizens of the method description. We also show results from applying the meta-method to the target domain, especially graph transformation rules for the specification of tasks, activities and transformations in a user interface development process.

8 Concluding Remarks

I have presented a framework for software and method engineering in this Part I of my thesis. According to this framework, I have defined a classification scheme for software engineering methods. This scheme was used to categorize a collection of previous method engineering endeavors of fundamental nature and in different system domains. Selected publications of these works are contributed to this thesis in Part III. The experience gained from developing these software engineering methods has led to research on the systematic development of software engineering methods and a *method engineering* meta-method. This meta-method will be presented in the second part of this thesis.

Currently, we are developing MMASQ, a model-based method for analysis, specification, and qualification of complex, distributed IT systems in cooperation with an industrial partner. There, we apply systematic method engineering and our meta-method to develop a method and dedicated tool support that spans from requirements engineering and specification to the qualification of eID and other distributed IT systems.

Since software engineering methods do only provide value if they are employed in practice, it is important to transfer and deploy them in real software development endeavors. We have seen that many of the aforementioned methods have been developed jointly with industrial partners or have been applied in practice. To this end, the s-lab – Software Quality Lab provides an institutional platform for both the collaborative research and development of software engineering methods together with industrial partners and the transfer of methodical software engineering knowledge between academia and industry – research and practice [EGS06]. This applies to software engineering methods as well as software quality assurance methods, especially software testing [BS10] – and their integration.

References

[BEH+09] Andrea Baumann, Gregor Engels, Alexander Hofmann, Stefan Sauer, Johannes Willkomm: A Holistic Software Engineering Method for Service-Oriented Application Landscape Development. In Proc. First NAF Academy Working Conference on Practice-Driven Research on Enterprise Transformation (PRET 2009), Amsterdam, The Netherlands, Volume 28 of Lecture Notes in Business Information Processing (LNBIP), pp. 1–17. Springer, Berlin Heidelberg 2009.

[CS08] Fabian Christ, Stefan Sauer: Open Source Stacks. In M. Asche, W. Bauhus, E. Mitschke, B. Seel (eds.): Open Source: Kommerzialisierungsmöglichkeiten und Chancen für die Zusammenarbeit von Hochschulen und Unternehmen, Vol. 3 of Patent Offensive Westfalen Ruhr, pp. 133–154, Waxmann, Münster 2008.

[DEM+98] Ralph Depke, Gregor Engels, Katharina Mehner, Stefan Sauer, Annika Wagner: Ein Ansatz zur Verbesserung des Entwicklungsprozesses von Multimedia-Anwendungen. In Proc. Softwaretechnik '98, September 7-9, 1998, Paderborn, Germany. Softwaretechnik-Trends 18(3):12–19, 1998.

[DEM+99] Ralph Depke, Gregor Engels, Katharina Mehner, Stefan Sauer, Annika Wagner: Ein Vorgehensmodell für die Multimedia-Entwicklung mit Autorensystemen. Informatik Forschung und Entwicklung 14(2):83–94, 1999.

[EGS01] Gregor Engels, Jens Gaulke, Stefan Sauer: Modelle für automobile Software – Objektorientierte Modellierung von eingebetteten, interaktiven Softwaresystemen im Automobil. Forschungsforum Paderborn 4:22–27. Universität Paderborn, 2001.

[EGS06] Gregor Engels, Matthias Gehrke, Stefan Sauer: Multi-Private Public Partnership (MPPP) – Softwaretechnik auf dem Weg in die Industrie. In Chr. Hochberger, R. Liskowsky (eds.): Proc. INFORMATIK 2006 - Informatik für Menschen, Band 1, October 2006, Dresden, Germany, Workshop Vernetzung von Software Engineering Expertise in Industrie und Forschung (VSEEIF), Volume P-93 of GI-Edition - Lecture Notes in Informatics (LNI), pp. 281–287. Köllen Druck+Verlag GmbH, Bonn 2006.

[EGS08] Gregor Engels, Baris Güldali, Stefan Sauer: Formalisierung der funktionalen Anforderungen mit visuellen Kontrakten und deren Einsatz für modellbasiertes Testen. Softwaretechnik-Trends 28(3):12–16, 2008.

[EGSP09] Gregor Engels, Silke Geisen, Stefan Sauer, Olaf Port: Sicherstellen der Betrachtung von nicht-funktionalen Anforderungen in SCRUM-Prozessen durch Etablierung von Feedback. In S. Fischer, E. Maehle, R. Reischuk (eds.): Proc. INFORMATIK 2009 – Im Focus das Leben, September 28 - October 2, 2009, Lübeck, Germany, Volume P-154 of GI-Edition - Lecture Notes in Informatics (LNI), p. 458, Köllen Druck+Verlag GmbH, Bonn 2009.

[EHHS00] Gregor Engels, Jan Hendrik Hausmann, Reiko Heckel, Stefan Sauer: Dynamic Meta Modeling: A Graphical Approach to the Operational Semantics of Behavioral Diagrams in UML. In A. Evans, S. Kent, B. Selic (eds.): Proc. UML 2000, October 2-6, 2000, York, UK, Volume 1939 of Lecture Notes in Computer Science (LNCS), pp. 323–337. Springer, Berlin Heidelberg 2000.

[EHHS02] Gregor Engels, Jan Hendrik Hausmann, Reiko Heckel, Stefan Sauer: Testing the Consistency of Dynamic UML Diagrams. In Proc. Sixth International Conference on Integrated Design and Process Technology (IDPT 2002), June 23-28, 2002, Pasadena, CA, USA.

[EHS99] Gregor Engels, Reiko Heckel, Stefan Sauer: Dynamic Meta Modelling: A Graphical Approach to Operational Semantics. In Proc. OOPSLA'99 Workshop on Rigorous Modeling and Analysis with the UML: Challenges and Limitations, November 2, 1999, Denver, Colorado, USA.

[EHS00] Gregor Engels, Reiko Heckel, Stefan Sauer: UML - A Universal Modeling Language? In M. Nielsen, D. Simpson (eds.): Proc. 21st International Conference on Application and Theory of Petri Nets (Petri Nets 2000), June 2000, Aarhus, Denmark, Volume 1825 of Lecture Notes in Computer Science (LNCS), pp. 24–38. Springer, Berlin Heidelberg 2000.

[EHSW99a] Gregor Engels, Roland Hücking, Stefan Sauer, Annika Wagner: UML Collaboration Diagrams and Their Transformation to Java. In R. France, B. Rumpe (eds.): Proc. UML'99 - The Unified Modeling Language, October 28-30, 1999, Fort Collins, Colorado, USA, Volume 1723 of Lecture Notes in Computer Science (LNCS), pp. 473–488. Springer, Berlin, Heidelberg 1999.

[EHSW99b] Gregor Engels, Roland Hücking, Stefan Sauer, Annika Wagner: UML Collaboration Diagrams and Their Transformation to Java. Technical Report tr-ri-99-208, Fachbereich Mathematik - Informatik, Universität Paderborn, Germany, June 1999. Extended version of [EHSW99a].

[ELS05a] Gregor Engels, Marc Lohmann, Stefan Sauer: Design by Contract zur semantischen Beschreibung von Web Services. In A. B. Cremers, R. Manthey, P. Martini, V. Steinhage (eds.): INFORMATIK 2005 - Informatik LIVE!, Band 1, September 19-22, 2005, Bonn, Germany, Workshop Service-orientierte Architekturen - Zusammenwirken von Business & IT, Volume P-68 of GI-Edition - Lecture Notes in Informatics (LNI), pp. 612–616, Köllen Druck+Verlag GmbH, Bonn 2005.

[ELS05b] Gregor Engels, Marc Lohmann, Stefan Sauer: Modellbasierte Entwicklung von Web Services mit Design by Contract. In A. B. Cremers, R. Manthey, P. Martini, V. Steinhage (eds.): INFORMATIK 2005 - Informatik LIVE! Band 1, September 19-22, 2005, Bonn, Germany, Workshop Modellbasierte Qualitätssicherung, Volume P-68 of GI-Edition - Lecture Notes in Informatics (LNI), pp. 491–495. Köllen Druck+Verlag GmbH, Bonn 2005.

[ELSH06] Gregor Engels, Marc Lohmann, Stefan Sauer, Reiko Heckel: Model-Driven Monitoring: An Application of Graph Transformation for Design by Contract. In A. Corradini, H. Ehrig, U. Montanari, L. Ribeiro, G. Rozenberg (eds.): Proc. Third International Conference on Graph Transformations (ICGT 2006), Volume 4178 of Lecture Notes in Computer Science (LNCS), pp. 336–350. Springer, Berlin Heidelberg 2006.

[ERMS09] Gregor Engels, Ralf Reussner, Christof Momm, Stefan Sauer (eds.): Design for Future – Langlebige Softwaresysteme 2009. Proc. 1. Workshop des GI-Arbeitskreises Langlebige Softwaresysteme (L2S2): Design for Future – Langlebige Softwaresysteme, October 15-16, 2009, Karlsruhe, Germany, Volume

537 of CEUR Workshop Proceedings, ISSN 1613-0073. http://CEUR-WS.org/Vol-537/

[ES02] Gregor Engels, Stefan Sauer: Object-oriented Modeling of Multimedia Applications. In S.K. Chang (ed.): Handbook of Software Engineering and Knowledge Engineering, Vol. 2, pp. 21–53, World Scientific, Singapore 2002.

[ES04a] Gregor Engels, Stefan Sauer: Guest Editors' Introduction. International Journal of Software Engineering and Knowledge Engineering (IJSEKE) 14(6):543–544. World Scientific Publishing, Singapore 2004.

[ES04b] Gregor Engels, Stefan Sauer (eds.): Modeling and Development of Multimedia Systems. Special Issue of the International Journal of Software Engineering and Knowledge Engineering 14(6), World Scientific Publishing, Singapore 2004.

[ES10] Gregor Engels, Stefan Sauer: A Meta-Method for Defining Software Engineering Methods. In G. Engels, C. Lewerentz, W. Schäfer, A. Schürr, B. Westfechtel (eds.): Graph Transformations and Model-Driven Engineering, Essays Dedicated to Manfred Nagl on the Occasion of his 65th Birthday, Volume 5765 of Lecture Notes in Computer Science (LNCS), pp. 411–440. Springer, Berlin Heidelberg 2010.

[ESN03] Gregor Engels, Stefan Sauer, Bettina Neu: Integrating Software Engineering and User-centred Design for Multimedia Software Developments. In Proc. IEEE Symposia on Human-Centric Computing Languages and Environments (HCC' 03), October 2003, Auckland, New Zealand, Symposium on Visual/Multimedia Software Engineering (VMSE '03), pp. 254–256. IEEE Computer Society Press, Los Alamitos, CA, 2003.

[ESS08] Gregor Engels, Stefan Sauer, Christian Soltenborn: Unternehmensweit verstehen – unternehmensweit entwickeln: Von der Modellierungssprache zur Softwareentwicklungsmethode. Informatik-Spektrum 31(5):451–459, Special Issue: Modellierung. Springer, Berlin Heidelberg 2008.

[GFJ+09] Baris Güldali, Holger Funke, Michael Jahnich, Stefan Sauer, Gregor Engels: Semi-automated Test Planning for e-ID Systems by Using Requirements Clustering. In Proc. 24th IEEE/ACM International Conference on Automated Software Engineering (ASE 2009), November 16-20, 2009, Auckland, New Zealand, pp. 29–39, 2009.

[GKM+10] Rainer Gimnich, Uwe Kaiser, Christof Momm, Jochen Quante, Volker Riediger, Stefan Sauer, Mircea Trifu, Andreas Winter (eds.): Proc. 12. Workshop Software-Reengineering (WSR) & 2. Workshop Design for Future (DFF) 2010, Bad Honnef, Germany, May 3-5, 2010. Softwaretechnik-Trends 30(2):28–85, 2010.

[GS10] Baris Güldali, Stefan Sauer: Transfer of Testing Research from University to Industry: An Experience Report. In Proc. International TestIstanbul Conference 2010. Turkish Testing Board, May 2010. http://www.testistanbul.org

[GSW+10] Baris Güldali, Stefan Sauer, Peter Winkelhane, Michael Jahnich, Holger Funke: Pattern-based Generation of Test Plans for Open Distributed Processing Systems. In Proc. International Conference on Software Engineering (ICSE 2010), 5th International Workshop on Automation of Software Test (AST 2010), pp. 119–126. ACM Press, 2010.

[HHS00] Jan Hendrik Hausmann, Reiko Heckel, Stefan Sauer: Ein Konzept zur anwendungsbezogenen UML-Semantikbeschreibung durch dynamische Metamodellierung. In H. Giese, St. Philippi (eds.): Proc. 8th GROOM Workshop: Visuelle Verhaltensmodellierung verteilter und nebenläufiger Softwaresysteme (VVVNS 2000), November 13-14, 2000, Münster, Germany, pp. 64–69. Technical Report no. 24/00 I, Universität Münster, Germany 2000.

[HHS01] Jan Hendrik Hausmann, Reiko Heckel, Stefan Sauer: Towards Dynamic Meta Modeling of UML Extensions: An Extensible Semantics for UML Sequence Diagrams. In Proc. IEEE Symposia on Human-Centric Computing Languages and Environments (HCC '01), September 2001, Stresa, Italy, Symposium on Visual Languages and Formal Methods, pp. 80–87.

[HHS02a] Jan Hendrik Hausmann, Reiko Heckel, Stefan Sauer: Dynamic Meta Modeling with Time: Specifying the Semantics of Multimedia Sequence Diagrams. In P. Bottoni, M. Minas (eds.): Proc. ICGT 2002 International Workshop on Graph Transformation and Visual Modeling Techniques (GT-VMT 2002), October 2002, Barcelona, Spain. Electronic Notes in Theoretical Computer Science (ENTCS) 72(3).

[HHS02b] Jan Hendrik Hausmann, Reiko Heckel, Stefan Sauer: Extended Model Relations with Graphical Consistency Conditions. In L. Kuzniarz, G. Reggio, J. L. Sourrouille, Z. Huzar (eds.): Proc. UML 2002 Workshop on Consistency Problems in UML-based Software Development, October 2002, Dresden, Germany, pp. 61–74. Research Report 2002:06, Blekinge Institute of Technology, Sweden, 2002.

[HHS04] Jan Hendrik Hausmann, Reiko Heckel, Stefan Sauer: Dynamic Meta Modeling With Time: Specifying the Semantics of Multimedia Sequence Diagrams. Journal of Software and Systems Modeling (SOSYM) 3(3):181–192, 2004.

[HKS01] Jan Hendrik Hausmann, Jochen M. Küster, Stefan Sauer: Identifying Semantic Dimensions of (UML) Sequence Diagrams. In A. Evans, R. France, A. Moreira, B. Rumpe (eds.) Proc. Practical UML-Based Rigorous Development Methods - Countering or Integrating the eXtremists, UML 2001 Workshop of the pUML-Group, October 2001, Toronto, Canada, Volume P-7 of GI-Edition - Lecture Notes in Informatics (LNI), pp. 142–157. Köllen Druck+Verlag GmbH, Bonn 2005.

[HS00a] Reiko Heckel, Stefan Sauer: Dynamische Metamodellierung als Methode zur Definition einer operationalen Semantik für die UML. In Proc. 7th GI-Workshop GROOM, April 4-5, 2000, Universität Koblenz-Landau. Softwaretechnik-Trends 20(2):43–44, 2000.

[HS00b] Reiko Heckel, Stefan Sauer: Strengthening the Semantics of UML Collaboration Diagrams. In G. Reggio, A. Knapp, B. Rumpe, B. Selic, R. Wieringa (eds.): Proc. UML 2000 Workshop on Dynamic Behavior in UML Models: Semantic Questions, October 2, 2000, York, UK, pp. 63–69. Technical Report no. 0006, Ludwig-Maximilians-Universität München, Germany, 2000.

[HS01] Reiko Heckel, Stefan Sauer: Strengthening UML Collaboration Diagrams by State Transformations. In H. Hussmann (ed.): Proc. 4th International Conference Fundamental Approaches to Software Engineering (FASE 2001), April 2001,

Genova, Italy, Volume 2029 of Lecture Notes in Computer Science (LNCS), pp. 109–123. Springer, Berlin Heidelberg 2001.

[LES06] Marc Lohmann, Gregor Engels, Stefan Sauer: Model-driven Monitoring: Generating Assertions from Visual Contracts. In Proc. 21st IEEE/ACM International Conference on Automated Software Engineering (ASE 2006), pp. 355–356. IEEE Computer Society, 2006.

[LRE+06] Marc Lohmann, Jan-Peter Richter, Gregor Engels, Baris Güldali, Oliver Juwig, Stefan Sauer: Semantische Beschreibung von Enterprise Services – Eine industrielle Fallstudie. s-lab Report No.1, Software Quality Lab (s-lab), Universität Paderborn, Germany, ISSN 1863-0774, May 2006.

[LSE05] Marc Lohmann, Stefan Sauer, Gregor Engels: Executable Visual Contracts. In Proc. 2005 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC' 05), September 21-24, 2005, Dallas, Texas, USA, pp. 63–70.

[LSS03] Marc Lohmann, Stefan Sauer, Tim Schattkowsky: ProGUM-Web: Tool Support for Model-Based Development of Web Applications. In P. Stevens, J. Whittle, G. Booch (eds.): Proc. 6th International Conference on the Unified Modeling Language (UML 2003), October 2003, San Francisco, CA, USA, Volume 2863 of Lecture Notes in Computer Science (LNCS), pp. 101–105. Springer, Berlin Heidelberg 2003.

[MGB+09a] Gerrit Meixner, Daniel Görlich, Kai Breiner, Heinrich Hußmann, Andreas Pleuß, Stefan Sauer, Jan Van den Bergh: Fourth International Workshop on Model Driven Development of Advanced User Interfaces. In Proc. 2009 International Conference on Intelligent User Interfaces, February 8, 2009, Sanibel Island, Florida, USA, pp. 503–504. ACM Press, 2009.

[MGB+09b] Gerrit Meixner, Daniel Görlich, Kai Breiner, Heinrich Hußmann, Andreas Pleuß, Stefan Sauer, Jan Van den Bergh (eds.): Proc. Workshop on Model Driven Development of Advanced User Interfaces (MDDAUI'09), February 8, 2009, Sanibel Island, Florida, USA, Volume 439 of CEUR Workshop Proceedings, ISSN 1613-0073. http://CEUR-WS.org/Vol-439/

[PVHS05] Andreas Pleuß, Jan Van den Bergh, Heinrich Hußmann, Stefan Sauer (eds.): MDDAUI'05 – Model Driven Development of Advanced User Interfaces 2005. Proceedings of the MoDELS'05 Workshop on Model Driven Development of Advanced User Interfaces. Volume 159 of CEUR Workshop Proceedings, ISSN 1613-0073. http://CEUR-WS.org/Vol-159/

[PVH+07] Andreas Pleuß, Jan Van den Bergh, Heinrich Hußmann, Stefan Sauer, Daniel Görlich (eds.): MDDAUI'07, Proceedings of the MODELS'07 Workshop on Model Driven Development of Advanced User Interfaces, Volume 297 of CEUR Workshop Proceedings, ISSN 1613-0073. http://CEUR-WS.org/Vol-297/

[PVSH06] Andreas Pleuß, Jan Van den Bergh, Stefan Sauer, Heinrich Hußmann: Workshop Report: Model Driven Development of Advanced User Interfaces (MDDAUI). In J.M. Bruel (ed.): Proc. Satellite Events at the MoDELS 2005 Conference, MoDELS 2005 International Workshops, Doctoral Symposium, Educators Symposium, Revised Selected Papers, Volume 3844 of Lecture Notes in Computer Science (LNCS), pp. 182–190. Springer, Berlin Heidelberg 2006.

[PVS+06] Andreas Pleuß, Jan Van den Bergh, Stefan Sauer, Heinrich Hußmann, Alexander Bödcher (eds.): MDDAUI'06 – Model Driven Development of Advanced User Interfaces 2006. Proceedings of the MoDELS'06 Workshop on Model Driven Development of Advanced User Interfaces. Volume 214 of CEUR Workshop Proceedings, ISSN 1613-0073. http://CEUR-WS.org/Vol-214/

[PVS+07] Andreas Pleuß, Jan Van den Bergh, Stefan Sauer, Heinrich Hußmann, Alexander Bödcher: Model Driven Development of Advanced User Interfaces (MDDAUI) – MDDAUI'06 Workshop Report. In T. Kühne (ed.): MoDELS 2006 Workshops, October 1-6, 2006, Genova, Italy, Volume 4364 of Lecture Notes in Computer Science (LNCS), pp. 100–104. Springer, Berlin Heidelberg 2007.

[PVS+08] Andreas Pleuß, Jan Van den Bergh, Stefan Sauer, Daniel Görlich, Heinrich Hußmann: Third International Workshop on Model Driven Development of Advanced User Interfaces. In: H. Giese (ed.): Models in Software Engineering, Workshops and Symposia at MoDELS 2007, Reports and Revised Selected Papers, September 30 - October 5, 2007, Nashville, TN, USA, Volume 5002 of Lecture Notes in Computer Science (LNCS), pp. 59–64. Springer, Berlin Heidelberg 2008.

[Sau11] Stefan Sauer: Applying Meta-Modeling for the Definition of Model-Driven Development Methods of Advanced User Interfaces. In: H. Hussmann, G. Meixner, D. Zuehlke (eds.): Model-driven Development of Advanced User Interfaces, Volume 340 of Studies in Computational Intelligence, pp. 67–86. Springer, Berlin Heidelberg 2011.

[SDGH06] Stefan Sauer, Markus Dürksen, Alexander Gebel, Dennis Hannwacker: GuiBuilder – A Tool for Model-Driven Development of Multimedia User Interfaces. In A. Pleuss, J. Van den Bergh, H. Hußmann, S. Sauer, A. Bödcher (eds.): Proc. MDDAUI'06 - Model Driven Development of Advanced User Interfaces, Volume 214 of CEUR Workshop Proceedings, ISSN 1613-0073. http://CEUR-WS.org/Vol-214/

[SE99a] Stefan Sauer, Gregor Engels: Extending UML for Modeling of Multimedia Applications. In Proc. 1999 IEEE Symposium on Visual Languages (VL '99), September 13-16, 1999, Tokyo, Japan, pp. 80–87. IEEE Computer Society, 1999. *Recognized with Most Influential Paper Award of IEEE Symposium on Visual Languages and Human-Centric Computing 2010.*

[SE99b] Stefan Sauer, Gregor Engels: MVC-Based Modeling Support for Embedded Real-Time Systems. Position Statement. In P. Hofmann, A. Schürr (eds.): Proc. Workshop Objektorientierte Modellierung eingebetteter Realzeitsysteme (OMER), May 28-29, 1999, Herrsching (Ammersee), Germany, pp. 11–14. Technical Report 1999-01, Fakultät Informatik, Universität der Bundeswehr München, Germany, May 1999.

[SE99c] Stefan Sauer, Gregor Engels: OMMMA: An Object-oriented Approach for Modeling Multimedia Information Systems. In L. Golubchik, V. J. Tsotras (eds.): Proc. 5th International Workshop on Multimedia Information Systems (MIS '99), October 21-23, 1999, Indian Wells, California, USA, pp. 64–71.

[SE99d] Stefan Sauer, Gregor Engels: UML-basierte Modellierung von Multimediaanwendungen. In J. Desel, K. Pohl, A. Schürr (eds.): Proc.

Modellierung '99, March 10-12, 1999, Karlsruhe, Germany, pp. 155–170. Teubner, Stuttgart 1999.

[SE01] Stefan Sauer, Gregor Engels: UML-based Behavior Specification of Interactive Multimedia Applications. In Proc. IEEE Symposia on Human-Centric Computing Languages and Environments (HCC '01), September 2001, Stresa, Italy, pp. 248–255. *Recognized with Best Paper Award of HCC'01 Symposium on Visual/Multimedia Approaches to Programming and Software Engineering.*

[SE07] Stefan Sauer, Gregor Engels: Easy Model-Driven Development of Multimedia User Interfaces with GuiBuilder. In C. Stephanidis (ed.): Universal Access in Human-Computer Interaction, Proc. 4th International Conference on UAHCI 2007, HCI International 2007, Part II: Universal Access Methods, Techniques and Tools, July 22-27, 2007, Beijing, China, Volume 4554 of Lecture Notes in Computer Science (LNCS), pp. 537–546. Springer, Berlin, Heidelberg 2007.

[SSE09a] Frank Salger, Stefan Sauer, Gregor Engels: An Integrated Quality Assurance Framework for Specifying Business Information Systems. In E. Yu, J. Eder, C. Rolland (eds.): Proc. Forum at the CAiSE 2009 Conference, Amsterdam, The Netherlands, Volume 453 of CEUR Workshop Proceedings, ISSN 1613-0073, pp. 25–30, 2009. http://CEUR-WS.org/Vol-453/

[SSE09b] Frank Salger, Stefan Sauer, Gregor Engels: Integrated Specification and Quality Assurance for Large Business Information Systems. In Proc. 2nd India Software Engineering Conference (ISEC '09), pp. 129–130. ACM Press, 2009.

[SSEB10] Frank Salger, Stefan Sauer, Gregor Engels, Andrea Baumann: Knowledge Transfer in Global Software Development - Leveraging Ontologies, Tools and Assessments. In Proc. 5th IEEE International Conference on Global Software Engineering (ICGSE 2010), pp. 336–341, 2010.

[VMB+10] Jan Van den Bergh, Gerrit Meixner, Kai Breiner, Andreas Pleuss, Stefan Sauer, Heinrich Hußmann: Model-driven Development of Advanced User Interfaces. In Proc. 28th International Conference on Human Factors in Computing Systems (CHI 2010), Extended Abstracts Volume, April 10-15, 2010, Atlanta, Georgia, USA, pp. 4429–4432. ACM Press, 2010.

[VMS10] Jan Van den Bergh, Gerrit Meixner, Stefan Sauer: MDDAUI 2010 Workshop Report. In Proc. 5th International Workshop on Model Driven Development of Advanced User Interfaces (MDDAUI 2010), Volume 617 of CEUR Workshop Proceedings, ISSN 1613-0073, pp. 53–56. http://CEUR-WS.org/Vol-617/, urn:nbn:de:0074-617-8

[VSB+10] Jan Van den Bergh, Stefan Sauer, Kai Breiner, Heinrich Hußmann, Gerrit Meixner, Andreas Pleuss (eds.): Proceedings of the 5th International Workshop on Model Driven Development of Advanced User Interfaces (MDDAUI 2010): Bridging between User Experience and UI Engineering, April 10, 2010, Atlanta, Georgia, USA, Volume 617 of CEUR Workshop Proceedings, ISSN 1613-0073. http://CEUR-WS.org/Vol-617/, urn:nbn:de:0074-617-8

Part II:

MetaME – A Meta-Method for Method Engineering of Software Engineering Methods

9 Engineering of Software Engineering Methods

Software systems such as business information systems are constantly growing in size and complexity. At the same time, they need to be produced in dependable quality while their development shall be cost and resource effective. To meet all these requirements, the development of software systems demands for sophisticated software engineering methods and processes.

In this work we use the term *software engineering method* to denote the full set of elements needed to describe a software development endeavor, such as a software development project, in all relevant aspects. This does not only cover the software development process and its contained activities, but also the artifacts that are to be produced, the tasks that need to be performed to achieve the development goals, the roles in an organization that participate in the development, the tools, techniques and utilities that are employed, as well as relationships between these concepts.

To obtain such software engineering methods, their own development should be done systematically and have a sound methodical foundation. This is the objective of *method engineering* [Bri96]. Method engineering is an engineering discipline that deals with the development of methods, techniques, and tools for the development of software systems. It started in the area of information systems in the 1990s (e.g. [Gut94, Bri96]), but was taken up by software engineering in the following (see e.g. [NFK94, Rol09, HR10]). Method engineering aims at providing a framework for defining and tailoring system and software engineering (SE) methods. It allows us to model and analyze even complex software engineering methods in a systematic way.

Different software engineering methods and processes have been proposed and are in use for different purposes, such as the Rational Unified Process (RUP) [IBM07] or agile methods like SCRUM [SB02] and many more. However, it is widely recognized that such standards are often too generic to be directly applicable and thus must be tailored to the problem at hand (see e.g. [Wie03]) before they can effectively be employed. It becomes also necessary to develop new methods due to the advent of new development paradigms; or domain-specific methods that account for the specifics of a certain domain like business information systems or business intelligence systems (see Section 6); or for a particular delivery model such as global software development [SSEB10]. Hence there is still a need to derive, evolve and develop new software engineering methods.

Tailoring of methods is necessary since there exists no standard method that perfectly suites all types of projects in all domains. It is also not reasonable to develop a new method every time when a context-specific method is needed. It is much more economic to tailor existing methods to the current development context and situation. A number of method engineering approaches have been proposed that especially deal with the development of methods for a particular situation, which is known as *situational method engineering* (see e.g. [RBH07, BKPJ07, HR10]). Mechanisms for reuse and adaptation play an important role in this field. In addition, component-like concepts that support modularity of methods are often used, such as viewpoint templates [NFK96], method fragments [Bri96], or method chunks [Rol09]. In recent publications, even the use of method services and the notion of method-as-a-service are discussed [Rol09].

Another line of research has been focusing on the use of meta-modeling for the representation of method contents. A number of meta-models have been developed in result, see e.g. [HG05, GMH05, BG08, JJM09]. With the Object Management Group's (OMG) Software & Systems Process Engineering Meta-Model (SPEM) [OMG08] and the ISO/IEC 24744 Software Engineering – Metamodel for Development Methodologies [ISO07] even two standards for describing the content of software engineering methods arrived (compare Sections 10.5 and 10.6, respectively).

The advantages of a meta-model-based approach are manifold: First, meta-modeling provides a formal and precise foundation for the specification of software engineering methods. Secondly, software engineering methods can be compared on the formal basis provided by the meta-model, acting as a reference framework. Third, the formalization provides a precise foundation for the development of tools and utilities that support the use of the method. Fourth, the meta-model can be employed for analyzing a software engineering method for certain properties such as consistency, conformance, etc. Last but not least, the meta-model provides a formal basis for tailoring. Changes to software engineering methods can be traced back to the meta-model and checked for their conformance and consequences, since methods are instances of the meta-model.

In my analysis I have observed a number of reasons why the current approaches still have some shortcomings. The first and most obvious point is the lack of a process definition that specifies how to develop a software engineering method based on the meta-modeling approach; in other words, how a method engineer should instantiate the meta-model. Neither do they define the tasks that are needed for method engineering nor a process workflow to follow. Strictly speaking, meta-models like SPEM do not define a meta-model for software engineering methods, but a meta-model for method descriptions. They provide a linguistic rather than an ontological meta-model. That means, they define a modeling language for software engineering methods.

Secondly, most approaches lack a sound integration of the product and the process aspect of a software engineering method in a coherent, yet manageable meta-modeling architecture. For example, the OMG favors to complement SPEM [OMG08] with the UML meta-model for the definition of the models to be produced, and some behavior modeling formalism like BPMN to define the behavior of the methods process part. ISO 24744 proposes to combine the different aspects by the use of powertype patterns and clabjects [ISO07, GH08]. ISO thus provides a sophisticated, yet challenging formal approach to address the integration issues that contrasts the ideas of strict meta-modeling as discussed in [AK01].

Additionally, although most approaches offer some means to interrelate the process and product aspects at least on a high level of abstractions – such as products that are used or created by tasks and activities in the process model, or roles that are responsible for products or perform tasks – they lack the possibility to model complex patterns of interlinked structural and behavioral models. For example, SPEM allows the method engineer to define work products (artifacts) and work elements such as tasks, processes, activities, and steps of tasks. Furthermore, state models can be assigned to work product definitions; and the states and transitions of the work products' state models can be related to work elements. But this linking is restricted to the work product lifecycle of individual work products. What cannot be expressed in a formal way is the effect of work elements on the artifact object structure, i.e., the network of interrelated work

product instances. Instead of coding this by states, we wish to have an explicit mechanism for defining such transformations.

Based on this observation, it is the objective of this work to combine method engineering, meta-modeling and ideas from language engineering for the development of software engineering methods and to present a concise *meta-method* in this part of the thesis for defining and tailoring software engineering methods. The meta-method is designed to support the definition of software engineering methods as well as the tailoring of software engineering methods for particular domains and projects, i.e., as an approach for situational method engineering.

The meta-method must cover both the product and the process dimension of the software engineering method. In the product dimension, the method engineer must specify which artifacts are to be produced in the course of applying the method and how these artifacts are related. In the process dimension, it must be defined how to proceed for producing the artifacts and what needs to be accomplished – and by whom – in getting from one artifact to the other. The former can be covered by process or workflow models. The latter can be achieved by the use of transformations. Such transformations can be executed as manual development tasks or by automated tasks as part of the software development process. Thus, we define the product part of the software engineering method in an artifact model and the process part of the method by workflow models and task models. The effect of task execution will be modeled by the use of transformations that operate on the artifact model.

We propose transformation rules that operate on instances of the product model. These rules can also make reference to the state model of individual work product elements, but show also their attribute values and links in the pre- and post-conditions of the transformation rule.

We use the same set of general specification means also to define the meta-method, being itself a method for the development of methods. We use a product meta-model from which software engineering methods are instantiated. The process dimension of method engineering is described by the workflow model and the task model of the meta-method. The workflow model defines how to proceed in order to define a software engineering method. The task model defines the tasks that need to be accomplished, again in the form of transformations.

This part of my thesis is structured as follows: I start with a brief description of the foundations that are relevant for the presented approach for method engineering of software engineering methods based on meta-modeling in Section 10. In particular, in Section 10.1, I give a very brief introduction to the core concepts of software engineering methods that are relevant for method engineering. Meta-modeling is introduced in Section 10.2 as a methodological means for developing software engineering methods. In Section 10.3, we deal with the domain of method engineering and then present OMG's SPEM [OMG08] as well as ISO/IEC 24744 in some detail. Section 11 constitutes the core of this part where we present our meta-method *MetaME* for method engineering of software engineering methods. Based on a foundational meta-model architecture of the meta-method, we present the product meta-model as a general information model of method engineering as well as a process model for developing software engineering methods. Steps 3 and 4 of that process that are

concerned with the development of an artifact meta-model and the software process model are looked at in detail. There we also introduce the method of specifying software engineering tasks as transformation rules that operate on the product model. The issue of tailoring in the framework of our meta-method is described in Section 12. We look at both the tailoring of software engineering methods and the meta-level tailoring of the method engineering meta-method. Section 13 concludes this part of the thesis with a summary and outlook.

10 Foundations of Method Engineering of Software Engineering Methods Based on Meta-Modeling

This section describes the foundations for the engineering of software engineering methods based on meta-modeling that are important in the context of this thesis and build the ground for the meta-method MetaME. Section 10.1 summarizes the core concepts of software engineering methods that are relevant for method engineering. Meta-modeling is introduced in Section 10.2 as a methodological means for developing software engineering methods. In Section 10.3, we consider the domain of method engineering. We then visit two meta-models for software engineering methods, namely OMG's SPEM (Section 10.5) and ISO/IEC 24744 (Section 10.6).

10.1 Software Engineering and Software Development

In order to manifest a common understanding of the fundamental concepts and central technical terms that are used throughout this part of the thesis, we will introduce these terms in this section. We look in the domain of software engineering, being the domain of the software engineering methods, and the domain of method engineering, being the domain of the meta-method.

From the software engineering perspective, the central concept of our approach is the software engineering method. We define *software engineering method* as a systematic procedure or technique of doing work in software engineering in order to reach a certain goal and/or produce a defined set of software artifacts. Software engineering methods structure, coordinate and document the development processes and activities as well as the produced artifacts (also called work products or work items). Software engineering methods are often also called software process models by other authors. However, the term process may lead to misunderstandings, since a software engineering method contains more than just a process definition. Rather, the software engineering method is concerned with the processes *and* products of software engineering. Such products are e.g. different kinds of software models that themselves may have a complex structure that needs to be specified in a comprehensive software engineering method.

Software development processes are defined by IEEE Standard 610.12 as follows [IEEE90]: "The process by which user needs are translated into a software product." The process involves *activities* such as translating user needs into software requirements, transforming the software requirements into design, implementing the design in code, testing the code, and sometimes, installing and checking out the software for operational use. Activities may overlap or be performed iteratively. From this definition we can derive the understanding, that processes are made from activities that are executed in some order. In accordance with [OMG08], we define that the processes of a software engineering method are hierarchically composed from activities. More specific concepts may be defined to capture specific kinds of sub-hierarchies on intermediate layers such as iterations and phases of development processes, like they are defined in RUP [IBM07]. These compositional structures can be seen as specific kinds of (composite) activities. Processes and activities thus define the *process structure* and *workflow* of a software engineering method.

Two other terms that are closely related to the process are software development cycle and software life cycle. They refer to the *temporal* aspect of the software engineering process. According to IEEE Standard 610.12 [IEEE90], a *software development cycle* is "the period of time that begins with the decision to develop a software product and ends when the software is delivered." The development cycle typically includes *phases* such as a requirements phase, design phase, implementation phase, test phase, and sometimes, installation and checkout phase. The phases of the cycle – alike the activities of a process – may overlap or be performed iteratively, depending on the software development approach that is employed. In contrast, the *software life cycle* is defined as "the period of time that begins when a software product is conceived and ends when the software is no longer available for use." The software life cycle thus goes beyond the software development cycle. It extends the development cycle with additional phases: The software life cycle typically also includes a concept phase upfront, and subsequent to development an installation and checkout phase, operation and maintenance phase, and, sometimes, retirement phase. Again, these phases may overlap or be performed iteratively.

A complete set of software engineering methods ideally covers the full software life cycle; at least it should cover the full software development cycle. If it only covers the software development cycle, the term software development method is as well appropriate. However, particular software engineering methods may have a more limited scope as regards the software life cycle. They may the be used as partial models and integrated in a comprehensive software engineering method that covers the full cycle.

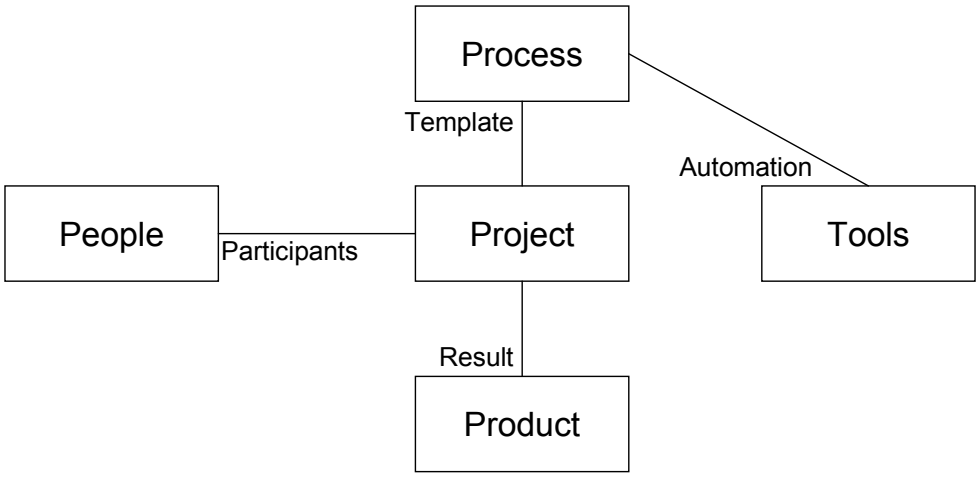


Figure 28: The core concepts that make up the Unified Software Development Process (redrawn from [JBR99])

A software engineering method comprises a set of concepts that are required for its definition. It is commonly agreed that a software engineering method has to cover at least three main aspects by its provided concepts (see e.g. [GH08]): the work products that are created and used, the process to follow, and the producers that are involved. Although different authors define different sets of concepts, some of them are commonly used (although sometimes using different terms with varying semantics) and can be seen as the agreed minimal set (sometimes further differentiated): roles (and/or people), processes and activities (and/or tasks), work products (or artifacts), tools (software tools and other utilities). The authors of the Unified Software Development

Process [JBR99], for example, identify five main concepts for software development methods: process, people, project, product, and tools (as depicted in Figure 28), thus adding the concept of project explicitly.

Gutzwiller [Gut94] has developed a meta-model for development methods and process models. It requires five general elements for describing a method: activities, roles, work products, techniques, and an information model (termed 'meta-model' in the original work), see Figure 29. These concepts are characterized as follows:

- **Activities** are functional units of work that produce one or more defined work products. They can be hierarchically structured and have a defined order of execution. This order implies the order in which work products are developed.
- **Roles** are responsible for performing one or more activities. They can be played by individual persons or groups of people.
- **Work products** are produced, used and modified by activities. They can thus be employed as input and output of an activity. Work products can be hierarchically structured.
- A **technique** provides guidance for producing one or more work products. It prescribes how and by which means to produce the work products. Techniques may be supported by tools.
- The **information model** specifies the artifact types that correspond to the work products, their attributes and the relationships between the artifact types, especially the composition hierarchy. The information model represents the conceptual data model of the development artifacts.

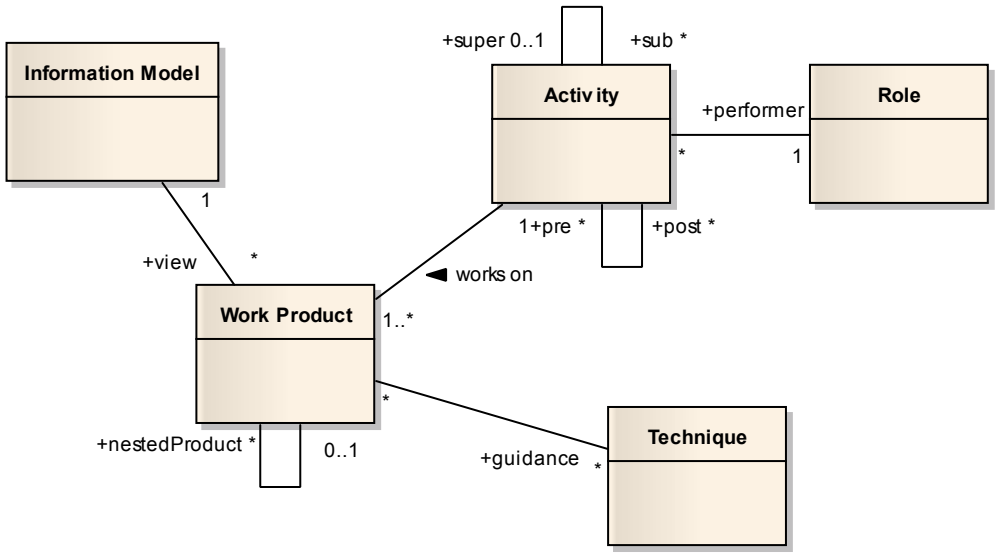


Figure 29: The core concepts of a software engineering method according to [Gut94]

According to [Bal98], a software engineering method defines the following aspects:

- workflow of development process,
- activities that are to be executed,
- definition of work products (or product parts) with respect to content and structure/layout,

- completion criteria for work products (or product parts),
- required skills for performing tasks,
- responsibilities and capabilities of workers,
- standards, guidelines, techniques and tools that are to be employed.

The Rational Unified Process [IBM07] distinguishes between the static and the dynamic aspect of a software development process. The static aspect describes the process structure that is built from activities, workflows, artifacts, and workers. The dynamic aspect of the process as it is enacted covers the temporal domain by the concepts lifecycle, phase, iteration, and milestone. Workflows group activities logically. Activities comprise activity steps. Artifacts are models, model elements, documents, source code, and executable software. Deliverables are a kind of artifact. Workers are a role concept for people and resources, having a responsibility relation with artifacts.

SPEM [OMG08] distinguishes between the method content and the process of a systems and software engineering method. The core concepts defined in SPEM [OMG08] for the method content are: work product, task, role, tool. The core concepts of the process are: activity, milestone, work product use, task use, and role use. Processes are defined in a general hierarchical breakdown structure with inner nodes being activities. The workflow is defined with temporal relationships, called work sequences. Work product use, task use, and role use are used to make reference to the corresponding method content elements.

ISO 24744 [ISO07] distinguishes between ten key concepts. Five of them are assigned to the method domain: language, notation, constraint, guideline, and outcome. The other five are assigned to the endeavor domain: stage, work unit, work product, model unit, and producer. Action is used to relate tasks to work products.

Stage is further specialized in instantaneous stage (e.g. milestone) and stage with duration (e.g. time cycle, phase, build). Specializations of work unit are task, technique, and process. Subtypes of work product are composite work product, software item, hardware item, document, and model. The elements of models are captured by model unit. Role, person, team as well as tool are subtypes of producer.

Guidelines can be associated with any methodology element. The observable results of performing any kind of work unit are given by the class `Outcome`. Constraints are aggregated by action kinds and are specialized in preconditions and post-conditions. Notation is associated with both document kind and language. The relation with document kind denotes that a document kind uses one or more notations. The association between notation and language states that multiple notations can depict a language and, vice versa, a notation can depict multiple languages. Language aggregates model unit kind to denote that any model unit kind is always defined in the context of at least one language, while a language can be the context for one or more model unit kinds. A direct association between model kind and language allows a method engineer to express which language is used for one or model kinds.

From this brief discussion of concepts provided in different meta-models or employed in software engineering methods one can already see the ontological diversity of the approaches. In an attempt to bring them closer together we have contrasted a selected set of core concepts of the different approaches in Table 2. In the last column we have

added the matching concepts of our approach that is named MetaME – meta-method for method engineering. They will be further discussed in Section 11.2.

Table 2: Comparison of core concepts for software engineering methods

SPEM	ISO24744	RUP	MetaME
	Stage	Discipline	Domain Discipline
			Concept
Work Product Definition, Work Product Use	Work Product (Model, SoftwareItem, HardwareItem, Document), ModelUnit	Artifact (Model, ModelElement, Document, SourceCode, Executable)	Artifact
Work Definition	WorkUnit	Work	Work
Activity	Process	Workflow Activity	Process Activity
Task Definition, Task Use	Task	Task	Task
Step	Action	ActivityStep	ActionStep
			Transformation
(Phase, Iteration) as Kind	TimeCycle, Phase, Build	Lifecycle, Phase, Iteration	Phase
Milestone	Milestone	Milestone	Milestone
Role Definition, Role Use	Role	Worker	Role
Tool Definition	Tool		Tool Utility
	Language, Notation		Notation
Guidance	Guildeline		Guidance
	Technique		Technique
	Constraint		Constraint

Comparable meta-models can also be derived from existing software engineering methods, and of course, there exist many more of such meta-models for software engineering methods, like those mentioned in Section 9. From the literature and my own project experience, I have derived the meta-model for software engineering methods that will be presented in Section 11.2. Some fundamentals of modeling and meta-modeling will be discussed in the next section.

10.2 Models and Meta-Models

A promising approach towards the systematic and structured development of software engineering methods is the use of meta-modeling techniques for specifying the software engineering methods.

A *model* is, according to scientific theory, a representation of a natural or artificial original that focuses on those characteristics and properties of the original that are relevant for the given purpose of modeling, and abstracts from irrelevant properties. The purpose depends on both the creator and user of the model, and the intended use of the model. In an engineering process, models are used for specification, documentation, and communication. They are themselves objects of processing and transformation, and are

a foundation for decision making, analysis, validation, verification, and testing. Models can be built upfront or retrospective in terms of forward engineering or reverse engineering, respectively.

A *meta-model* is a model of a model. *Meta-modeling* is, according to [GH08], "the act and science of creating meta-models, which are a qualified variant of models." The specialty of a meta-model is that the information it represents is itself a model. The meta-model's concern is the modeling itself. In the domain of method engineering, the meta-model is a model of a software engineering method.

In object-oriented meta-modeling, a model conforms to its meta-model in the way that it is an instance of the meta-model. A software engineering method can then be understood as a model that is an instance of this meta-model. The meta-model together with the definition of the semantics of software engineering concepts that are contained in the meta-model define an ontology for software engineering methods.

Meta-models are commonly used for defining modeling languages. However, they may also be used for defining in a wider sense the process of modeling (compare [Str98]). Meta-modeling is already widely used for defining software modeling languages as well as models of software development methods, e.g. in the case of UML [OMG09a, OMG09b] and SPEM [OMG08] or ISO 24744 [ISO07], respectively. UML is a standard language for modeling software systems. SPEM and ISO 24744 are languages for describing software development methods and process models, i.e., meta-models for method descriptions.

For the *domain of method engineering*, we adopt the definition from [GH08]: A meta-model is a domain-specific language that is intended to represent software development methods. ISO 24744 defines the meta-model in the context of method engineering to be the "specification of the concepts, relationships and rules that are used to define a methodology" [ISO07]. (Note: methodology is synonymously used to method here, denoting the "specification of the process to follow together with the work products to be used and generated, plus the consideration of the people and tools involved" [ISO07].)

The OMG has defined a four-layer meta-model reference architecture in the Meta Object Facility (MOF) [OMG06] that builds on the concepts of object-orientation and is commonly used in meta-modeling (see Figure 30).

According to this meta-model hierarchy, we can characterize the levels for the domain of method engineering as follows:

M0 (Runtime layer) – M0 denotes the lowest level of the MOF 4-layer meta-model hierarchy. In this layer, objects of the real world are denoted that exist at execution time of the modeled system. More generally, M0 represents the area of concern (or domain), which may be business, software engineering, or method engineering. In the domain of method engineering these are the concrete objects (i.e., software artifacts) that are produced or modified during the lifecycle of a concrete software engineering endeavor.

M1 (Model layer) – M1 is the layer where user models are located. Reality is modeled in a modeling language (defined by M2), such that elements of M0 are instances of

elements in M1. In the domain of method engineering, the model of the method is allocated on this level. The method engineer acts as the modeler.

M2 (Meta-model layer) – M2 is the layer where meta-modeling takes place. It contains meta-models (models of models) such as the UML meta-model or SPEM which define modeling languages for describing the user models of layer M1. Elements of user models from M1 are then instances of meta-model elements of layer M2. This level holds the meta-method's model in the domain of method engineering.

M3 (Meta-meta-model layer) – M3 is the highest level of the 4-layer meta-model hierarchy. Meta-meta-models are defined at this layer. They are used to describe the meta-models on layer M2. In the MOF hierarchy, the Meta Object Facility itself is defined on this level. Defining method engineering within an object-oriented meta-model hierarchy, we use MOF for the domain of method engineering on this level as well.

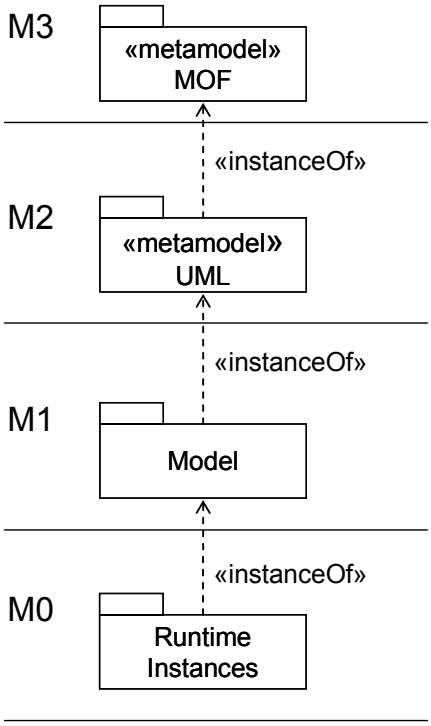


Figure 30: General 4-layer MOF meta-model hierarchy

Based on this meta-model hierarchy and the concepts that can be derived from the characterization of software engineering methods, we define the meta-model of our meta-method (M2). It contains the meta-classes for the important concepts that are required to model a software engineering method. We will build on this four-layer meta-modeling architecture as the guiding principle in the definition of our meta-method's architecture. This will be explained in Section 11.1.

10.3 Method Engineering

After having introduced the basic ideas of meta-modeling in the previous section, we now continue with the topic "method engineering", and then show how meta-modeling

has been applied for method engineering in the next section. SPEM and ISO 24744 will be briefly explained as example meta-models in the following sections.

Method engineering has been an active research area in the field of information systems engineering since the early 1990s. Method engineering is concerned with the systematic construction of software development methods [Gut94]. [Hey95] defines method engineering as the systematic and structured process of development, modification and adaptation of methods by describing the components of the method and their relationships. In general, it is the objective of method engineering to formalize the use of methods for systems development [HR10]. More precisely, method engineering can be defined as the *engineering discipline to design, construct and adapt methods, techniques and tools for the development of (information) systems* [Bri96, HR10]. The objective of method engineering is to develop a methodical approach for systems development in a given context (and situation) such as an organization or project.

Method engineering mainly addresses two perspectives: a) the systematic development of methods and b) the enactment and execution of methods. Both aspects may themselves be supported by dedicated tools, such as a method development environment or a workflow engine.

Applying method engineering to the domain of software engineering methods provides a number of advantages:

- method engineering provides a methodological framework and conceptual infrastructure for method knowledge,
- method engineering supports a systematic development of SE methods,
- by providing specific means for method adaptation, methods can be adapted to a particular situation and context of use *(*cf. *situational method engineering*, see [HR10] for a recent survey),
- concepts of method modularization, reuse and configuration [BKPJ07] can be used to assemble methods from methodical building blocks, such as *viewpoint templates* [NFK96], *method fragments* [Bri96], *method chunks* or *method services* [Rol09],
- the meta-models that are used for the definition of methods enable analysis and comparison of methods, even quantitatively, by the use of an accompanying quality model and metrics,
- method engineering can ease reuse and provide means for compositional method development, and method integration,
- method engineering builds a sound basis for tool support, e.g. computer-aided software engineering (CASE) tools that may be built by using Meta-CASE tools.

The product of a method engineering process is a method. In the context of this work, the users of this product are software engineers who develop software-based systems.

The lifecycle of a method is similar to the lifecycle of a software system. We can interpret a method as a conceptual system for system development. Method engineering manages and controls this method lifecycle and may even itself be computer-supported by its own software system, a computer-aided method engineering (CAME) tool [Bri96]. The general overall lifecycle model of method engineering is depicted in Figure 31. Once the domain of discourse has been identified (being software engineering in our

case), the requirements for the method are analyzed. It follows a multi-stage development process. Then the method is deployed, used, and evaluated in order to start another evolution cycle.

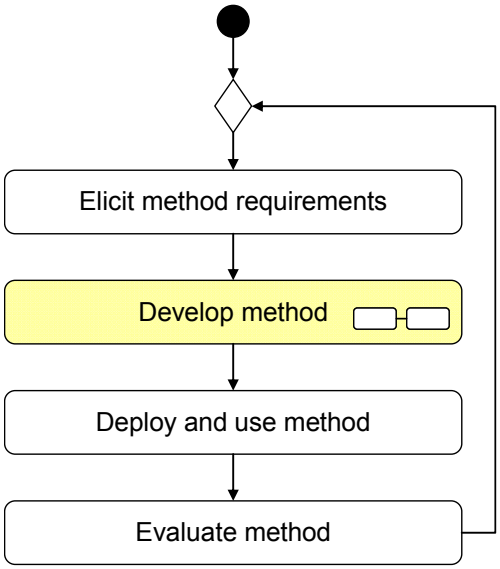


Figure 31: Lifecycle of a software engineering method

10.4 Meta-Modeling for Method Engineering

Meta-modeling has been identified as a promising means for method engineering. Several meta-models have been defined in the literature by different authors, see e.g. [JJM09], [BG08], [GMH05], [HG05]. Two standards also exist that use meta-models for the definition of software development methods: ISO 24744:2007 Software Engineering − Metamodel for Development Methodologies [ISO07] and SPEM, OMG's Software & Systems Process Engineering Meta-Model Specification [OMG08]. The latter provides a meta-model as well as a UML profile for the specification of software development methods. We present SPEM as the most acknowledged meta-model and the ISO 24744 standard as examples in the following.

The engineering of software engineering methods happens within three domains: meta-method engineering, method engineering, and software engineering (see also Section 2.1). Each of them corresponds to a distinct level of abstraction. These levels of abstraction correspond to the layers of the meta-modeling hierarchy depicted in Figure 30. Different tasks of SE method engineering have to be performed in the three domains for producing the required products on the different levels of the meta-model hierarchy. These tasks are performed by dedicated roles according to our meta-method (see Figure 32). The meta-method engineer is responsible for defining the meta-method for method engineering (M2) in the meta-method engineering domain. This meta-method is applied by the method engineer in the method engineering domain in order to develop a concrete software engineering method (M1). The software engineering method is then used by software engineers in the software engineering domain for developing the software system in a real software development project (M0).

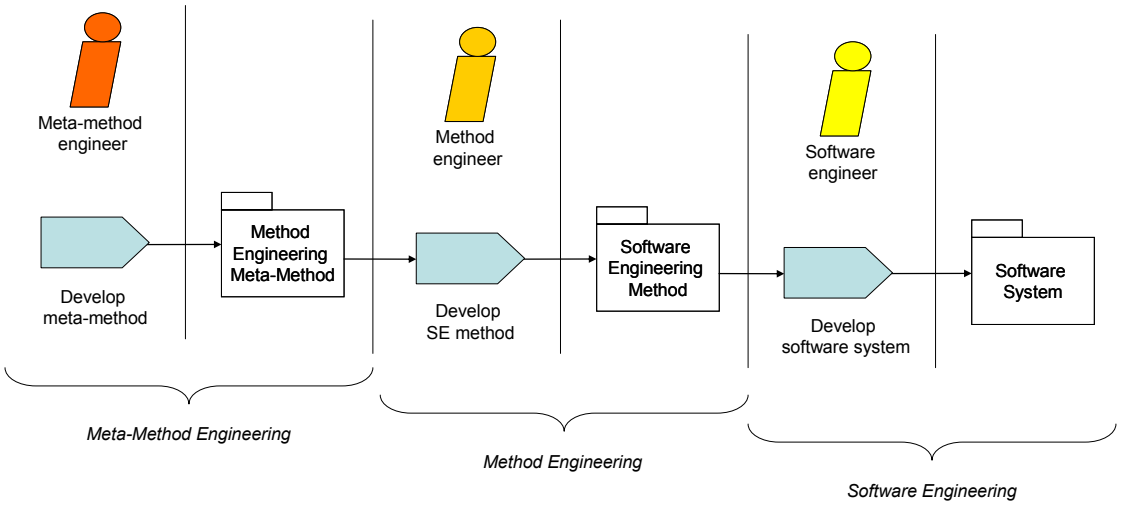


Figure 32: Dedicated roles are responsible for producing the work products on the different layers of the meta-model hierarchy

Gutzwiller proposes a 4-layer model in [Gut94] (depicted in Figure 33) that is similar to our hierarchy and to the MOF meta-model hierarchy (see Figure 29). On level 3, the topmost level, the engineering of development methods is located. This corresponds to the meta-method in our terminology. Level 2 contains the development methods. Information systems are on level 1. Finally, level 0 represent the real world.

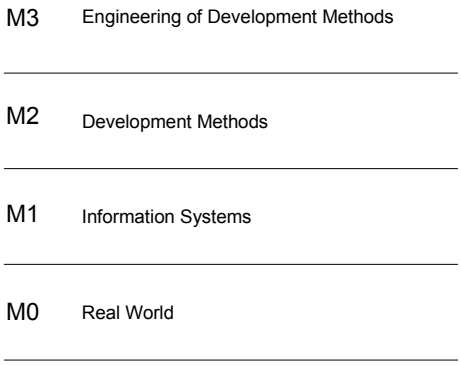


Figure 33: Gutzwiller's 4-layer model of method engineering

10.5 SPEM

The Software & Systems Process Engineering Meta-Model (SPEM) [OMG08] is intended for defining software and system development processes. The OMG characterizes SPEM as "a process engineering meta-model as well as conceptual framework, which can provide the necessary concepts for modeling, documenting, presenting, managing, interchanging, and enacting development methods and processes" [OMG08].

SPEM is a meta-model that is based on MOF and UML. Objective of SPEM is to provide description elements for software development methods that are independent of parameters such as the development paradigm ('method domain' in Section 2.2) being deployed (e.g. agile, architecture-centric or code-centric, test-driven or model-driven

software development), the degree of formalization or the cultural background. The set of language elements is intended to be minimal for this purpose.

SPEM thus really is a meta-model for describing software engineering methods, and not a method engineering method, since it does not contain a method engineering process definition. Furthermore, SPEM has not been intended to be a process modeling language for software development processes nor does it even provide its own behavior modeling mechanisms. It provides neither a concrete process modeling language nor guidance for selecting such a process model. It only provides the interface for docking a complementary behavior modeling mechanisms. Hence, SPEM is just a description language for software engineering methods.

An important principle of SPEM is the separation of the method content and the development process. Method content denotes descriptions of how to achieve particular development goals. Such contents are independent of their use in a specific development process. The method contents are then applied within a process and brought into a temporal order.

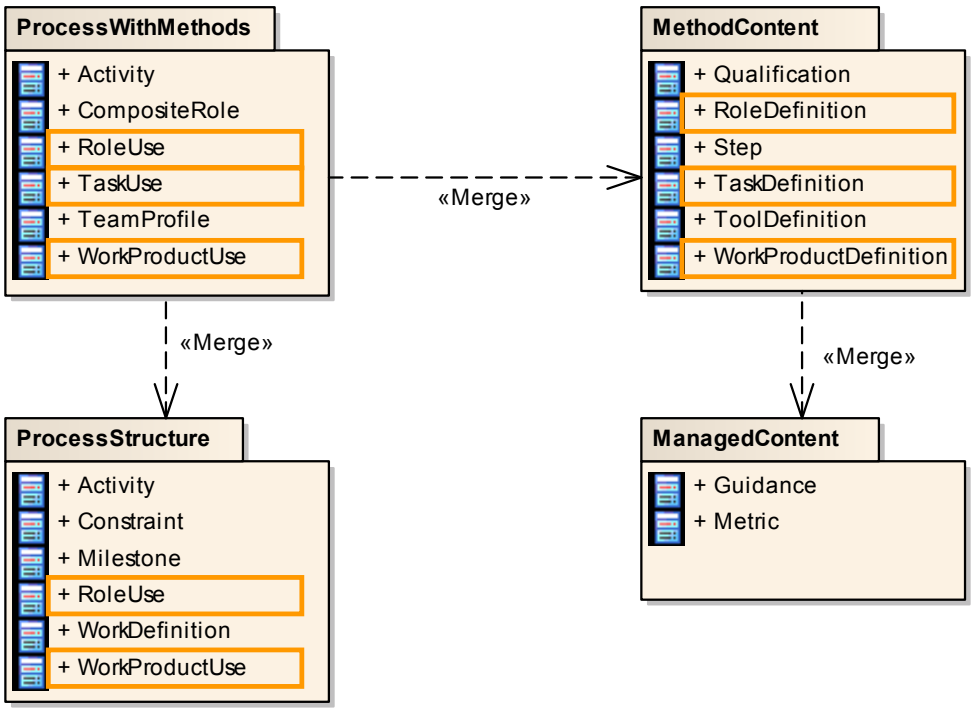


Figure 34: Separation of method content and development process in SPEM

Figure 34 visualizes the separation of method content and process. It shows four meta-model packages of SPEM including the meta-classes for the most relevant concepts. It can be seen that elements such as work products, tasks, and roles are defined as method contents (highlighted on the right-hand side) and then applied in the process part by work product use, task use, and role use (highlighted in the two packages on the left-hand side). What can be seen from this meta-model excerpt is that the package `ProcessWithMethods` actually integrates process structure and method content by its respective merge dependencies. Furthermore, guidance and metric from the package `ManagedContent` are available as general concepts in the merging packages too.

10.6 ISO 24744

ISO standard 24744 Software Engineering – Metamodel for Development Methodologies [ISO07] introduces the Software Engineering Metamodel for Development Methodologies (SEMDM), a comprehensive meta-model that defines software engineering methods (called methodologies in the standard; but methodology is declared to be synonymous with method) based on the concept of powertype. SEMDM is targeted towards the definition of methods in so-called information-based domains, i.e. "areas characterized by their intensive reliance on information management and processing, such as software, business or systems engineering". It considers the integration of the aspects process, modeling and people.

ISO 24744 distinguishes three domains: the meta-model domain, the method domain, and the endeavor domain. Software developers work in the endeavor domain, method engineers in the method domain. The three domains constitute three different areas of expertise that correspond to three different levels of abstraction.

In contrast to the conventional approach of meta-modeling for method engineering, where the meta-model is defined as a model of a modeling language, process or methodology that developers may employ and instantiate to design a method, SEMDM supports a dual-layer modeling approach. Driven by the observation that objects in the method domain are often used as classes by developers to create elements in the endeavor domain during method enactment, the meta-model is constructed as a model of both the method and the endeavor domains. Modeling the method and endeavor domains at the same time leads to pairs of classes in the meta-model that represent the same concept at different levels of classification.

The central infrastructural concept for the definition of ISO 24744 is the *powertype*. A powertype of another type, called the partitioned type, is a type whose instances are subtypes of the partitioned type. Along with the powertype comes the concept of *clabject*: a dual entity that is a class and an object at the same time. With these two infrastructural concepts it is possible to define concepts that belong to the method domain and concepts that belong to the endeavor domain in a common meta-model.

More precisely, the powertype concept is used to form a pattern of two classes in which one of them represents "kinds of" the other. This pattern is called a *powertype pattern*, since the "kind" class is a powertype of the other class, called the partitioned type.

In accordance with the meta-modeling concept, method-level elements must be instances of meta-model-level elements, and elements at the endeavor level must be instances of some element at the method level. This means that elements in the method domain act *at the same time* as objects (since they are instances of meta-model classes) and classes (since endeavor-level elements are instances of them). This class/object duality is named *clabject*. Clabjects have a class facet and an object facet. Within SEMDM, clabjects are used to construct a method from the powertype patterns in the meta-model. A powertype pattern is instantiated into a clabject by making the object facet of the clabject an instance of the powertype class in the powertype pattern, and the class facet of the clabject a subclass of the partitioned type in the powertype pattern.

As stated in Section 10.1, ISO 24744 includes ten key concepts. Five of them are assigned to the method domain: language, notation, constraint, guideline, and outcome. The other five are assigned to the endeavor domain: stage, work unit, work product, model unit, and producer. Action is used to relate tasks to work products.

After having presented the foundations from meta-modeling, method engineering and the use of meta-modeling in method engineering – using SPEM and ISO 24744 as an example – we will now present our meta-method for the definition of software engineering methods in the next section.

11 A Meta-Method for Method Engineering of Software Engineering Methods

Based on the foundations of method engineering and meta-modeling, I have developed MetaME, a meta-method for the engineering of software engineering methods. I deploy a four-layer meta-model architecture in order to define the meta-method. The meta-method covers the *product* of the method engineering process, i.e., the method description, and the *process* that is used to build a software engineering method. We first look at the meta-model architecture that we use for integrating the product models and the process models across the different meta-modeling layers in Section 11.1. We then describe the respective meta-models on the meta-modeling layer M2 and their integration (Sections 11.2 to 11.4). Sections 11.5 and 11.6 exemplify the artifact model and the process model on the method level M1 that instantiate the meta-model from M2. Finally, the use of transformation rules in the process model is shown in Section 11.7.

11.1 Meta-Model Architecture of the Meta-Method

We build on meta-modeling in the definition of our meta-method for method engineering. However, we have discovered that simply employing object-oriented meta-modeling in the sense of the Meta Object Facility has some shortcomings (see also Section 9). MOF is restricted to defining the structure (abstract syntax) of a modeling language. It does not comprise any means for modeling behavior. Furthermore, it only allows an object-oriented type-instance relationship between classes and objects of directly adjacent meta-modeling layers.

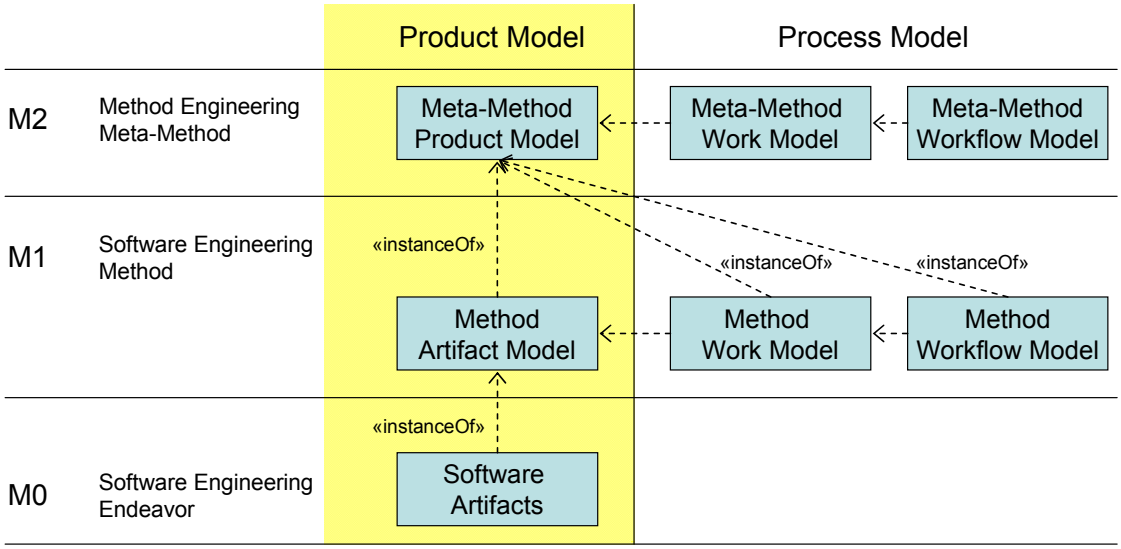


Figure 35: A method consists of a product model and a process model; this applies to the meta-method on level M2 as well as to the software engineering methods on level M1

To define a method, we combine the method's product model (shaded in Figure 35 and Figure 36) with its process model. The process model is composed of a work model that defines work elements such as activities and tasks and a workflow model that defines the temporal ordering of activities. We apply this method pattern on the meta-method level and the method level as shown in Figure 35.

However, while the meta-method process model (M2) must be an instance of a process meta-model (M3) to have execution semantics (see Figure 36), all parts of the method (M1) are defined as instances of the meta-method product model (M2), since the complete method is the product of the method engineering process. Yet, the method process model (M1) must also be an instance of the process meta-model (M3), since it is a process model itself. Yet, this instantiation relationship skips the M2 level and thus is not compliant with strict meta-modeling as described in [AK01].

We solve this problem by bootstrapping the process meta-model into the meta-method product model with a «merge» relationship (see [OMG06]), like this was done for MOF and UML too. Thereby, we also convert between ontological and linguistic meta-modeling, since the meta-method product model on M2 as well defines the method modeling language. The method is engineered and modeled by instantiating the meta-method product model (M2) and enacting the meta-method's process model. This relation is represented by the dependency of type «producedBy» between the method and the instance of the meta-method process model (in level M1). The same pattern applies in the M0 level for the production of the development project's software artifacts.

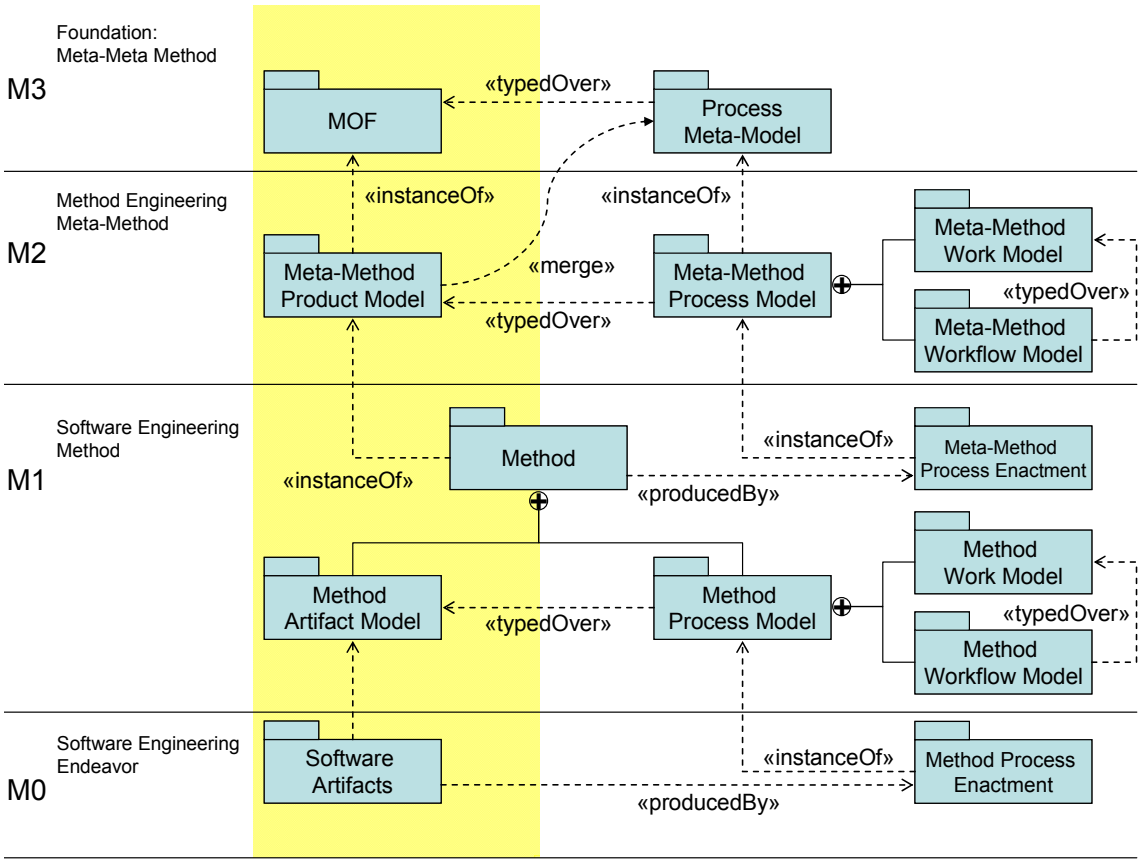


Figure 36: Applying the meta-modeling approach for the engineering of methods

Figure 37 shows a focused view of the product model dimension across the layered architecture. It can be seen, that we distinguish between the method engineering domain (left-hand side) and the software engineering domain (right-hand side). The former is concerned with the engineering of software engineering methods (and software engineering endeavors that are executed according to the software engineering

methods), the latter with the engineering of software systems. Nevertheless, these two hierarchies are interrelated. First, both hierarchies are founded in the Meta Object Facility in their respective M3 layers. Secondly, the software engineering methods that are developed in the method engineering domain are applied in the software engineering domain. For example, the product model of the software engineering method (M1 of method engineering domain) contains an artifact model in which the artifact types of the method such as `Task` and `Class` are defined. Instances of these artifact types are created as the concrete objects in a software development project (i.e., software engineering endeavor). Additionally, these artifact types are transferred to the software engineering domain where they are deployed in the software system meta-model (M2 of software engineering domain) as the elements of a modeling language for the software system model (M1 of software engineering domain). Figure 37 shows an example. Such a software system model is produced in a concrete software engineering endeavor (M0 of method engineering domain) which in turn instantiates the software engineering method. As a consequence, the two domains coincide (with a switch of layers) and must be either integrated or coupled.

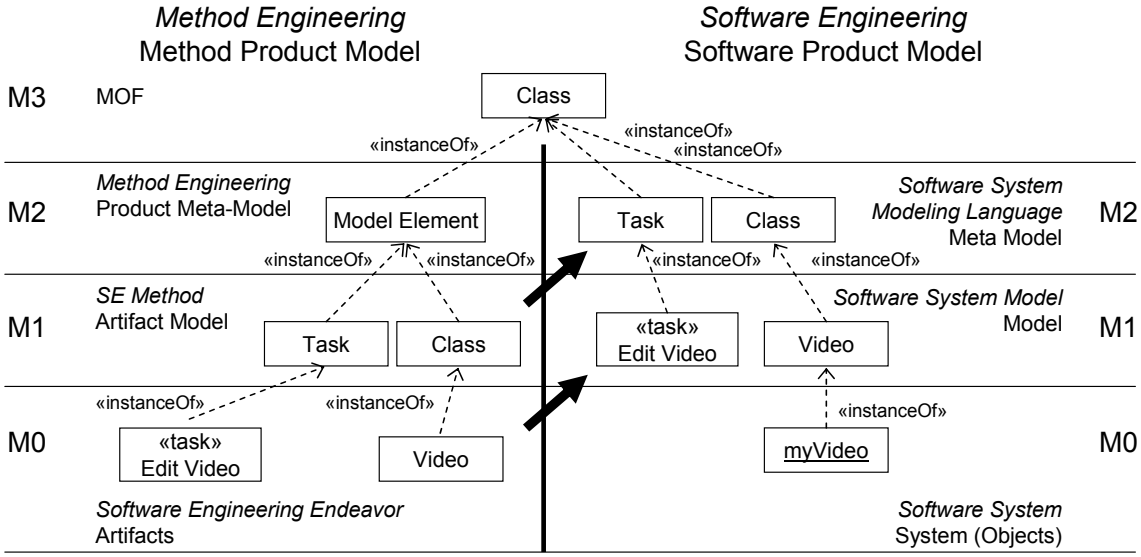


Figure 37: Example for the relationship between the artifact model of a software engineering method (M1, left) and the meta-model of a modeling language (e.g. UML) used in software development (M2, right)

Based on this integrated meta-modeling architecture of product and process models, we describe in the following which constituents make up the meta-method in the different parts of the meta-model hierarchy. We start on the meta-method level M2 (Sects. 11.2 to 11.4) and will then exemplify its instantiation on the method level M1 in the remaining sub-sections of Section 11.

11.2 Method Engineering Meta-Method: Product Model

The meta-method for method-engineering on layer M2 of the method engineering domain is a method itself. Therefore, all relevant aspects for defining a method need to be defined for the meta-method as well: product model and process model. The product model (see Figure 38) of the meta-method defines the fundamental types of elements within software engineering methods. They can basically be boot-strapped to the

method engineering meta-method as well. In addition, I will briefly explain the process of how to develop a software engineering method according to the product model in the next section.

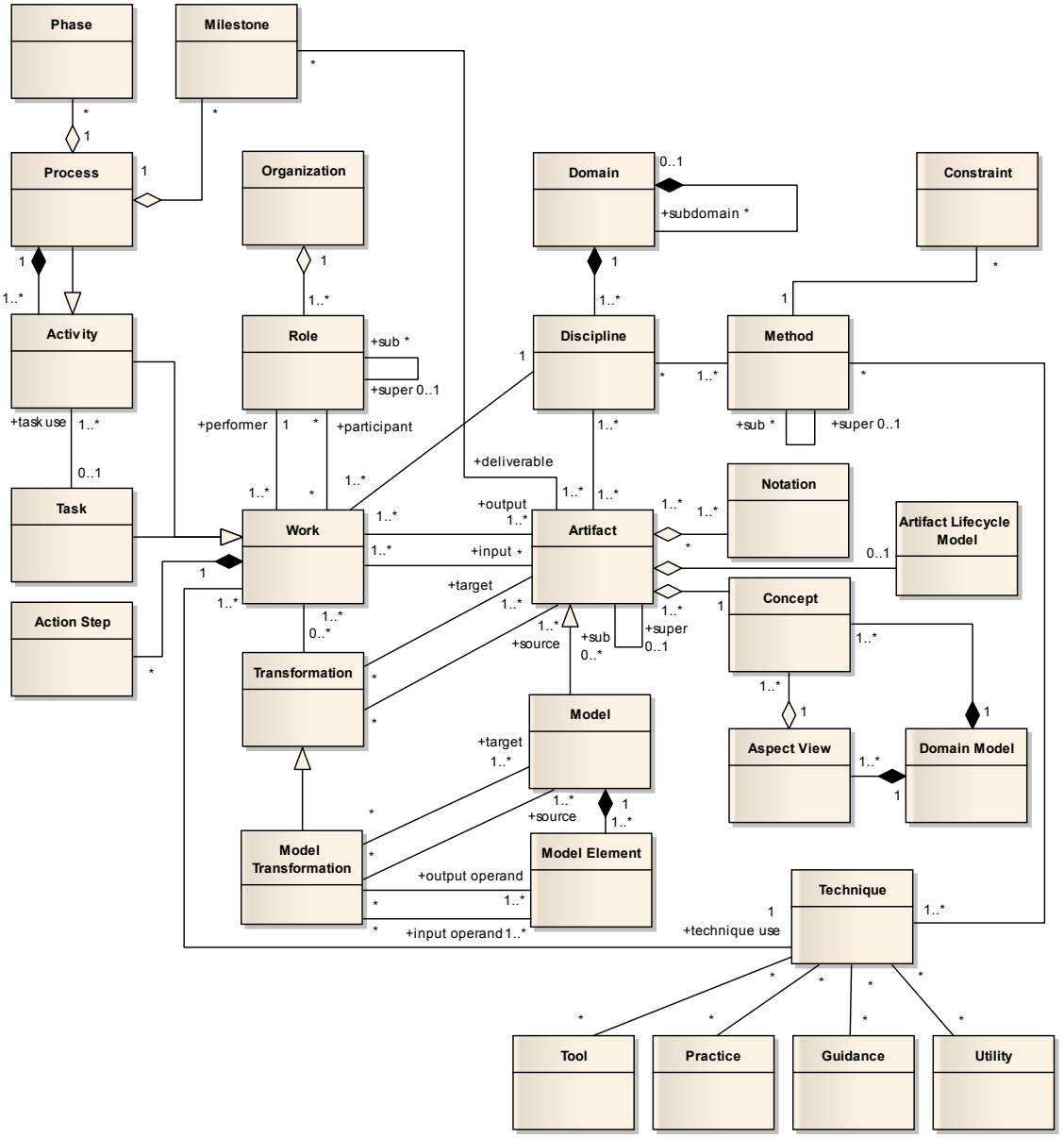


Figure 38: Core domain concepts of software engineering methods defined in the meta-model of our meta-method

Figure 38 depicts the core concepts that are required for describing a software engineering method. We have graphically structured the meta-class diagram according to the main aspects that have to be covered (see Section 10.1). In the upper right, one can find the method content defining the structure of the method (domain, discipline, artifact, concept (semantics), notation (syntax)), and the general concept of constraint. On the left-hand side, the process dimension is covered by the concepts process, activity, task, action step, role, transformation, phase, milestone). At the bottom end, techniques, tools, and practices of the methods that provide guidance on how to

accomplish the tasks and produce the work products are depicted (technique, tool, utility, guidance, practice).

Domain captures the engineering or application domain for which a method is to be specified. Domains can be further decomposed into a set of disciplines. Artifact types are assigned to disciplines in which they are created or used. Each artifact type combines a concept and a set of notation elements. The lifecycle of an artifact can be modeled with an artifact lifecycle model, e.g. a state model. Techniques are associated with methods to which they belong and/or to activities and tasks in which they are used. Tools, utilities, guidance and practices are assigned to techniques.

Models are an important concept in model-based and model-driven development paradigms. To account for that, we introduced the class `Model` as a specialization of the class `Artifact` in our meta-method's artifact model. As a sub-class of `Artifact`, it refers to a model including its syntactic representation in a modeling language. From the perspectives of semantic formalization and semantic model transformations, we could as well sub-type the class `Concept`; but to also account for the notational properties of models and, furthermore, syntactic model transformations, we choose to use the shown inheritance relationship. Models contain model elements, as indicated by the composition relationship between the classes `Model` and `Model Element`. This allows method engineers to define model types and their element types directly as they commonly define artifact types in their methods.

In the model-driven development paradigm, *model transformations* play a prominent role. They should therefore be considered as first-class citizens of any model-driven development method. We account for this by specializing the class `Transformation` in the meta-model into the class `Model Transformation` for model-driven development methods. `Model Transformation` has source and target relations to class `Model`. A transformation rule then operates on the model elements of the related models. The classes `Model`, `Model Element` and `Model Transformation` have been added specifically for the model-based development paradigm and thus extend the meta-class model of [ES10].

Work elements are specialized into tasks and activities. They can be further decomposed into atomic action steps. Work is also related with role to indicate its responsible performer as well as participants. Roles can be part of organizations. Transformations are associated with work elements as well, but also relate to the artifacts they use (source) or produce (target).

Processes are hierarchically composed from activities, can be organized in phases for capturing the temporal domain, and can also include milestones. Each milestone is related to the corresponding artifacts that need to be finished or be available in a specified status to reach the milestone).

11.3 Method Engineering Meta-Method: Process Model

In Figure 39, the fundamental workflow of the meta-method for developing software engineering methods is defined. This workflow model belongs to the meta-method process model (M2) in Figure 36. It refines the composite activity "Develop Method" in the method lifecycle process depicted in Figure 31. The numbers in Figure 39

corresponds to the phases of the meta-method process in order to produce a software engineering method by instantiating the meta-method's product model. The process model is thus typed over the product model. Steps 1 to 5 correspond to the steps that we have described in [ESS08]. The first step is of foundational character:

0. **Define domain and disciplines**: The domain is software engineering methods in our case, and disciplines are used to further structure the software engineering method into areas of concern, such as the disciplines of the Unified Software Development Process [JBR99] (compare Figure 39).

In [ESS08] we have proposed a meta-method for defining software development methods based on domain models of software engineering concepts and artifact types. While the development of the domain models (steps 1 to 3, see [ESS08] for details) is the focus of that article, the definition of the process and the assignment of tools on top of these models are only sketched. The meta-method in [ESS08] has been applied, analyzed and extended in [Sta09]. I have revised those five development steps in the course of defining the meta-method MetaME.

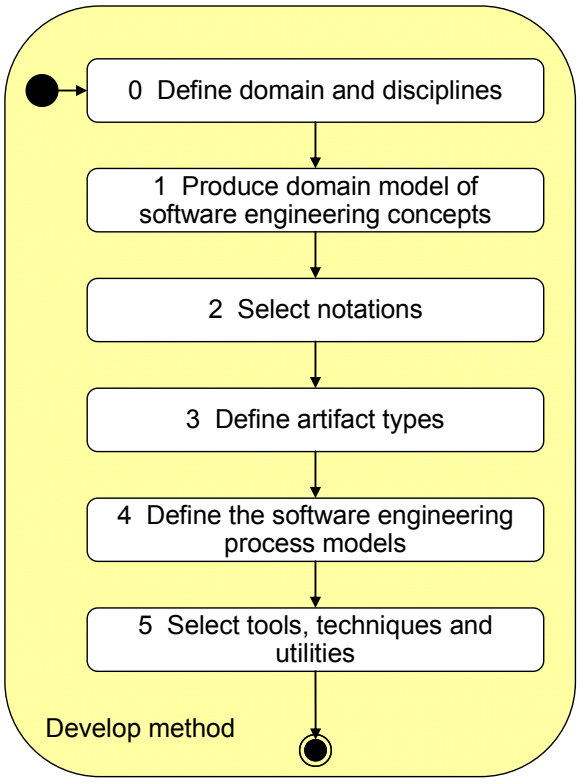


Figure 39: The fundamental workflow of the meta-method

1. **Produce domain model of software engineering concepts**: In the sense of a product model, the domain model of software engineering (SE) concepts is set up and organized according to the identified disciplines (in the form of packages that may be hierarchically nested). Such disciplines may as well correspond to levels of abstraction (requirements, analysis, design, etc.) or views (partial models) of the system (requirements model, analysis model, design model, etc.) Core activities are the definition of SE concepts and assigning them to disciplines. The concepts can be further classified according to the aspect they model, for

example, structure, behavior, or context. Concepts can be engineering or management related.
Relationships between concepts are added such as composition and aggregation relationships, dependencies, and associations. Refinement is an important type of relationship for describing forward engineering methods: a set of concept instances on a lower level of abstraction (partially) refines a set of concept instances on a higher level of abstraction. For example, a business use case can be refined by a system use case, and the system use case can be refined by a set of activities. Another example is a conceptual component in a system specification that is refined into a set of logical components within a software design. The class-model representation of the meta-model is accompanied by a glossary that contains an entry for each meta-model class. Each entry holds a description of the semantics, purpose and properties of the concept and relationships to other concepts.

2. **Select notations:** In order to represent the software engineering concepts appropriately, notations for their representation are required. Languages, together with possible sub-languages (e.g., UML diagram types) and language elements must be identified (either by newly defining them or by reusing existing notations) and enumerated as candidates. Among them are typically languages for modeling, documentation and implementation of software engineering concepts.

3. **Define artifact types**: Candidate notations, i.e., languages and language elements that have been identified in step 2 are assigned to SE concepts from step 1 according to the properties of the software engineering concepts that need to be expressed. While the domain model of software engineering concepts can be interpreted as the semantic domain of the software engineering method, languages define the syntax for denoting them (notation particularly refers to the concrete syntax). The domain model of software engineering artifacts then defines the semantics of the languages by linking language elements (and particularly their syntactic representation) with SE concepts. Consequently, the given semantics of the proposed candidate notations must be conformant with the semantics imposed by the composition of step 3, and the semantics of each language element shall still be unambiguous. Composition hierarchies in the domain models of SE concepts and artifacts must be compatible.

4. **Define the software engineering process models**: The definition of the software engineering process reifies the definition of a roadmap through the network of development artifacts. Tasks and activities are defined and ordered into workflows that produce the required artifacts in the specified order. Sequential, parallel, iterative, incremental, evolutionary and agile development processes shall be supported. The process model is composed from work models and workflow models.
We need to define activities for accomplishing tasks of software engineering in a software development project and the process structure, comparable to the work breakdown structure in [OMG08]. The process structure contains activities, milestones and control-flow elements. The definition of tasks and activities can be extended by object flows of used and produced work items (of the given artifact types) and roles that are responsible for executing activities or participate in the activities' execution (the responsibility for an artifact is a structural issue, that is modeled separately; a role model is also provided separately). For defining the process, the following sub-activities need to be performed, which will be explained in more detail with the M1-level examples in Section 11.6 and Section 11.7:

(a) identify tasks and activities,
(b) define the process structure (workflow) including processes, activities, and milestones,
(c) specify work element structures for tasks, activities and possibly workflow patterns including roles and work products,
(d) define the temporal course using phases and possibly other kinds of time period concepts such as iterations, releases, builds,
(e) describe transformations and constraints.

5. **Select tools, techniques and utilities**: The selection of tools, techniques, and utilities as well as the provision of concepts of use for these tools are required for guiding and simplifying the software engineering work and producing the required software artifacts. Tools, utilities, and guidance are assigned to techniques. Techniques are in turn related with methods to which they belong and/or to activities and tasks in which they are used. Guidance on how to proceed in an activity or task to produce the artifacts of a particular type shall be explicitly provided, e.g. in the form of guidelines, good and best practices, whitepapers, checklists, templates, examples, or roadmaps. However, even the assignment of languages to software engineering concepts in step 3 can be interpreted as partly associating a technique for the development artifact. Both languages and tools typically have implications on how to produce an artifact. Eventually, tools and utilities are thus related to the activities of the software engineering process model as well. By this, it is shown which activities are supported by tools and utilities and, in turn, which of them are to be used when accomplishing the task of the activity.

Enacting this process in the M1 level of the method engineering domain, we systematically create a software engineering method as an instance of the meta-methods M2 product meta-model.

11.4 Integrating the Views of the Meta-Method

As can be seen from the description of the product and process models of the meta-method in the last two sections, a number of consistency issues arise from the different views on the software engineering method's elements:

- consistency of artifacts as defined in the artifact model and their use as work products in the process,
- conformance of hierarchical composition structures of the domain model of software engineering concepts and the artifact model,
- consistent composition of process structures from activities, obeying all given constraints such as the hierarchical composition of activities and flow relationships (predecessor) between activities, or the linking of roles and work products,
- consistency between work product use in activities and processes and the artifact lifecycle model.

Such issues need to be resolved in the method engineering domain when a software engineering method is defined. In addition to the method's conformance with the meta-

methods product meta-model, it is also possible to define additional constraints for software engineering methods as instances of the `Constraint` meta-model class.

Yet, a common meta-model for software engineering methods does not only provide a common language for describing software engineering methods; it can also be used as a standardized and unified reference model for software engineering methods. Different methods can be analyzed, compared and exchanged on this basis.

In addition to the definition of activities and process structures (e.g. depicted as UML activity diagrams holding the different process elements and specifying the workflow of activities), we deploy collaborations in the work model defining the effect of work elements, i.e., tasks and activities, on the artifact structure (which can be interpreted as graph transformation rules that are typed over the artifact model). Such models of the dynamics of a method can for instance be used for reasoning or analyzing certain properties of a process. These transformation rules are explained in more detail in Section 11.7.

Following this general discussion of the meta-method level, we will present the artifact model (result of step 3) and the software process model (step 4) in more detail in the following two sections. They represent instances of the meta-method product model (M2) on the method level (M1).

11.5 Defining the Artifact Model of the Software Engineering Method

The result of step 3 of our meta-method is a domain model of software engineering artifacts. The objective of this model is to establish a common understanding of the software artifacts that are to be produced in a software development project. This comprises the purpose and meaning of each artifact type and its relevant properties as well as the relationships, associations, dependencies, and generalizations between artifact types. The artifact model is a type model that is used in the role of a meta-model in the software engineering domain and thus instantiated for describing a software system in a software development project (compare Figure 37). It defines the set of software engineering artifacts that are produced, edited, and used throughout a software development project as part of the software engineering method. It thus constitutes the product model of the software engineering method (M1 in the method engineering domain) and acts as the backbone of a family of methods where it can be combined with dedicated process models, languages and tools.

Figure 40 shows as an example an excerpt from the artifact model for system specifications in the disciplines business modeling and analysis that has been developed together with an industrial partner.

For each artifact type, we can specify an artifact lifecycle model (state model, see Figure 38) which defines the lifecycle of artifacts that are instances of this artifact type.

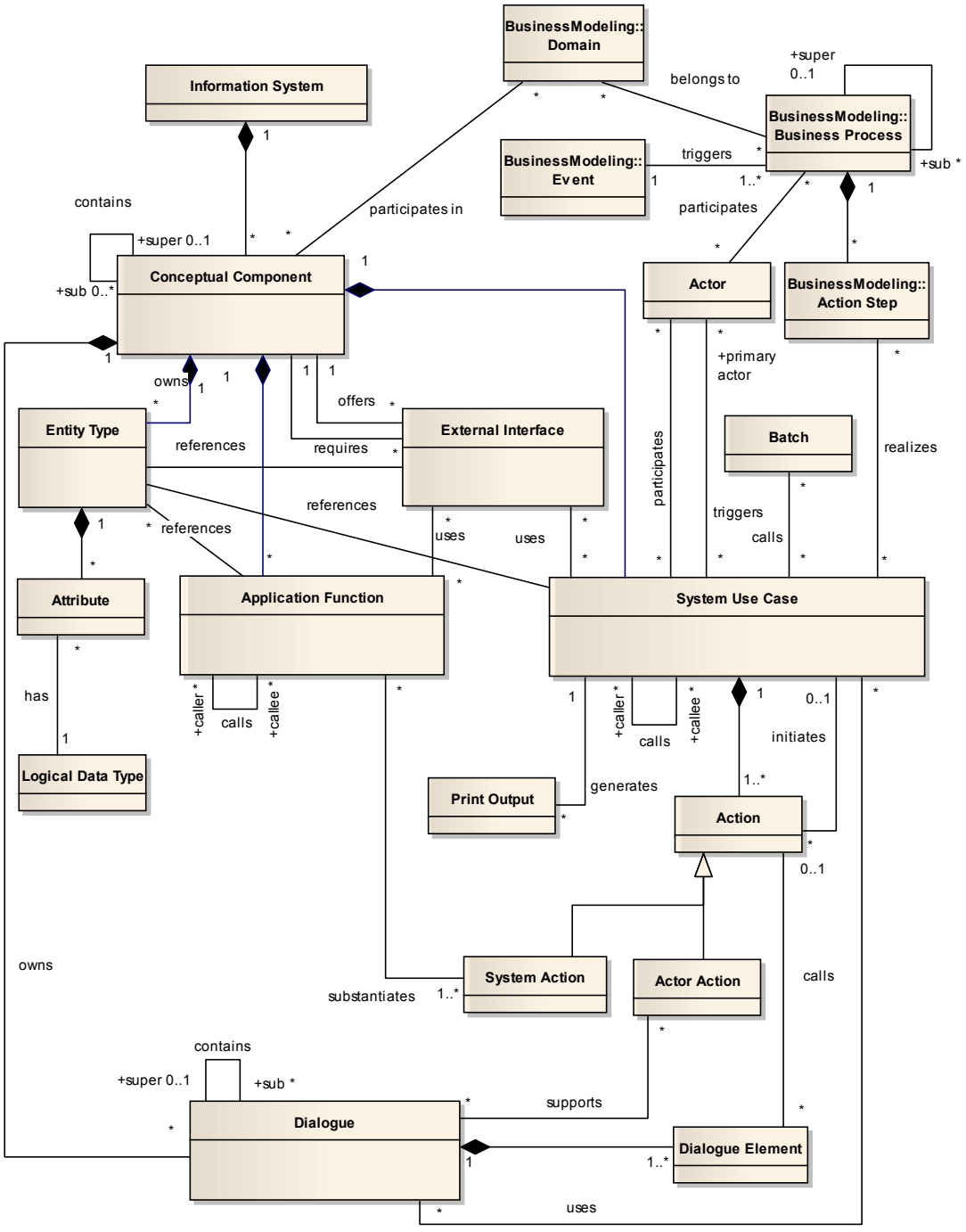


Figure 40: Excerpt from the artifact model for system specification in the discipline analysis with reference to elements from the discipline business modeling

11.6 Software Process Modeling in the Software Engineering Method

The workflow structure of the process defines an ordered and hierarchically nested structure of activities and milestones. Ordering relations, i.e., direct predecessor dependencies between elements, are explicitly specified. They may be marked e.g. as dependencies of the appropriate temporal relationship kind (such as typical interval relations startToStart, startToFinish, finishToStart, finishToFinish, compare [OMG08]).

Transitive dependencies need to be computed for scheduling the activities. Other properties of the work elements that are relevant for the ordering and execution may be indicated by meta-attributes, such as "hasMultipleOccurences", „isOptional", „isRepeatable", „isOngoing" and „isEventDriven" (compare [OMG08]).

The flow of activities can also be represented in UML activity diagrams or other process languages such as BPMN, Petri Nets, etc. The flow diagrams can contain activity elements (as actions) as well as control flow patterns and nodes such as sequential, alternative and parallel execution, conditional flows and iteration. Thus it is possible to specify different kinds of software engineering processes.

Figure 41 shows an example of an activity diagram depicting the flow of high-level activities according to the disciplines of the Unified Software Development Process [JBR99]. In each iteration of the process, the engineering activities on the left-hand side are executed sequentially. The three activities corresponding to the supporting disciplines on the right-hand side run across the whole development lifecycle.

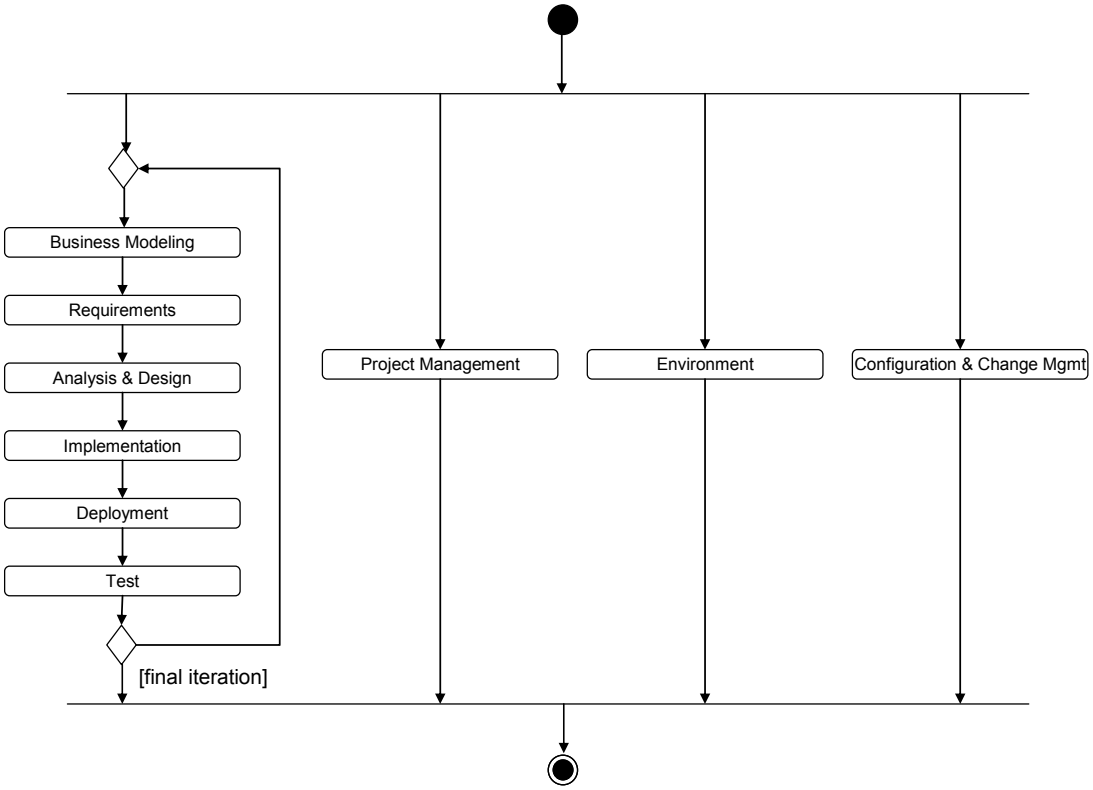


Figure 41: Flow of high-level activities for the Unified Software Development Process

Constraints may be defined that restrict the possible flow of activity. For example, we can use temporal expressions to define that each activity for specifying a use case must eventually be followed by an activity specifying a test case. This may also be depicted graphically, e.g. by the notation used in Figure 42.

Roles and work products (artifacts) shall be related to activities and milestones. Therefore, they are to be included in the work model. (Within this task, one may finally select from alternative artifact types, by defining the references to the artifact types of the information model.) For activities, we can indicate whether work products are used

as parameters of kind „input/output/inoutput“. For milestones, we can indicate which work products are required results that have to be completed for achieving that milestone (responsibility assignment). For all elements that reference roles and work products, it may be specified whether they are mandatory or optional by a meta-attribute „isOptional“.

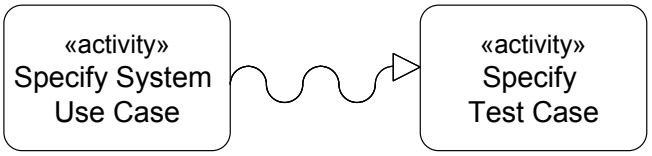


Figure 42: Process constraint depicting that each activity for specifying a use case must eventually be followed by an activity specifying a test case

Activity diagrams with object flows can be used to depict the input and output work items of each activity. The use of work products can be represented as object flows according to their parameter kind: in, out, inout. One can use ObjectNodes, Pins ActivityParameterNodes of UML activity diagrams for this purpose. Roles can be integrated by the use of ActivityPartitions (aka. “swimlanes”) for assigning activity elements to the corresponding roles, representing the relationship between activity and the role being used. Figure 41 shows an example how the activity pattern for structuring the use case-model according to the Unified Software Development Process [JBR99] can be represented. This diagram shows the activity together with the related role and work products as well as the deployment of the tool Enterprise Architect according to step 5 of the meta-method.

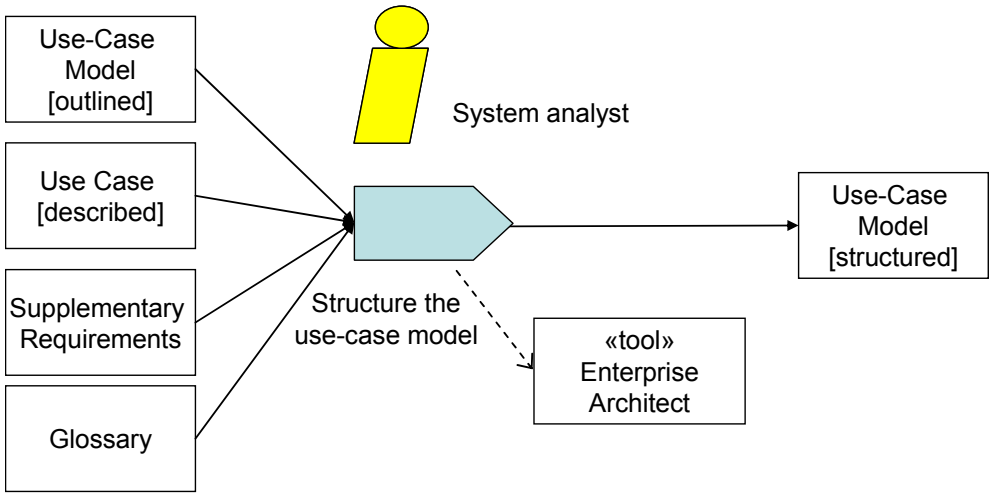


Figure 43: Activity pattern for the activity “Structure the use-case model” adapted from [JBR99]

Consequently, the following relationships are represented in such a model:

- a role is responsible for performing an activity,
- a work product is used or produced by an activity as input, output or input-output parameter,
- a tool shall be employed for accomplishing the activity,

- if a relationship of type performs exists between an activity and a role use and a parameter relationship of kind out or in-out exists between the same activity and a work product use, then there also exists a relationship of type responsible (responsibility assignment) between the work product use and the role use.

11.7 Defining Work of Software Engineering Methods as Transformations

As can be seen from Figure 43, the representation of the effects of work elements on the artifact model can only be expressed in a limited way by using activity diagrams or composite structures. Even if object flows are represented, they can only make reference to the state of individual objects (such as the states "outlined" and "structured" for the use-case model and the state "described" for use case in Figure 43). We therefore include transformations that are depicted as UML collaborations (i.e., the structural part of UML 2 interaction diagrams) that are interpreted as graph transformation rules on the type graph of the artifact model (as introduced in [HS01]).

Figure 44 gives an example of such a transformation rule. It states for the activity "Identify system use cases" that for each occurrence of the pattern on the left-hand side in an instance of the artifact model, the structure on the right-hand side must be produced by the activity. In particular it states that if a business process has an action step that shall be supported by the software system being modeled (property "isManual" = false), then a system use case needs to be included in the system model that realizes this action step and whose primary actor is the same actor who is responsible for executing the action step of the business process. The rule can be interpreted as a visual contract [LSE05] stating pre- and post-conditions of the activity.

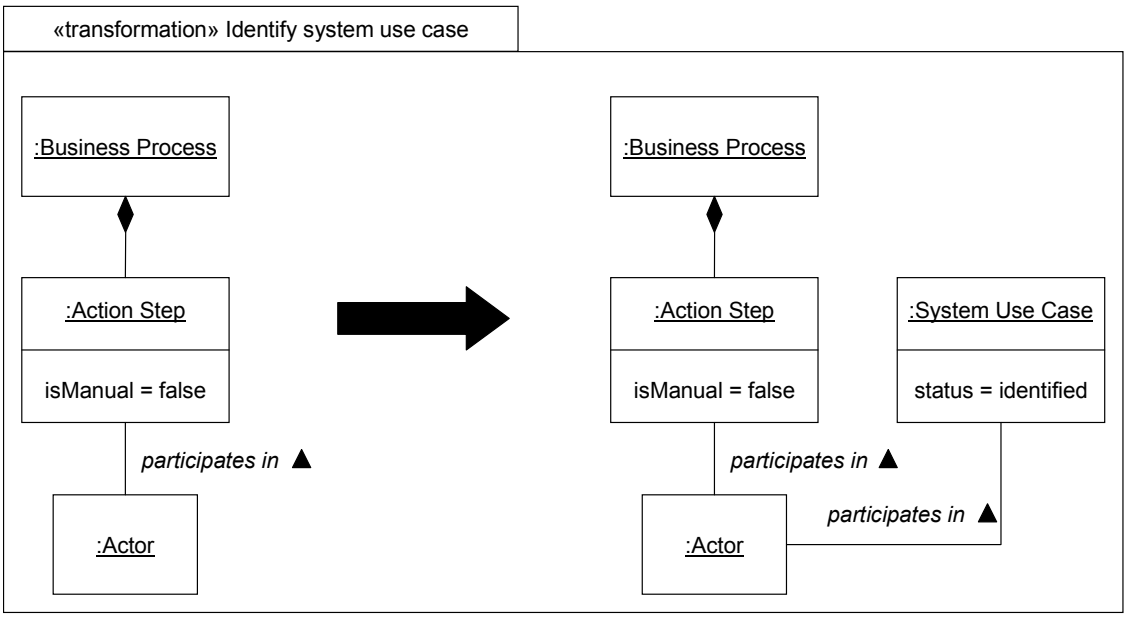


Figure 44: Transformation rule for activity "Identify system use case"

With this example of a transformation rule that specifies the effect of a work element in a method we conclude our tour of how to develop software engineering methods with MetaME. So far we have focused on how to create new software engineering methods,

but we can distinguish between initial development of a software development and its modification. Modification can have different causes such as the need to evolve, tailor, specialize or extend the software engineering method. We will exemplify tailoring scenarios in the next section.

12 Tailoring and Reuse of Software Engineering Methodology

Software engineering methods evolve over time due to changing requirements, new achievements in software engineering – such as new process models and development practices – and experience from software engineering projects. However, each software development project is individual too, at least to some notable degree. This results in at least slightly different requirements for every software development project. Thus, software engineering methods must be tailorable. Such tailoring is a possibility to implement situational method engineering (see Section 9). It happens on the M1 level of our meta-modeling hierarchy. Changes to the meta-method on the M2 are also possible, but they should only appear in a controlled evolution process within the meta-method engineering domain, and are not part of the method engineering domain.

Tailoring can thus be applied to different levels of our approach according to the tailoring scenarios that are required in an organization. We will give examples for the tailoring of methods on the method (M1) and the meta-method (M2) level.

12.1 Tailoring the Software Engineering Method

Changes on the M1 level, where the software engineering method is located, can be manifold and will occur rather often. For example, if a project is of limited size and budget, it may not be appropriate to execute all activities in full and to produce all the work products that are defined in the general method. If a method engineer (or a project leader who is responsible for the tailoring of the software engineering method) wants to change the artifact types, their properties or relationships, she has to adapt the software engineering artifacts model. The consequences of such modifications on the network of artifact types can be directly observed from the artifacts model. For example, if a project decides to specify the software system without use cases and to use a combination of business processes, business rules, dialogue specifications and application functions instead, this will have an impact on numerous artifact types that are typically related to use cases.

Changes to the artifacts model will typically also affect the defined activities in the process dimension, since tasks and activities use artifacts from the artifacts model as their input and output objects. If use cases are no longer produced, the work elements producing use cases are no longer required. Work elements that typically depend on the provision of use cases as input need to be altered for using the other supplied artifacts that are available as work products. Maybe even the flow of activity must be altered due to changes in the parameter object types. For example, if dialogue specifications are to be used in an activity instead of use cases, but dialogues were only specified later in the process so far, the activity of specifying dialogues will have to be moved upfront before the dependent activity.

A number of other changes to the software engineering method are also quite common:

- changes of techniques how to produce a work product,
- change of notations for representing the content of artifacts,

- changes of tools, e.g. due to the wish of a customer, to enable tool chaining with another tool, or to interoperate with development partners using a common tool basis,
- addition of roles and responsibilities, and many more.

The advantage of the formal meta-model based approach of method engineering that we have presented in this work is that its parts and their relationships within and among each other are precisely and explicitly modeled. Thus tailoring can as well be executed and described in a systematic way.

12.2 Tailoring the Meta-Method

The method engineering meta-method on level M2 has been designed to be flexible enough to support a wide range of software engineering methods. However, if the need occurs to define an at least partly new type of development method, or to add concepts that have so far not been recognized, it may be necessary to start the tailoring process on the meta-method level already and not just to alter the software engineering method on the M1 level. This will be the case if elements are needed in the software engineering method that cannot be built by instantiating the meta-classes of the method-engineering meta-method.

For example, let us assume that the current meta-method only includes the concepts of developers as roles that are responsible for certain work products and participate in the execution of activities, possibly representing tasks. From an engineering perspective this may be sufficient. If the software engineering method shall now be extended to include a staffing perspective of project management, however, such a restriction may no longer be acceptable. Especially in distributed offshore development it may be necessary to make assumptions about the skills of the prospective developers on a project team (who may not be known in advance), to specify the required skills, and eventually to match the required and provided skills to actually form the development team. This makes sense for estimating the effort and costs for accomplishing a certain development task or for controlling the expected and achieved quality of the produced artifacts. In this case, the product model of the meta-method for method engineering needs to be extended by a class `Person` with a property for skills. The existing class `Role` also needs to be extended by a property for skills. Such fundamental changes need to be grounded on the M2 level and can then be applied on the M1 level by instantiating the modified elements.

13 Conclusion

In this part of the thesis, I have presented MetaME, a meta-method for method engineering of software engineering methods. It builds on a four layer meta-model hierarchy which combines the two domains method engineering and software engineering. We have described a meta-model as a general product model of method engineering as well as a process model for developing software engineering methods. The process consists of 5+1 steps. Together they cover the product and the process dimension of the meta-method. The most important steps of that process are Steps 3 and 4. They are concerned with defining an artifact model and software process modeling, respectively. In the process model, we also introduce the concept of specifying software engineering tasks as transformation rules that are typed over the artifact model, thus integrating the two dimensions. The issue of tailoring in our meta-method for the engineering of software engineering methods is discussed in the final section.

Although method engineering of software engineering methods is not a new domain, there is still work to be done. The integration of the different aspects and views of such a method is not yet complete. Especially the use of constraints and patterns in a meta-model-based approach still needs to be better understood. Furthermore, we have made a first step towards the integration of structural and behavioral meta-modeling of software engineering methods. This integration needs to be continued and evaluated. The integrated meta-modeling architecture underlying the method engineering and software engineering domains appears to be a qualified basis for this.

References

[AK01] Atkinson, C; Kühne, T.: Processes and products in a multi-level metamodeling architecture. International Journal of Software Engineering and Knowledge Engineering (IJSEKE) 11(6):761–783, 2001.

[Bal98] Balzert, H.: Lehrbuch der Software-Technik: Software-Management, Software-Qualitätssicherung, Unternehmensmodellierung, Spektrum Akademischer Verlag, Heidelberg Berlin 1998.

[BG08] Bollain, M.; Garbajosa, J.: A metamodel for defining development methodologies, In: Filipe, J. et al. (eds): ICSOFT/ENASE 2007, CCIS 22, pp. 414−425. Springer, Berlin Heidelberg 2008.

[BKPJ07] Becker, J.; Knackstedt, R.; Pfeiffer, D.; Janiesch, C.: Configurative method engineering - on the applicability of reference modeling mechanisms in method engineering. In Proc. Americas Conference on Information Systems (AMCIS 2007), paper 56, 2007. http://aisel.aisnet.org/amcis2007/56

[Bri96] Brinkkemper, S.: Method engineering: engineering of information systems development methods and tools. Information and Software Technology 38(4): 275–280, 1996.

[ESS08] Engels, G.; Sauer, S.; Soltenborn, C.: Unternehmensweit verstehen – unternehmensweit entwickeln: Von der Modellierungssprache zur Softwareentwicklungsmethode. Informatik-Spektrum 31(5):451–459, Themenheft Modellierung. Springer, Berlin Heidelberg 2008.

[GH08] Gonzalez-Perez, C.; Henderson-Sellers, B.: Metamodelling for Software Engineering, Wiley & Sons, 2008.

[GMH05] Gonzalez-Perez, C.; McBride, T.; Henderson-Sellers, B.: A metamodel for assessable software development methodologies. Software Quality Journal 13:195–214.

[Gut94] Gutzwiller, T. A.: Das CC RIM-Referenzmodell für den Entwurf von betrieblichen, transaktionsorientierten Informationssystemen, Physica-Verlag, Heidelberg 1994.

[Hey95] Heym, W.: Prozeß- und Methoden-Management für Informationssysteme: Überblick und Referenzmodell, Springer, Berlin Heidelberg 1995.

[HG05] Henderson-Sellers, B.; Gonzalez-Perez, C.: A comparison of four process metamodels and the creation of a new generic standard. Information and Software Technology 47:49–65, 2005.

[HR10] Henderson-Sellers, B., Ralyté, J.: Situational method engineering: state-of-the-art review. Journal of Universal Computer Science 16(3):424–478.

[HS01] Heckel, R.; Sauer, S.: Strengthening UML collaboration diagrams by state transformations. In Proc. Fundamental Approaches to Software Engineering, 4th International Conference, FASE 2001, pp. 109–123. Springer, Berlin Heidelberg 2001.

[IBM07] IBM Corporation: Rational Unified Process. Version 7.0.1, 2007.

[IEEE90] IEEE: Standard Glossary of Software Engineering Terminology, IEEE Standard 610.12, The Institute of Electrical and Electronics Engineers, New York, NY 1990.

[ISO07] ISO: ISO/IEC 24774:2007 Software engineering – metamodel for development methodologies, International Organization for Standardization, Geneva 2007.

[JBR99] Jacobson, I.; Booch, G.; Rumbaugh, J.: The Unified Software Development Process: The complete guide to the Unified Process from the original designers, Addison Wesley, 1999.

[JJM09] Jeusfeld, A., Jarke, M., Mylopoulos, J. (eds): Metamodeling for method engineering, MIT Press, Cambridge, MA 2009.

[LSE05] Lohmann, M.; Sauer, S.; Engels, G.: Executable visual contracts. In Proc. 2005 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC ’05), pp. 63–70. IEEE Computer Society, 2005.

[NFK94] Nuseibeh, B.;, Finkelstein, A.; Kramer, J.: Method engineering for multi-perspective software development. Information and Software Technology 38:267–274, 1994.

[OMG06] Object Management Group: Meta Object Facility (MOF) Core Specification, Version 2.0, 2006. http://www.omg.org/spec/MOF/2.0/PDF/

[OMG08] Object Management Group: Software & Systems Process Engineering Meta-Model Specification, Version 2.0, 2008. http://www.omg.org/specs/

[OMG09a] Object Management Group: OMG Unified Modeling Language (OMG UML), Infrastructure, V2.2, 2009. http://www.omg.org/uml/

[OMG09b] Object Management Group: OMG Unified Modeling Language (OMG UML), Superstructure, V2.2, 2009. http://www.omg.org/uml/

[RBH07] Ralyté, J.; Brinkkemper, S.; Henderson-Sellers, B. (eds.): Situational Method Engineering: Fundamentals and Experiences. Proc. IFIP WG 8.1 Working Conference, Springer, Boston 2007.

[Rol09] Rolland, C.: Method engineering: towards methods as services. Software Process: Improvement and Practice 14:143–164.

[SB02] Schwaber, K.; Beedle, M.: Agile Software Development with Scrum. Prentice Hall, Upper Saddle River 2002.

[SSEB10] Salger, F.; Sauer, S.; Engels, G.; Baumann, A.: Knowledge transfer in global software development – leveraging ontologies, tools, and assessments. In Proc. 5th International Conference on Global Software Engineering (ICGSE ’10), pp. 336–341. IEEE Computer Society, 2010.

[Sta09] Stadtler, D.: Eine generische Methode zur unternehmens- bzw. projektspezifischen Festlegung von Vorgehensmodellen zur Entwicklung von Software, Diplomarbeit, Universität Paderborn, Germany, 2009.

[Str98] Strahringer, S.: Ein sprachbasierter Metamodellbegriff und seine Verallgemeinerung durch das Konzept des Metaisierungsprinzips. In: Pohl, K.; Schürr, A.; Vossen, G. (eds.): Modellierung '98. Volume 9 of CEUR Workshop Proceedings, ISSN 1613-0073, 1998. http://CEUR-WS.org/Vol-9/

[Wie03] Wiegers, K. E.: Software Requirements, Microsoft Press, 2003.

Part III:

Examples of Software Engineering and

Method Engineering Methods

14 Contributed Works and Publications

In the third part of my thesis, I provide a selection of the most important research publications that I contributed to the field of software engineering methods and their systematic development. All but the last (and latest) are examples of (partial) software engineering methods for different purposes. The last one is concerned with the application of the contributed method engineering meta-method (see Part II) to the domain of model-driven development of advanced user interfaces. List entries are ordered according to topics and indexed with their section numbers. Citation markers correspond to Part I.

15 Gregor Engels, Roland Hücking, Stefan Sauer, Annika Wagner: UML Collaboration Diagrams and Their Transformation to Java. In R. France, B. Rumpe (eds.): Proc. UML'99 - The Unified Modeling Language, October 28-30, 1999, Fort Collins, Colorado, USA, pp. 473–488. LNCS 1723, © Springer-Verlag, Berlin Heidelberg 1999. **[EHSW99a]**

16 Reiko Heckel, Stefan Sauer: Strengthening UML Collaboration Diagrams by State Transformations. In H. Hussmann (ed.): Proc. 4th Intl. Conf. Fundamental Approaches to Software Engineering (FASE 2001), April 2001, Genova, Italy, pp. 109–123. LNCS 2029, © Springer-Verlag, Berlin Heidelberg 2001. **[HS01]**

17 Gregor Engels, Jan Hendrik Hausmann, Reiko Heckel, Stefan Sauer: Dynamic Meta Modeling: A Graphical Approach to the Operational Semantics of Behavioral Diagrams in UML. In A. Evans, S. Kent, B. Selic (eds.): Proc. UML 2000, October 2-6, 2000, York, UK, pp. 323–337. LNCS 1939, © Springer-Verlag, Berlin Heidelberg 2000. **[EHHS00]**

18 Jan Hendrik Hausmann, Reiko Heckel, Stefan Sauer: Dynamic Meta Modeling With Time: Specifying the Semantics of Multimedia Sequence Diagrams. Journal of Software and Systems Modeling (SOSYM) 3(3):181–192, 2004. © Springer-Verlag **[HHS04]**

19 Gregor Engels, Stefan Sauer: Object-oriented Modeling of Multimedia Applications. In S.K. Chang (ed.): Handbook of Software Engineering and Knowledge Engineering, vol. 2, pp. 21–53, © World Scientific, Singapore 2002. **[ES02]**

20 Stefan Sauer, Gregor Engels: UML-based Behavior Specification of Interactive Multimedia Applications. In Proc. IEEE Symposia on Human-Centric Computing Languages and Environments (HCC'01), September 2001, Stresa, Italy, pp. 248–255. © IEEE, 2001. **[SE01]**

21 Stefan Sauer, Gregor Engels: Easy Model-Driven Development of Multimedia User Interfaces with GuiBuilder. In C. Stephanidis (ed.): Proc. 4th Intl. Conf. Universal Access in Human-Computer Interaction (UAHCI 2007), Part of HCI International 2007, Beijing, China, July 22-27, 2007, Part II: Universal Access Methods, Techniques and Tools, pp. 537–546. LNCS 4554, © Springer-Verlag, Berlin Heidelberg 2007. **[SE07]**

22 Stefan Sauer: Applying Meta-Modeling for the Definition of Model-Driven Development Methods of Advanced User Interfaces. In: H. Hussmann, G. Meixner, D. Zuehlke (eds.): Model-driven Development of Advanced User Interfaces, pp. 67–86. © Springer-Verlag, Berlin Heidelberg 2011. **[Sau11]**

UML Collaboration Diagrams and Their Transformation to Java

Gregor Engels[1], Roland Hücking[2], Stefan Sauer[1], and Annika Wagner[1]

[1] University of Paderborn, Dept. of Computer Science
D 33095 Paderborn, Germany
{engels,sauer,awa}@uni-paderborn.de
[2] SAP AG, Lo. Dev. PP-PI, Neurottstr. 16
D 69190 Walldorf, Germany
roland.huecking@sap-ag.de

Abstract. UML provides a variety of diagram types for specifying both the structure and the behavior of a system. During the development process, models specified by use of these diagram types have to be transformed into corresponding code. In the past, mainly class diagrams and state diagrams have been considered for an automatic code generation. In this paper, we focus on collaboration diagrams. As an important prerequisite for a consistent transformation into Java code, we first provide methodical guidelines on how to deploy collaboration diagrams to model functional behavior. This understanding yields a refined meta model and forms the base for the definition of a transformation algorithm. The automatically generated Java code fragments build a substantial part of the functionality and prevent the loss of important information during the transition from a model to its implementation.
Keywords: Collaboration diagram, methodical guidelines, code generation, Java, pattern-based transformation algorithm

1 Introduction

The Unified Modeling Language (UML, [8,9,13]) provides a variety of diagram types for an integrated specification of both the structure and the behavior of a system. Collaboration diagrams belong to the behavioral diagrams like sequence diagrams, statecharts and activity diagrams.

Tools to support the development of software, so-called CASE tools, often do not only support the analysis and design of systems, but also contain code generators to automatically create code fragments of the specified system in a target programming language. Unfortunately, the capabilities of code generators to transform the design to an implementation are often restricted to produce class definitions consisting of attributes and operation signatures captured in class diagrams, but not the methods to implement the procedural flow within the operations.

Using also behavioral information for code generation prevents the loss of substantial information during the transition of a model to its implementation. Existing approaches in this direction transform statecharts into executable code [5,1,2]. Statecharts are used as object controllers for specifying when an object

is willing to accept requests. CASE tools supporting code generation from statecharts are e.g. Statemate [15], Omate [5], and Rhapsody [11].

In contrast, it is our aim to transform the specification of the *functional behavior* of objects into code fragments. The functional model can be described in terms of interactions between objects in an abstract way by UML interaction diagrams.

The only tool known to us that is capable of generating code from interaction diagrams is Structure Builder [16]. Sequence diagrams are used there, but code is not directly generated from them, but from an intermediate representation called Sequence Methods. Sequence Methods are based on the concept of Interaction Graphs [14], resulting from the Demeter project [7], which are directed labeled trees with nodes representing object variables and edges representing actions. They basically resemble a representation of additional information that, in agreement with our approach, needs to be interactively entered by a developer to extend the interaction modeled in UML diagrams. Such information being necessary for the generation of working Java code is e.g. how objects can be accessed, how they are transported between methods, and instantiation of links etc. These details can not be specified in sequence diagrams, but most of them are already captured in collaboration diagrams.

Thus, we selected collaboration diagrams from UML interaction diagrams as the source for the transformation process since, in contrast to sequence diagrams, they do not only supply the message flow information of an interaction, but also the underlying structural information building the context of the interaction, i.e. the links via which messages are sent. Additionally, we stay within the diagram types of UML whereas Sequence Methods are outside the UML.

Java was selected as the target language because it is a purely object-oriented programming language of growing importance and it offers concepts for concurrent programming to extend the transformation mechanisms to parallel flow of control.

The paper is organized as follows: In Sect.2, we introduce the main features of collaboration diagrams and state methodical guidelines for their deployment. A general overview of the transformation approach for collaboration diagrams based on the transformation of class diagrams is given in Sect.3. The next section introduces a refined meta model which forms the basis for a detailed description of the transformation algorithm for collaboration diagrams in Sect.5. The paper ends with some concluding remarks and perspectives.

Further details can be found in an extended version of this paper that is available as a technical report (see [4]).

2 Deploying UML Collaboration Diagrams

In this section, we outline a methodical approach on how to deploy UML collaboration diagrams to model functional behavior. This approach is based on the general UML specification [8,9], but it extends it by additional pragmatic guidelines and constraints. A systematic usage of this approach will ensure that collaboration diagrams describing the functionality of methods can automatically be translated into corresponding Java code. In the following, we assume that the reader is familiar with the standard UML notations (see [8,13]).

In general, collaboration diagrams can be used to model system functionality or more precisely the control flow within a system. This is described by sending messages between instances of classes. Collaboration diagrams are feasible to model not only the behavioral, but also the structural context of such an interaction, called a *collaboration*.

In [8], the following two possibilities, among others, of deploying collaboration diagrams in the above sense are introduced:

- Method: specify the implementation of an operation as an interaction,
- Use case: describe the functionality of a main operation of a system on an abstract level.

Both kinds of usage differ not only on their level of abstraction, but also in their main intention. Whereas use cases are deployed in earlier phases of modeling, the method-oriented usage is already close to implementation. Use cases describe scenarios. They are intended to examplify a certain situation, i.e., very often they describe only one possible control flow path. In contrast, within a method specification, the general situation with all posssible control flow paths has to be modeled. As a consequence, collaboration diagrams are used on the instance level in the first case, describing the interaction of different objects with each other. In the second case, they are used on the type level possibly containing iterations or conditional flows [9]. Type level modeling is in accordance with the specification of methods within classes of object-oriented programming languages.

With this background, two main steps can be identified within the development process producing the systems functionality. The first task is stepping from different scenarios to the general situation. And the second task is stepping from a model of the general situation to its implementation. In this paper, we concentrate on the second task, where one type level model for each method, i.e., exactly one collaboration diagram per method, serves as the basis for automatic code generation.

The first task of combining different instance level collaboration diagrams specifying the same operation can not be done automatically in the general case. Collaborations define views on the classes specified in the class diagram. Therefore, problems in combining several collaboration diagrams resemble typical problems of view integration [3]. Input by a developer is necessary to handle conflicts or to specify details of combination like contextual constraints or conditions. This interactive intervention should receive support by code generation tools. Situations where an automatic combination is possible are, e.g., mutual exclusive execution conditions for different occurences of the same operation for branching as well as iteration.

On the other hand, collaboration diagrams are not able to fully model the functionality of an operation. One restriction is their inability to model operations on data types, i.e., primitive base types like Integer, Real or predefined enumeration types like Boolean, whose values do not posses an identity. Thus, collaboration diagrams can not serve as a fully-fledged visual programming language. Moreover, usually not all aspects of a system are completely modeled. Exception handling, for example, will usually be separately specified and added

later in the implementation. For these reasons, code generation from collaboration diagrams is by their definition restricted to object interactions. Generating this kind of working code, prevents the loss of information during the step from modeling to implementation and simplifies the task of transition what states our objectives.

Before we start explaining our approach in more detail, we introduce a running example for a system to be modeled. Figure 1 shows the class diagram of an example application where a Company object is related to zero or more Store, Order, and Delivery objects. A Store is related to multiple Delivery objects, which in turn are related to one Customer and one Order. A Customer can place several instances of Order, and one or none Delivery objects belongs to an Order.

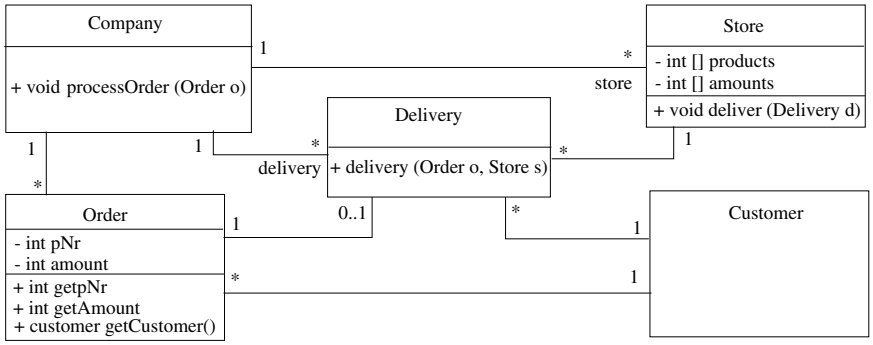


Figure 1. Class diagram of a modeling example

A typical scenario within that setting is the situation where a customer orders a product from the company. On the use case level one would model that scenario by sending an order from a customer to the company, followed by forwarding that order from the company to one of its stores, followed by delivering the ordered products from the store to the customer.

After the step of refining and combining different use cases into a method-oriented specification one might end up with a collaboration diagram for a method processOrder as depicted in Fig.2. Here, the company first obtains the product number pNr and the ordered amount a of that order using defined access functions. It then checks all stores to find one that can supply the requested amount of the demanded product. A delivery is created, and the selected store is called to send it out. Finally, the delivery is added to a container holding all deliveries of the company.

We will now introduce methodical guidelines as a foundation of the later on presented transformation approach. As a consequence of deploying a unique collaboration diagram for specifying the implementation of one operation, two basic model entities build the basis for the forthcoming concepts:

– The *specified operation* is the operation whose implementation is modeled by the collaboration diagram (processOrder in Fig.2).

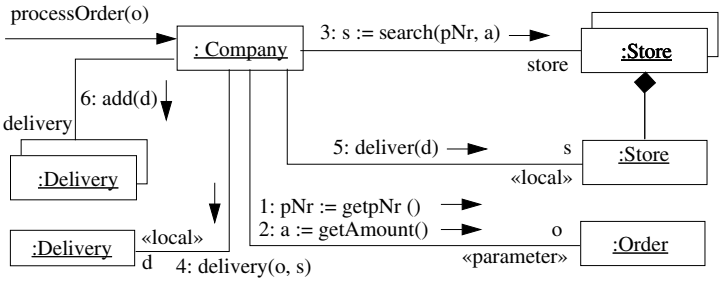


Figure 2. Collaboration diagram for the modeled example

– The *target object* is that object on which the specified operation is called. The specified operation belongs to the class of the target object, its signature must be declared in the operation compartment of the corresponding class in the class diagram (:Company in Fig.2).

As a result of the refinement and combination of different scenario-oriented collaboration diagrams we obtain a collaboration diagram with a single level of nesting. Thus, we specify which operations are called in the specified operation directly, but we do not consider those that are subordinately called within these nested operations. We consider this to be meaningful when we specify the implementation of an operation, since the subordinately called operations belong to collaboration diagrams for the nested operations. This is alike the definition of procedures and procedural calls in programming languages. As an implication, the target object is the sender of the call message for all operations in a collaboration diagram except for the specified operation.

One end of a stereotyped link must be directly connected to the target object (see Fig.2). Conventional links based on associations can also be indirectly connected to the target object. They can be accessed by traversing along a path of links of which only the first may be a stereotyped link. If a link with the stereotype ≪parameter≫ (e.g. to :Order in Fig.2) is used, then a reference to the object on the other end of the link must be transported to the target object as a parameter of the specified operation. The names of objects that are connected to the target object by a ≪parameter≫ link must be identical to the parameter names of the corresponding operation in the collaboration diagram.

Stereotyped links of kind ≪local≫ (e.g. between :Company and s:Store in Fig.2) depict that the linked objects are locally accessible within the specified operation. This stereotype can be used either if the reference to the linked object was obtained as a return value of a previously called operation or if the linked object was initialized by calling a constructor within the specified operation. The same restrictions apply to stereotyped links of kind ≪global≫. Additionally, global variables can also be initialized within another collaboration, i.e. in a different collaboration diagram.

To prevent ambiguities, role names on association links are needed in the case that multiple links exist between two objects. Calling a constructor across an

association link implies that both the receiver object and the link are implicitly {new} (see 4: in Fig.2). Thus, the constraint is optional. In contrast, adding to and deleting from multiobjects (notion for container in UML collaboration diagrams) can be explicitly defined by the modeler in order to specify the exact sequence of messages (see 6: in Fig.2).

Objects may not be marked with the constraint {destroyed} because Java does not contain a predefined destructor. Otherwise, one would have to solve the problem that all references to that object must be deleted to make the garbage collector delete the object, even those references specified in other collaborations.

Further details of the implications of our approach will be shown in Sect.4 where the refined meta model for collaborations is presented.

3 Transformation Approach

In Sect.2, methodical guidelines on how to deploy collaboration diagrams have been explained. Following these, all collaboration diagrams to be translated have a well-formed structure. This is an important prerequisite and enables a systematic translation of collaboration diagrams into corresponding Java code.

The translation algorithm for collaboration diagrams is based on a standard algorithm for translating class diagrams. The underlying idea is to translate class definitions into corresponding Java class definitions and to translate associations into bi-directional references between the two participating classes. This standard algorithm has been refined, e.g., with respect to automatically generated "get" and "put" access operations for attributes or a generic search operation to select certain objects from a set of existing objects. Further details on the refined class translation algorithm can be found in [4].

The basic idea of the overall transformation algorithm from a class diagram and associated collaboration diagrams into corresponding Java code is to identify standard patterns in a given diagram and to translate those patterns into corresponding Java code. This *pattern-based transformation algorithm* will be presented in a technical, formal way in Sect.5. Here, we give two simple examples to sketch informally how this pattern-based translation works.

First, Fig.3(a) shows a part of the collaboration diagram given in Fig.2 where operation getpNr() is sent via a parameter link with role o to an object of class Order. This collaboration diagram is depicted in the lower left part of Fig.3(a), while the corresponding class diagram can be found in the upper left part. The right hand side shows the generated Java code for such a parameter link pattern within a collaboration diagram.

Second, Fig.3(b) shows another pattern taken from Fig.2. Here, the collaboration diagram comprises a pattern consisting of a local link combined with a newly created object of class Delivery. The resulting Java code comprises a definition of a local variable d of type Delivery, as well as the invocation of the constructor of class Delivery in order to create a new instance.

The complete structured and pattern-based transformation algorithm will be explained in Sect.5. In order to be able to describe certain patterns within a class or collaboration diagram, a uniform internal representation of diagrams is an important prerequisite. As known from the UML language definition, such

an internal representation can best be defined by a meta model. Therefore, the next section will present an adapted UML meta model, which incorporates the restrictions introduced in Sect.2.

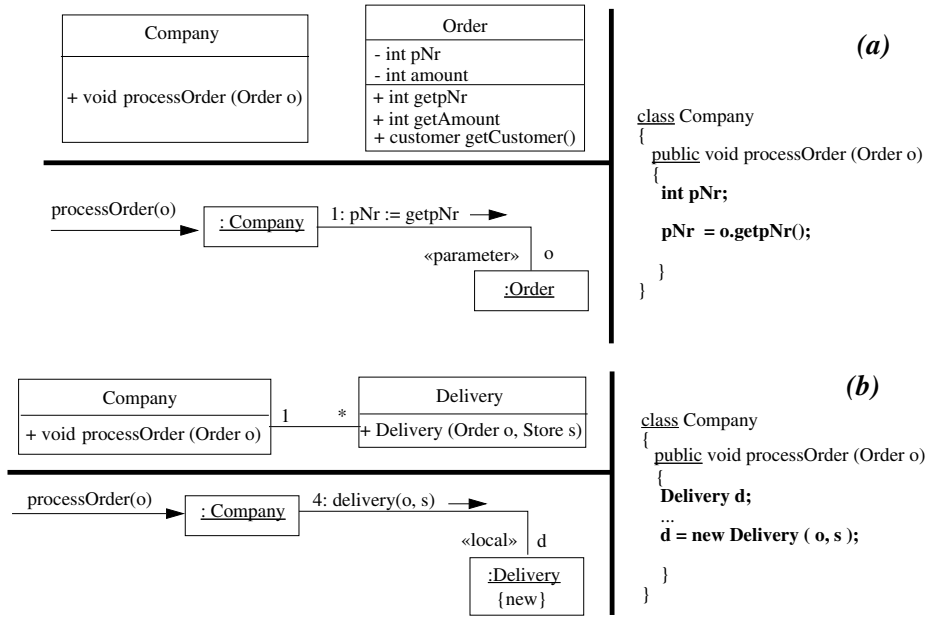


Figure 3. Transformation of (a) parameter and (b) local links into Java code

4 Refined Meta Model

Based on the UML meta model, we present a refined meta model for collaborations, that has been adapted according to the assumptions and restrictions described in Sect.2. The methodical guidelines for deploying collaboration diagrams to model method implementations have been integrated and are thus reflected on the meta model level now. Since the transformation algorithm presented in the next section is based on this meta model representation, the methodical guidelines also affect the code generation. The benefits of this adapted meta model are two-fold. First, the methodical guidelines have become part of the modeling language. Thus, only well-structured collaboration diagrams can be instantiated from this meta model. Second, the adapted meta model shows a granularity which is very well suited as basis for the pattern-based transformation algorithm.

Figure 4 depicts the changes to the original UML meta model. Elements that are replaced or deleted are crossed out, while new or changed meta classes and associations are shaded. Note that associations connected to new classes are also new even if they are not explicitly marked for simplicity. Some classes from other

meta model packages of UML have been included, but all changes to existing associations have been marked.

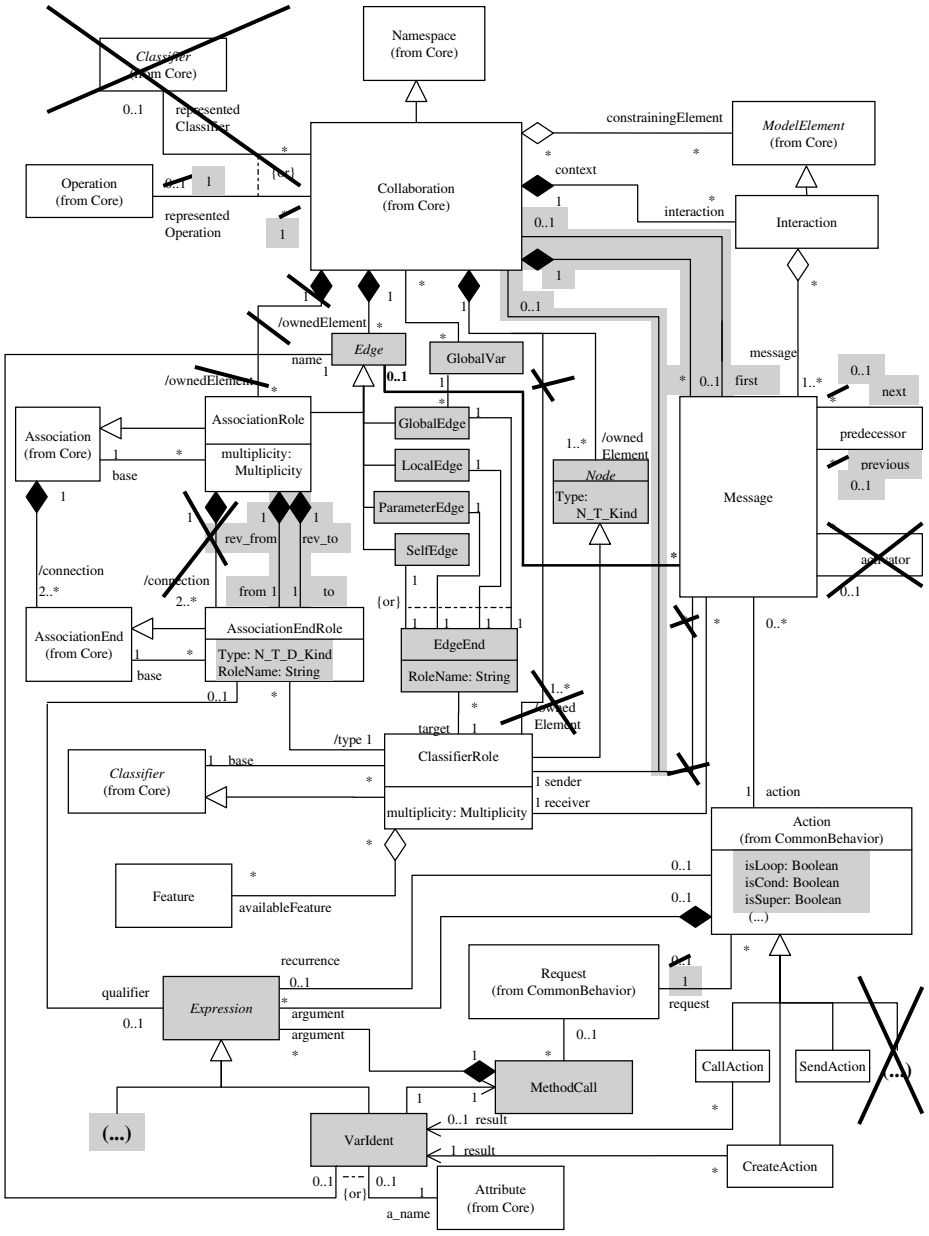


Figure 4. Extended meta model for transforming collaboration diagrams

Due to the use of collaboration diagrams for specifying the implementation of operations, the upper left occurence of class Classifier disappears from the meta model. Additionally, we argued (see Sect.2) that the implementation of every operation is specified by exactly one collaboration diagram what is reflected by the one-to-one association between the corresponding classes.

Two new associations between the meta model classes Collaboration and Message are added to simplify navigation through the meta model according to the specified message sequence. The multiplicities on the predecessor association are changed and the activator association is removed because the transformed diagrams contain only one level of nesting. For the same reason, the association to ClassifierRole with the role name sender is bent, now connecting ClassifierRole and Collaboration: The sender for messages within this collaboration is the target object on which the specified operation is called (see Sect.2).

To account for the distinct algorithmic transformation of the different link types, we introduce meta model classes for stereotyped links LocalEdge, GlobalEdge, ParameterEdge, and SelfEdge, and the abstract super class Edge. The new class EdgeEnd builds the counterpart to AssociationEndRole for the stereotyped links. We replace the composition relation between AssociationRole and AssociationEndRole by two associations modeling directed association roles. This is possible since we have only one level of nesting and we restrict the transformation to binary associations. The transformation algorithm uses the roles to and from to traverse association links in the direction of message flow.

New is also the abstract meta class Node as a super class of ClassifierRole. Its purpose is to hold an attribute of type N_T_Kind representing the default constraints {new} and {transient} that can be attached to an object (ClassifierRole) in a collaboration diagram. An equivalent attribute of the super type N_T_D_Kind has been added to the class AssociationEndRole. Due to this extension, the mapping of constraints on the appropriate subclasses of the meta model class Action [8] is no longer needed.

We further introduce a meta model class Expression and subclasses (not shown on the diagram) for data values, operators and their operands, etc. to decompose expressions in their components. This enables the definition of access functions for objects that are referenced by a link based on an association. The recurrence attribute of the class Action is changed into an association. Simple expressions are either instances of a base type or a variable identifier.

If an operation yields a result, the return action in UML is specified by a separate return message [8]. In contrast, the return message is not explicitly modeled in the refined meta model. Instead, the name of a variable for the return value is explicitly stored in the meta model class VarIdent. This variable name is related to either an attribute of the target object, a stereotyped link, or an association link, represented by alternative associations to Attribute and Edge. The role name belonging to such an edge is equivalent to the variable name. The meta model class MethodCall is used to specify a method called on a variable identifier using the dot notation.

Another meta model class GlobalVar is added to hold the names of global variables that are referenced by ≪global≫ links within all collaboration diagrams.

Only three subclasses of the meta model class Action remain in the meta model for the transformation of collaboration diagrams to Java, since Java has

no predefined destructor. For every instance of Action or its subclasses, exactly one Request instance is linked.

5 Transformation Algorithm

In this section, the algorithm for transforming collaboration diagrams to Java is specified in a rule-based way. In order for the algorithm to work correctly, collaboration diagrams are assumed to be syntactically and static-semantically correct. Moreover, the whole model consisting of a class diagram and a collaboration diagram for each operation defined in the class diagram has already been translated into an instance of the meta model as described in Sect.4.

We use a kind of *meta rules* consisting of a rule scheme and an additional pattern. The *rule scheme* describes the generation of syntactically correct Java code. It has the form of a context free rule expression. But it is still independent of a concrete collaboration diagram. It contains two kinds of non-terminal symbols. The first are non-terminals in the usual sense replaced by sequences of non-terminals and terminals by the application of rules. Only those will be called non-terminals in the following. The second kind are parameters of the rule scheme, which allow its instantiation for a concrete diagram to be transformed. These parameters are formulated using terms of the meta model. This approach stems originally from the compiler construction area, where it is known as a two-level grammar approach ([17]).

The *pattern* is a part of an instance diagram of the meta model. It is used to represent those parts of a concrete diagram which shall be actually transformed. Hence, the occurence of the pattern in the instance diagram for the example application for which code shall be generated serves as an application condition for the whole meta rule to be applied. Moreover, the concrete occurence links together the general code generation possibilities, described by the rule scheme, and the actual elements of the concrete collaboration diagram that has to be transformed. The parameters of the rule scheme occur in the pattern and can hence be replaced by actual values in order to instantiate the rule scheme.

Figure 5 shows two meta rules for the transformation of class diagrams. These meta rules will be used in the following to illustrate how the algorithm is specified in principle. On the left hand side, the part of the class diagram actually translated by the meta rule is shown. In the middle, we give its translation to part of an instance of the meta model, which forms the pattern. On the right hand side, the rule scheme for generating Java code is shown. Words in capital letters denote non-terminal symbols, whereas words in small letters denote terminal symbols if they are underlined, or they denote parameter expressions over the pattern if not. These parameters will be evaluated to terminal symbols as soon as a concrete occurence of the pattern is chosen, leading to an instantiation of the rule scheme for the concrete diagram.

The first meta rule shown in Fig.5 allows to transform a single class into the frame for a class declaration in Java. Here, c refers to the instance of classifier which represents the class in the instance of the meta model. Hence c.name is a parameter which will be replaced by the name of the class, i.e., the concrete value of this attribute in an occurence of the pattern. The non-terminal symbols

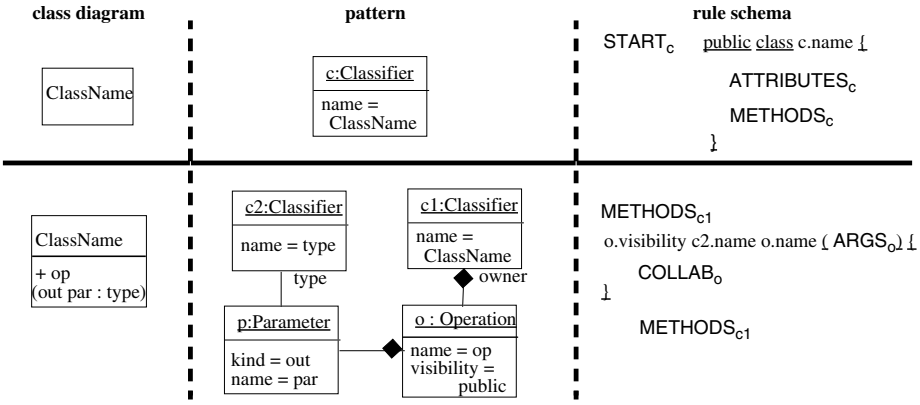


Figure 5. Meta rules for class diagram

START$_c$, ATTRIBUTES$_c$ and METHODS$_c$ will also be instantiated with more concrete non-terminal symbols. The name c of the classifier object is used to keep track of the concrete classifier object currently dealt with during the next steps of code generation. It already determines partly the occurence of the pattern belonging to the meta rule for replacing this non-terminal. The second meta rule shown in Fig.5 serves for the generation of the method frames for each operation defined in the class diagram in an analogous way. The instantiation of the non-terminal symbol COLLAB$_o$ will be replaced by the code generated for the collaboration diagram of this operation.

Note that the meta rules are only applied once for each occurence of the according pattern. Different occurences may overlap. For example, in case of the second rule the same classifier object may occur as owner of an operation and as parameter of another or even of the same operation.

Consider again our example application introduced in Sect.2. The class diagram shown in Fig.1 can be transformed into Java code using the above meta rules in the following way: We search for an occurence of the pattern of the first meta rule in the instance diagram of the meta model. Classifier c is mapped to classifier com, whose name attribute has the value "Company". For this occurence of the pattern we instantiate the rule scheme leading to

$$\text{START}_{com} \longrightarrow \underline{\text{public}}\ \underline{\text{class}}\ \underline{\text{Company}}\ \underline{\{}\ \text{ATTRIBUTES}_{com}\ \text{METHODS}_{com}\ \underline{\}}$$

In the second step, we want to replace the non-terminal METHODS$_{com}$. This could be done by using the second meta rule. But we need a particular instantiation of the according rule scheme (METHODS$_{com}$ instead of METHODS$_{c1}$). Hence the occurence for the pattern has to obey this constraint. If an occurence is found that maps operation o to operation procOrd with name processOrder and visibility public, the rule scheme can be instantiated to

$$\text{METHODS}_{com} \longrightarrow \underline{\text{public}}\ \underline{\text{void}}\ \underline{\text{processOrder}}\ \underline{(}\ \text{ARGS}_{procOrd}\ \underline{)\ \{}\ \text{COLLAB}_{procOrd}\ \underline{\}}$$

With the above two rules, we can deduce a primitive class frame from the start symbol START_{com}. Note that the instantiation process leads to a set of different start symbols since the generated Java code has to be stored in different files.

We now advance to the transformation of collaboration diagrams. We start with meta rules for replacing the non-terminal COLLAB_o by a sequence of other non-terminals in order to determine the structure of the generated code of the body of a method. First, the local variables have to be declared. Then, we invoke the methods in the order which is indicated in the collaboration diagram by the sequence numbers. Finally, we have to add newly inserted links, which are not used to invoke a method, and to remove links, which are indicated as destroyed. The according meta rule is depicted in Fig.6.

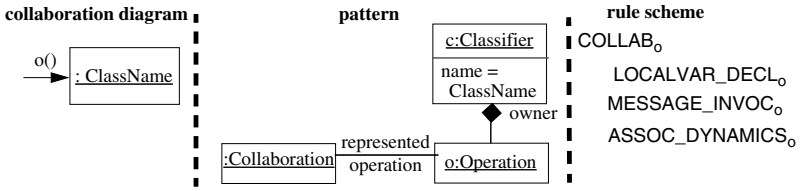


Figure 6. Meta-Rule for splitting of COLLAB

In the sequel, the first two meta rules generated by this substitution are explained in detail. Figure 7 shows the meta rule for declaring local variables. Remember that we also assume that indirectly declared local variables (return values of method invocations referencing objects) are to be represented as local edges in the instance of the meta model. Hence each LocalEdge uniquely represents a local variable, the name of which is stored as the RoleName of its EdgeEnd. The LocalEdge belongs to the collaboration of operation o. The type of a local variable is given by the name of the base (classifier) of the target of the EdgeEnd. This information is represented in the pattern. Moreover, it is used in the rule scheme by the parameters c.name for the type and e.RoleName for the name of the local variable. We add the possibility of declaring more than one local variable within the same operation o by repeating the non-terminal LOCALVAR_DECL_o. Again, different applications of the meta rule imply different occurences of the pattern ensuring that each local variable is declared only once. The meta rule in the lower part of Fig.7 serves for the end of the declaration process. The rule scheme replaces the non-terminal LOCALVAR_DECL_o by the empty string. It may only be applied, if the upper meta rule is not applicable any more.

Now we come to the generation of the real body of an operation, namely the invocation of methods. Generally, we have to generate the method invocation code in the order indicated by the sequence numbers in the collaboration diagram. This order is represented in the meta model by the predecessor edge between messages and by the edge assigning the first message to a collaboration.

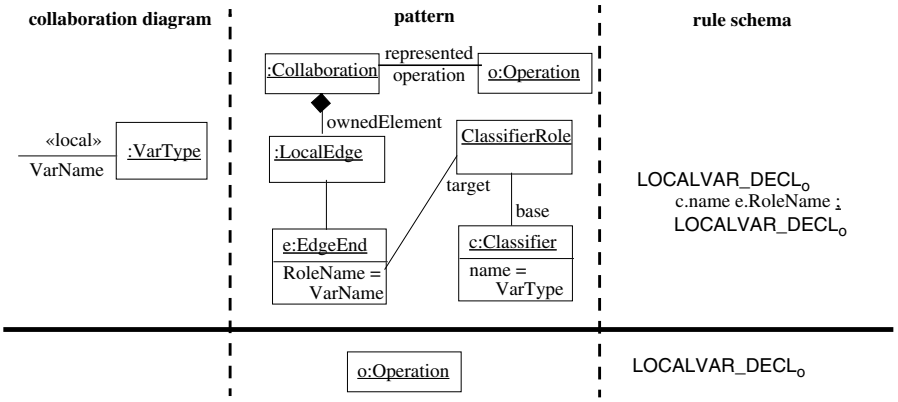


Figure 7. Meta rule for local variable declaration

Hence, we have three kinds of meta rules: The first kind serves for invoking the first method. The second kind traverses the predecessor edge from the previous to the next message. The third kind ends the process. Meta rules of the last kind look like the last one discussed for the local variable declaration above. They are neglected in the following.

For the first two kinds of meta rules, we additionally have to distinguish many different cases: whether the receiver of the message is a multiobject, whether it is a newly created object, whether a parameter, a local or global variable or an existing resp. new association is used to send the message to the receiver, whether a return value is expected or not, whether the method invocation is conditional or an iteration, and whether the method itself or the method of the super class is called. Due to space limitations, we are not able to present all according meta rules in this paper (see [4] for an exhaustive presentation).

Instead, Fig.8 shows as an example a meta rule for invoking a method on a parameter object. It is a rule of the first kind, meaning that the method invocation is the first one in the actually transformed collaboration. A method for operation o is invoked. The kernel of this method invocation is that an operation r.name is called on the parameter object referred to by e.RoleName. The arguments for this call are generated from the non-terminal symbol ARGS_r. We omit a more detailed view on that, since we left out the specialization of class Expression in the meta model in Sect.4 that is necessary for this purpose. The meta rules for invoking an operation on a local or global variable look quite similar. Only the parameter edge in the pattern is replaced by a local or global edge, respectively. The transformation of a «self» link is handled analogously, distinguishing between using a this-pointer or a super-pointer to call a redefined method of a super class.

The pattern of the meta rule for method invocation via an association link differs in that ParameterEdge and EdgeEnd are replaced by AssociationRole and AssociationEndRole, respectively. Other additional requirements on attributes of the CallAction and the receiving ClassifierRole ensure that one deals with the simplest case and not with multiobjects, for instance. Another difference is

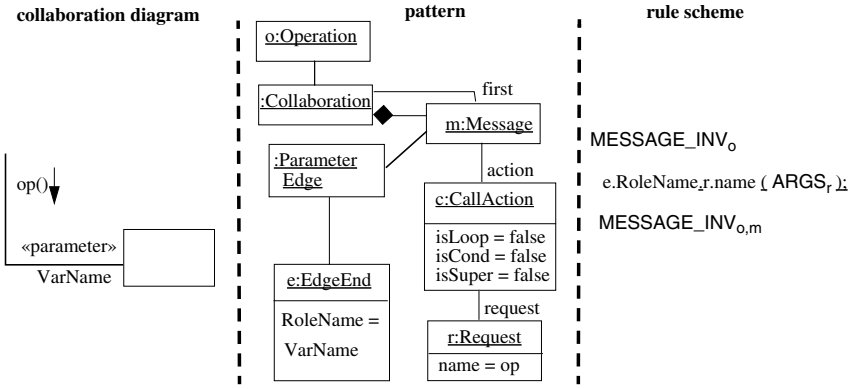


Figure 8. Meta rule for method invocation on parameter object

that the method may not directly be called using the RoleName stored in the AssociationEndRole if sender and receiver are only indirectly linked. Hence we include a non-terminal symbol $\text{PATH}_{s,r}$ which has to be replaced by an expression determining the shortest existing link path from the sender s to the receiver r.

In order to allow more than one method invocation in the body of an operation, the rule scheme in the above meta rule generates a new non-terminal symbol $\text{MESSAGE\_INV}_{o,m}$, distinguished by the differing parameter expression. This second kind of non-terminals for method invocations can be replaced by the second kind of meta rules.

Using the complete set of meta rules as shown for the generation of the code for the class diagram by instantiating the meta rules and reducing the non-terminals to terminal symbols, the following Java code is generated from the collaboration diagram depicted in Fig.2.

```
public class Company {
        ...
        public void processOrder (Order o) {
           Delivery d;
           Store s;
           int pNr;
           int a;
           pNr = o.getpNr();
           a = o.getAmount();
           s = search_stores(pNr, a);
           d = new Delivery(o,s);
           s.deliver(d);
           add_deliveries(d);
        }
        ...
}
```

6 Conclusion and Perspectives

In this paper, we have investigated the modeling of behavior by UML collaboration diagrams and their automatic transformation into Java code. We have

introduced methodical guidelines how to deploy collaboration diagrams in a structured way. This formed the basis for the formulation of a transformation algorithm.

The objective of this automatic transformation is to prevent a loss of substantial information during the transition from a model to its implementation. But, this does not imply that UML collaboration diagrams offer a means to specify the behavior of a system completely and that UML can be used as a visual programming language. UML collaboration diagrams focus on the modeling of object interactions, while computations on data values are neglected, and thus have to be added to the generated Java code by hand.

This paper focussed on the transformation of sequential behavior descriptions. The next steps will be to implement the transformation algorithm by extending the often used, commercial tool Rational Rose [10] and to extend the transformation algorithm to the transformation of concurrent behavior as well as of asynchronous and synchronous communication descriptions. The already chosen target language Java will facilitate this development. First results of that extension can be found in [6].

Finally, it is intended to investigate whether and how the in this paper reused approach of two-level grammars (cf. [17]) is an appropriate means to specify and realize easily adaptable code generators for forthcoming versions of UML and for a visual modelling language in general.

References

1. Ali, J., Tanaka, J.: Generating executable code from the dynamic model of OMT with concurrency. In: *Proc. IASTED International Conference on Software Engineering (SE'97)*, San Francisco, 1997, pp. 291–297
2. Ali, J., Tanaka, J.: Implementation of the dynamic behavior of object oriented systems. In: *Integrated Design and Process Technology (IDPT)*, Vol. 4, Society for Design and Process Science, 1998, pp. 281–288
3. Engels, G., Heckel, R., Taentzer, G., Ehrig, H.: A view-oriented approach to system modelling using graph transformations. In Jazayeri, M., Schauer, H. (eds.): *Proceedings European Software Engineering Conference (ESEC'97)*, Zürich, LNCS 1301, Springer, 1997, pp. 327–343
4. Engels, G., Hücking, G., Sauer, S., Wagner, A.: UML Collaboration Diagrams and Their Transformation to Java. Technical Report TR-RI-99-208, University of Paderborn, 1999
5. Harel, D., Gery, E.: Executable Object Modeling with Statecharts. *IEEE Computer*, **30** (July 1997) 31–42
6. Hücking, R.: UML Collaboration Diagrams and Their Transformation to Java (in German). Master's Thesis, University of Paderborn, September 1998
7. Lieberherr, K.: *Adaptive Object-Oriented Software: The Demeter Method with Propagation Patterns*. PWS Publishing Company, Boston MA, 1996
8. OMG: UML Notation Guide, Version 1.1. The Object Management Group, Document ad/97-08-05, Framingham MA, 1997
9. OMG: UML Semantics. Version 1.1. The Object Management Group, Document ad/97-08-04, Framingham MA, 1997
10. *Rational Rose 98*. Rational Software Corporation, Cupertino CA, 1998
11. *Rhapsody*. Version 2.1. I-Logix, Andover MA, 1998

12. Rumbaugh, J., Blaha, M., Premerlani, W., Eddy, F., Lorensen, W.: *Object-Oriented Modelling and Design*. Prentice-Hall, 1991
13. Rumbaugh, J., Jacobson, I., Booch, G.: *The Unified Modeling Language Reference Manual*. Addison-Wesley, Reading MA, 1999
14. Sangal, N., Farrel, E., Lieberherr, K.: Interaction Graphs: Object interaction specifications and their compilation to Java. Technical Report NU-CCS-98-11, Northeastern University, Oct. 1998
15. *Statemate MAGNUM*. Release 1.2. I-Logix, Andover MA, 1999
16. *Structure Builder*. Version 3.1.5. Tendril Software Inc., Westford MA, 1999
17. A. van Wijngaarden: The Generative Power of Two-Level Grammars. In J. Loeckx (ed.): *Automata, Languages and Programming*, 2nd Colloquium, University of Saarbrücken, 1974. LNCS 14, Springer, 1974, pp. 9 –16

Strengthening UML Collaboration Diagrams by State Transformations*

Reiko Heckel and Stefan Sauer

University of Paderborn, Dept. of Mathematics and Computer Science
D-33095 Paderborn, Germany
reiko|sauer@uni-paderborn.de

Abstract. Collaboration diagrams as described in the official UML documents specify patterns of system structure and interaction. In this paper, we propose their use for specifying, in addition, pre/postconditions and state transformations of operations and scenarios. This conceptual idea is formalized by means of graph transformation systems and graph process, thereby integrating the state transformation with the structural and the interaction aspect.

Keywords: UML collaboration diagrams, pre/postconditions, graph transformation, graph process

1 Introduction

The Unified Modeling Language (UML) [24] provides a collection of loosely coupled diagram languages for specifying models of software systems on all levels of abstraction, ranging from high-level requirement specifications over analysis and design models to visual programs. On each level, several kinds of diagrams are available to specify different aspects of the system, like the structural, functional, or interaction aspect. But even diagrams of the same kind may have different interpretations when used on different levels, while several aspects of the same level may be expressed within a single diagram.

For example, interaction diagrams in UML, like sequence or collaboration diagrams, often represent sample communication scenarios, e.g., as refinement of a use case, or they may be used in order to give a complete specification of the protocol which governs the communication. Collaboration diagrams allow, in addition, to represent individual snapshots of the system as well as structural patterns.

If such multi-purpose diagrammatic notations shall be employed successfully, a precise understanding of their different aspects and abstraction levels is required, as well as a careful analysis of their mutual relations. This understanding, once suitably formalized, can be the basis for tool support of process models, e.g., in the form of consistency checks or refinement rules.

* Research partially supported by the ESPRIT Working Group APPLIGRAPH.

In this paper, we address these issues for UML collaboration diagrams. These diagrams are used on two different levels, the instance and the specification level, both related to a class diagram for typing (cf. Fig. 1). A specification-level diagram provides a pattern which may occur at the instance level.[1]

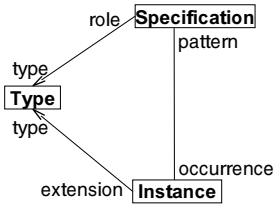


Fig. 1. Two levels of collaboration diagrams and their typing

In addition, in the UML specification [24] two aspects of collaboration diagrams are identified: the *structural aspect* given by the graph of the collaboration, and the *interaction aspect* represented by the flow of messages. These aspects are orthogonal to the dimensions in Fig. 1: A specification-level diagram may provide a structural pattern as well as a pattern of interaction. At the instance level, a collaboration diagram may represent a snapshot of the system or a sample interaction scenario. Moreover, both aspects are typed over the class diagram, and the pattern-occurrence relation should respect this typing.

One way to make precise the relationships between different diagrams and abstraction levels is the approach of *meta modeling* used in the UML specification [24]. It allows to specify the syntactic relation between different diagrams (or different uses of the same diagram) by representing the entire model by a single *abstract syntax graph* where dependencies between different diagrams can be expressed by means of additional links, subject to structural constraints specifying consistency. This approach provides a convenient and powerful language for integrating diagram languages, i.e., it contributes to the question, *how* the integration can be specified. However, it provides no guidelines, *what* the intended relationships between different diagrams should be.

Therefore, in this paper, we take the alternative approach of translating the diagrams of interest into a formal method which is conceptually close enough in order to provide us with the required semantic intuition to answer the *what* question. Once this is sufficiently understood, the next step is to formulate these results in the language of the UML meta model.

Our formal method of choice are graph transformation systems of the so-called *algebraic double-pushout (*DPO*) approach* [10] (see [5] for a recent survey).

[1] The use of collaboration diagrams for *role modeling* is not captured by this picture. A role model provides a refinement of a class diagram where roles restrict the features of classes to those relevant to a particular interaction. A collaboration diagram representing a role model can be seen as a second level of typing for instance (and specification-level) diagrams which itself is typed over the class diagram. For simplicity, herein we restrict ourselves to a single level of typing.

In particular, their typed variant [4] has built in most of the aspects discussed above, including the distinction between pattern, instance, and typing, the structural aspect and (by way of the partial order semantics of *graph processes* [4]) a truly concurrent model for the interaction aspect. The latter is in line with the recent proposal for UML action semantics [1] which identifies a semantic domain for the UML based on a concurrent data flow model.

The direct interpretation of class and collaboration diagrams as graphs and of their interrelations as graph homomorphisms limits somewhat the scope of the formalization. In particular, we deliberately neglect inheritance, ordered or qualified associations, aggregation, and composition in class diagrams as well as multi-objects in collaboration diagrams. This oversimplification for presentation purpose does not imply a general limitation of the approach as we could easily extend the graph model in order to accommodate these features, e.g., using a meta model-based approach like in [21].

Along with the semantic intuition gained through the interpretation of collaboration diagrams in terms of graph transformation comes a conceptual improvement: the use of collaboration diagrams as a visual query and update language for object structures. In fact, in addition to system structure and interaction, we propose the use of collaboration diagrams for specifying the *state transformation* aspect of the system. So far, this aspect has been largely neglected in the standard documents [24], although collaboration diagrams are used already in the CATALYSIS approach [6] and the FUSION method [3] for describing pre- and postconditions of operations and scenarios.

Beside a variety of theoretical studies, in particular in the area of concurrency and distributed systems [9], application-oriented graph transformation approaches like PROGRES [29] or FUJABA [14] provide a rich background in using rule-based graph transformation for system modeling as well as for testing, code generation, and rapid prototyping of models (see [7] for a collection of survey articles on this subject). Recently, graph transformations have been applied to UML meta modeling, e.g., in [16, 2, 11].

Therefore, we believe that our approach not only provides a clarification, but also a conceptual improvement of the basic concepts of collaboration diagrams.

Two approaches which share the overall motivation of this work remain to be discussed, although we do not formally relate them herein. Övergaard [26] uses sequences in order to describe the semantics of interactions, including notions of refinement and the relation with use cases. The semantic intuition comes from trace-based interleaving models which are popular, e.g., in process algebra. Knapp [22] provides a formalization of interactions using temporal logic and the *pomset* (partially ordered multi-set) model of concurrency [27]. In particular, the pomset model provides a semantic framework which has much similarity with graph processes, only that pomsets are essentially set-based while graph processes are about graphs, i.e., the structural aspect is already built in. Besides technical and methodological differences with the cited approaches, the main additional objective of this work is to *strengthen collaboration diagrams by incorporating graph transformation concepts.*

The presentation is structured according to the three aspects of collaboration diagrams. After introducing the basic concepts and a running example in Sect. 2, Sect. 3 deals with the structural and the transformation aspect, while Sect. 4 is concerned with interactions. Section 5 concludes the paper.

A preliminary sketch of the ideas of this paper has been presented in [20].

2 UML Collaboration Diagrams: A Motivating Example

In this section, we introduce a running example to motivate and illustrate the concepts in this paper. First, we sketch the use of collaboration diagrams as suggested in the UML specification [24]. Then, we present an improved version of the same example exploiting the state transformation aspect.

Figure 2 shows the class diagram of a sample application where a Company object is related to zero or more Store, Order, and Delivery objects. Order objects as well as Delivery objects are related to exactly one Customer who plays the role of the customer placing the order or the receiver of a delivery, respectively. A Customer can place several instances of Order and receive an unrestricted number of Delivery objects.

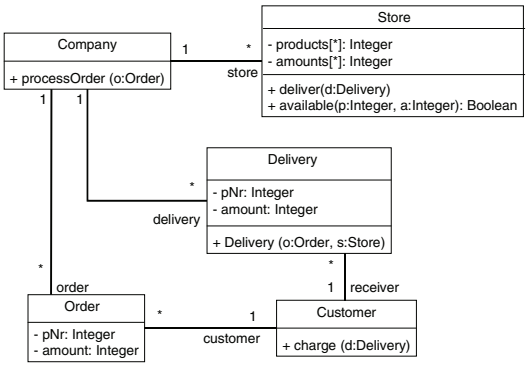


Fig. 2. A class diagram defining the structure of the example

A typical scenario within that setting is the situation where a customer orders a product from the company. After the step of refining and combining different use cases into a method-oriented specification one might end up with a collaboration diagram specifying the implementation of operation processOrder as depicted in the top of Fig. 3. Here, the company first obtains the product number pNr and the ordered amount using defined access functions. It then checks all stores to find one that can supply the requested amount of the demanded product. A delivery is created, and the selected store is called to send it out. Concurrently, the customer is charged for this delivery. After an order has been processed, it will be deleted.

Collaboration diagrams like this, which specifies the execution of an operation, may be used for generating method implementations in Java [12], i.e., they

can be seen as visual representations of programs. However, in earlier phases of development, a higher-level style of specification is desirable which abstracts from implementation details like the get functions for accessing attributes and the implementation of queries by search functions on multi-objects.

Therefore, we propose to interpret a collaboration as a visual query which uses pattern matching on objects, links, and attributes instead of low-level access and search operations. In fact, leaving out these details, the same operation can be specified more abstractly by the diagram in the lower left of Fig. 3. Here, the calls to getPNr and getAmount are replaced by variables p and a for the corresponding attribute values, and the call of search on the multi-object is replaced by a boolean function available which constrains the instantiation of /s:Store. (As specified in the lower right of the same figure, the function returns true if the Store object matching /s is connected to a Product object with the required product number p and an amount b greater than a.) The match is complete if all items in the diagram not marked as {new} are instantiated. Then, objects marked as {destroyed} are removed from the current state while objects marked as {new} are created, initializing appropriately the attributes and links. For example, the new Delivery object inherits its link and attribute values from the destroyed Order object.

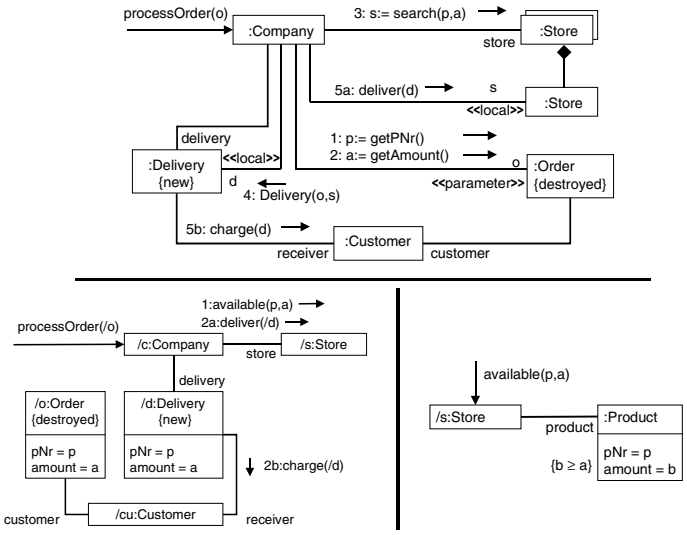


Fig. 3. An implementation-oriented collaboration diagram (top), its declarative presentation (bottom left), and a visual query operation (bottom right)

In the following sections, we show how this more abstract use of collaboration diagrams can be formalized by means of graph transformation rules and graph processes.

3 Collaborations as Graph Transformations

A collaboration on specification level is a graph of classifier roles and association roles which specifies a view of the classes and associations of a class diagram as well as a pattern for objects and links on the instance level. This triangular relationship, which instantiates the type-specification-instance pattern of Fig. 1 for the structural aspect, shall be formalized in the first part of this section. Then, the state transformation aspect shall be described by means of graph transformations. The interaction aspect is considered in the next section.

Structure. Focusing on the structural aspect first, we use graphs and graph homomorphisms (i.e., structure-compatible mappings between graphs) to describe the interrelations between class diagrams and collaboration diagrams on the specification and the instance level.

The relation between class and instance diagrams is formally captured by the concept of *type* and *instance graphs* [4]. By *graphs* we mean directed unlabeled graphs $G = \langle G_V, G_E, src^G, tar^G \rangle$ with set of vertices G_V, set of edges G_E, and functions $src^G : G_E \to G_V$ and $tar^G : G_E \to G_V$ associating to each edge its source and target vertex, respectively. A graph homomorphism $f : G \to H$ is a pair of functions $\langle f_V : G_V \to H_V, f_E : G_E \to H_E \rangle$ compatible with source and target, i.e., for all edges e in G_E, $f_V(src^G(e)) = src^H(f_E(e))$ and $f_V(tar^G(e)) = tar^H(f_E(e))$.

Let TG be the underlying graph of a class diagram, called *type graph*. A legal *instance graph* over TG consists of a graph G together with a typing homomorphism $g : G \to TG$ associating to each vertex and edge x of G its type $g(x) = t$ in TG. In UML notation, we write $x : t$. Observe that the compatibility of g with source and target ensures that, e.g., the class of the source object of a link is the source class of the link's association. Constraints like this can be found in the meta class diagrams and well-formedness rules of the UML meta model for the meta associations relating classifiers with instances, associations with links, association ends with link ends, etc. ([24], Sect. 2.9). The typing of specification-level graphs is described in a similar way ([24], Sect. 2.10).

An interpretation of a graph homomorphism which is conceptually different, but requires the same notion of structural compatibility, is the occurrence of a pattern in a graph. For example, a collaboration on the specification level occurs in a collaboration on the instance level if there exists a mapping from classifier roles to instances and from association roles to links preserving the connections. Thus, the existence of a graph homomorphism from a given pattern graph implies the presence of a corresponding instance level structure. The occurrence has to be type-compatible, i.e., if a classifier role is mapped to an instance, both have to be of the same classifier. This compatibility is captured in the notion of a *typed graph homomorphism* between typed graphs, i.e., a graph homomorphism which preserves the typing. In our collaboration diagrams, this concept of graphical pattern matching is used to express visual queries on object structures.

In summary, class and collaboration diagrams have a homogeneous, graph-like structure, and their triangular relationship can be expressed by three com-

patible graph homomorphisms. Next, this triangular relation shall be lifted to the state transformation view.

State Transformation. Collaborations specifying queries and updates of object structures are formalized as *graph transformation rules*, while corresponding collaborations on the instance level represent individual *graph transformations*.

A *graph transformation rule* $r = L \to R$ consists of two graphs L, R such that the union $L \cup R$ is defined. (This ensures that, e.g., edges which appear in both L and R are connected to the same vertices in both graphs.) Consider the rule in the upper part of Fig. 4 representing the collaboration of processOrder in the lower left of Fig. 3. The precondition L contains all objects and links which have to be present before the operation, i.e., all elements of the diagram except for /d:Delivery which is marked as {new}. Analogously, the postcondition R contains all elements except for /o:Order which is marked as {destroyed}. (The {transient} constraint does not occur because a graph transformation rule is supposed to be atomic, i.e., conceptually there are no intermediate states between L and R.)

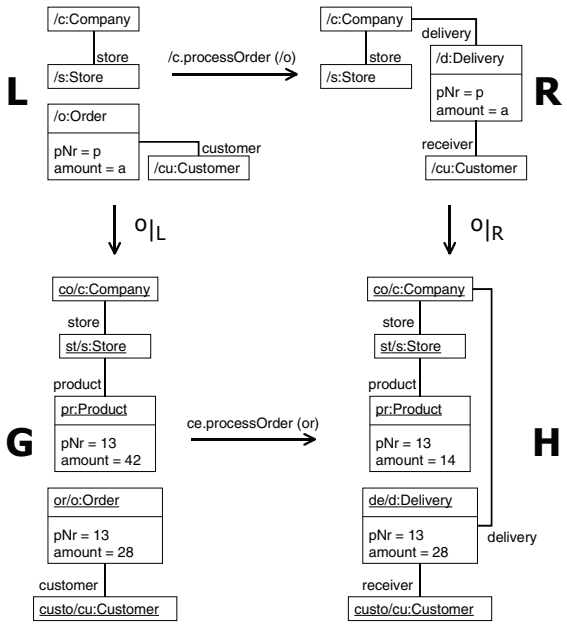


Fig. 4. A graph transition consisting of a rule $L \to R$ specifying the operation processOrder (top), and its occurrence o in an instance-level transformation (bottom)

A similar diagram on the instance level represents a graph transformation. Graph transformation rules can be used to specify transformations in two different ways: either operationally by requiring that the rule is applied to a given graph in order to rewrite part of it, or axiomatically by specifying pre- and postconditions. In the first interpretation (in fact, the classical one [10], in

set-theoretic formulation), a *graph transformation* $G \overset{r(o)}{\Longrightarrow} H$ from a pre-state G to a post-state H using rule r is represented by a graph homomorphism $o : L \cup R \to G \cup H$, called *occurrence*, such that

1. $o(L) \subseteq G$ and $o(R) \subseteq H$ (i.e., the left-hand side of the rule is matched by the pre-state and the right-hand side by the post-state),
2. $o(L \setminus R) = G \setminus H$ and $o(R \setminus L) = H \setminus G$ (i.e., all objects of G are $\{\mathsf{destroyed}\}$ that match classifier roles of L not belonging to R and, symmetrically, all objects of H are $\{\mathsf{new}\}$ that match classifier roles in R not belonging to L).

That is, the transformation creates and destroys exactly what is specified by the rule and the occurrence. As a consequence, the rule together with the occurrence of the left-hand side L in the given graph G determines, up to renaming, the derived graph H, i.e., the approach has a clear operational interpretation, which is well-suited for visual programming.

In the more liberal, axiomatic interpretation, requirement 2 is weakened to

2′. $o(L \setminus R) \subseteq G \setminus H$ and $o(R \setminus L) \subseteq H \setminus G$ (i.e., *at least* the objects of G are $\{\mathsf{destroyed}\}$ that match classifier roles of L not belonging to R and, symmetrically, *at least* the objects of H are $\{\mathsf{new}\}$ that match classifier roles in R not belonging to L).

These so-called *graph transitions* [19] allow side effects not specified by the rule, like in the example of Fig. 4 where the amount of product pr changes without being explicitly rewritten by the rule. This is important for high-level modeling where specifications of behavior are often incomplete.

In both cases, instance transformations as well as specification-level rules are typed over the same type graph, and the occurrence homomorphism respects these types. This completes the instantiation of the type-specification-instance pattern for the aspect of state transformation.

Summarizing, a collaboration on the specification level represents a pattern for state transformations on the instance level, and the occurrence of this pattern requires, beside the structural match of the pre- and postconditions, (at least) the realization of the described effects. Graph transformations provide a formal model for the state transformation aspect which allows to describe the overall effect of a complex interaction. However, the interaction itself, which decomposes the global steps into more basic actions, is not modeled. In the next section, this finer structure shall be described in terms of the model of concurrency for graph transformation systems [4].

4 Interactions as Graph Processes

In this section, we shall extend the triangular type-specification-instance pattern to the interaction part. First, we describe the formalization of the individual concepts and then the typing and occurrence relations.

Class diagrams. A class diagram is represented as a *graph transformation system*, briefly GTS, $\mathcal{G} = \langle TG, \mathcal{R} \rangle$ consisting of a type graph TG and a set of transformation rules $\mathcal{R}$. The type graph captures the structural aspect of the class diagram, like the classes, associations, and attributes, as well as the types of call and return messages that are sent when an operation is invoked. For the fragment of the class diagram consisting of the classes Order, Customer, and Delivery, the customer association[2], and the attributes and operations of the first two classes, the type graph is shown in Fig. 5. Classes are as usually shown as rectangular, data types as oval shapes. Call and return messages are depicted like UML action states, i.e., nodes with convex borders at the two sides. Return messages are marked by overlined labels. Links from call message nodes represent the input parameters of the operation, while links from return message nodes point to output parameters (if any). The self link points to the object executing the operation. The rules of the GTS in Fig. 5 model different kinds of basic actions that are implicitly declared within the class diagram. Among them are state transformation actions like destroy o, control actions like send charge, and actions representing the execution of an operation like cu.charge(d).

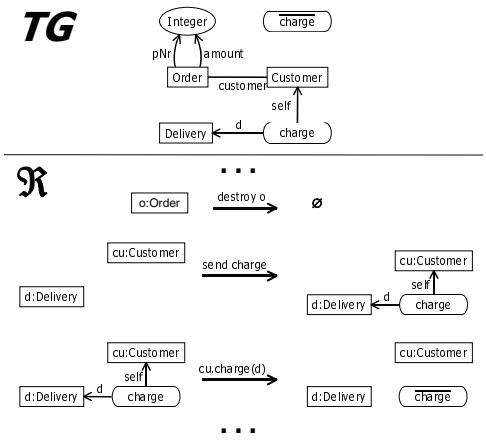


Fig. 5. Graph transformation system for a fragment of the class diagram in Fig. 2

Interactions. An interaction consists of a set of messages, linked by control and data flow, that stimulate actions like access to attributes, invocation of operations, creation and deletion of objects, etc. While the control flow is explicitly given by sequence numbers specifying a partial order over messages, data flow information is only implicitly present, e.g., in the parameters of operations and in their implementation, as far as it is given. However, it is important that control and data flow are compatible, i.e., they must not create cyclic dependencies.

[2] More precisely, in UML terms this is an unnamed association in which class Customer plays the role customer.

Such constraints are captured by the concept of a *graph process* which provides a partial order semantics for graph transformation systems.

The general idea of process semantics, which have their origin in the theory of Petri nets [28], is to abstract, in an individual run, from the ordering of actions that are not causally dependent, i.e., which appear in this order only by accident or because of the strategy of a particular scheduler. If actions are represented by graph transformation rules specifying their behavior in a pre/postcondition style, these causal dependencies can be derived by analyzing the intersections of rules in the common context provided by the overall collaboration.

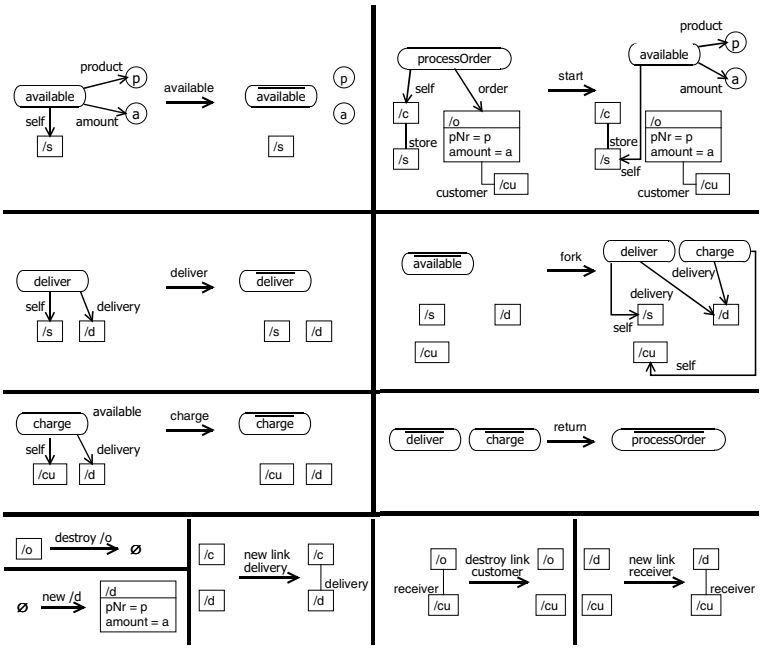


Fig. 6. Graph process for the collaboration diagram of operation processOrder. The three rules in the upper left section represent the operations, those in the upper right realize the control flow between these operations, and the five rules in the lower part are responsible for state transformations (note that attribute values a and p of /d can be instantiated by new /d since all the rules of the graph process act in a common context given by the collaboration)

The graph process for the collaboration diagram in the lower left of Fig. 3 is shown in Fig. 6. It consists of a set of rules representing the internal actions, placed in a common name space. That means, e.g., the available node created by the rule start in the top right is the same as the one deleted by the rule available in the top left. Because of this causal dependency, available has to be performed after start. Thus, the causality of actions in a process is represented by the overlapping of the left- and right-hand sides of the rules.

Graph processes are formally defined in three steps. A *safe graph transformation system* consists of a graph C (best to be thought of as the graph of the collaboration) together with a set of rules $\mathcal{T}$ such that, for every rule $t = G \to H \in \mathcal{T}$ we have $G, H \subseteq C$ (that is, C provides a common context for the rules in $\mathcal{T}$). Intuitively, the rules in $\mathcal{T}$ represent transformations, i.e., occurrences of rules. In order to formalize this intuition, the notion of *occurrence graph transformation system* is introduced requiring, in addition, that the transformations in $\mathcal{T}$ can be ordered in a sequence. That means, the system has to be acyclic and free of conflicts, and the causality relation has to be compatible with the graphical structure. In order to make this precise, we define the causal relation associated to a safe GTS $\langle C, \mathcal{T} \rangle$. Let $t : G \to H$ be one arbitrary transformation in $\mathcal{T}$ and e be any edge, node, or attribute in C. We say that

- t *consumes* e if $e \in G \setminus H$
- t *creates* e if $e \in H \setminus G$
- t *preserves* e if $e \in G \cap H$

The relation $\leq$ is defined on $\mathcal{T} \cup C$, i.e., it relates both graphical elements and operations. It is the transitive and reflexive closure of the relation $<$ where

- $e < t_1$ if t_1 consumes e
- $t_1 < e$ if t_1 creates e
- $t_1 < t_2$ if t_1 creates e and t_2 preserves e, or
  t_1 preserves e and t_2 consumes e

Now, a safe GTS is called an *occurrence graph transformation system* if

- the causal relation $\leq$ is a partial order which respects source and target, i.e., if $t \in \mathcal{T}$ is a rule and a vertex v of C is source (or target) of an edge e, then
  - $t \leq v$ implies $t \leq e$ and
  - $v \leq t$ implies $e \leq t$
- for all elements x of C, x is consumed by at most one rule in $\mathcal{T}$, and it is created by at most one rule in $\mathcal{T}$.

The objective behind these conditions is to ensure that each occurrence GTS represents an equivalence class of sequences of transformations "up-to-rescheduling", which can be reconstructed as the linearizations of the partial order $\leq$. Vice versa, from each transformation sequence one can build an occurrence GTS by taking as context C the colimit (sum) of all instance graphs in the sequence [4].

The causal relation between the rules in the occurrence GTS in Fig. 6 is visualized by the Petri net in the left of Fig. 7. For example, the dependency between available and start discussed above is represented by the place between the corresponding transitions. Of these dependencies, which include both control- and data-flow, in the UML semantics only the control-flow dependencies are captured by a precedence relation on messages, specified using sequence numbers. This part is presented by the sub-net in the upper right of Fig. 7. In comparison, the net in the lower right of Fig. 7 visualizes the control flow if we replace the

synchronous call to deliver by an asynchronous one: The call is delegated to a thread which consumes the return message and terminates afterwards. This strategy for modeling asynchronous calls might seem a little *ad-hoc*, but it follows the implementation of asynchronous calls in Java as well as the formalization of asynchronous message passing in process calculi [23]. The essential property is the independence of the deliver and the charge action.

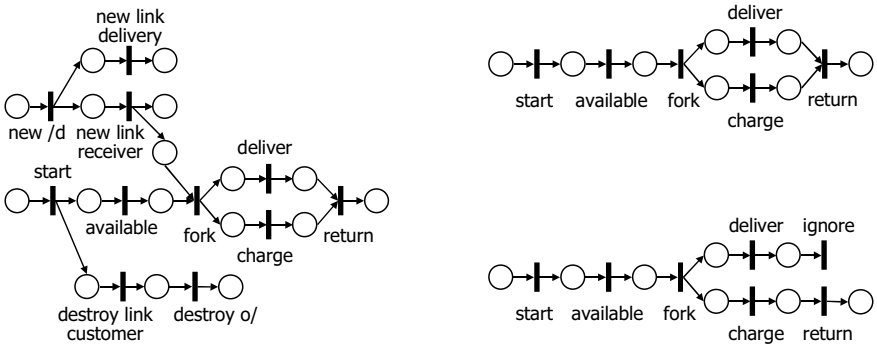


Fig. 7. Control flow and data dependencies of the occurrence GTS in Fig. 6 (left), sub-net for control flow dependencies of occurrence GTS (top right), and control flow dependencies with asynchronous call of deliver (bottom right)

We have used graph transformation rules for specifying both the overall effect of an interaction as well as its basic, internal actions. A fundamental fact about occurrence GTS [4] relates the state transformation with the interaction aspect: Given an occurrence GTS $\mathcal{O} = \langle C, \mathcal{T} \rangle$ and its partial order $\leq$, the sets of minimal and maximal elements of C w.r.t. $\leq$ form two graphs $Min(\mathcal{O}), Max(\mathcal{O}) \subseteq C$. This allows us to view a process p externally as a transformation rule $\tau(p)$, called *total rule of* p, which combines the effects of all the local rules of $\mathcal{T}$ in a single, atomic step. The total rule of the process in Fig. 6 is shown in the top of Fig. 4.

Summarizing, the three corners of our triangle are represented by a GTS representing the type level, and two occurrence GTS formalizing interactions on the specification and the instance level, respectively. It remains to define the relation between these three levels, i.e., the concepts of typing and occurrence.

Typing. In analogy with the typing of graphs, an occurrence GTS $\mathcal{O} = \langle C, \mathcal{T} \rangle$ is typed over a GTS $\mathcal{G} = \langle TG, \mathcal{R} \rangle$ via a *homomorphism of graph transformation systems*, briefly GTS morphism. A GTS morphism $p : \mathcal{O} \to \mathcal{G}$ consists of a graph homomorphism $c : C \to TG$ typing the context graph C over the type graph TG, and a mapping of rules $f : \mathcal{T} \to \mathcal{R}$ such that, for every $t \in \mathcal{T}$, the rules t and $f(t)$ are equal up to renaming. Such a *typed* occurrence GTS is called a *graph process* [4].

Since all graphs in the rules of Fig. 6 are well-typed over the type graph TG, their union C is typed over TG by the union of the typing homomorphisms of its

subgraphs. The rules representing basic actions, like operation invocations and state transformations, can be mapped directly to rules in $\mathcal{R}$. The control flow rules, which are more complex, are mapped to compound rules derived from the elementary rules in $\mathcal{R}$.[3]

Occurrence. The occurrence of a specification-level interaction pattern at the instance level is described by a plain homomorphism of graph transformation systems: We want the same granularity of actions on the specification and the instance level. An example of an instance-level collaboration diagram is given in Fig. 8. It does not contain additional actions (although this would be permit-

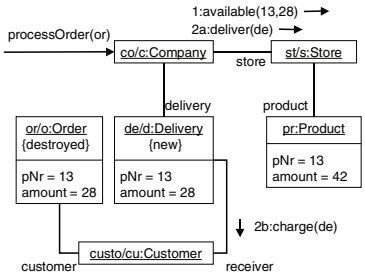


Fig. 8. Collaboration diagram on the instance level

ted by the definition of occurrence), but the additional context of the Product instance. In the process in Fig. 6, this would lead to a corresponding extension of the start rule.

As before, the GTS homomorphisms forming the three sides of the triangle have to be compatible. That means, an occurrence of a specification-level collaboration diagram in an instance-level one has to respect the typing of classes, associations, and attributes and of operations and basic actions.

This completes the formalization of the type-specification-instance triangle in the three views of collaboration diagrams of structure, state transformation, and interaction.

5 Conclusion

In this paper, we have proposed a semantics for collaboration diagrams based on concepts from the theory of graph transformation. We have identified and formalized three different aspects of a system model that can be expressed by collaboration diagrams (i.e., structure, state transformation, and interaction) and, orthogonally, three levels of abstraction (type, specification, and instance level). In particular, the idea of collaboration diagrams as state transformations

[3] Categorically, this typing of an occurrence GTS can be formalized as a Kleisli morphism mapping elementary rules to derived ones (see, e.g., [18, 17] for similar approaches in graph transformation theory).

provides new expressive power which has so far been neglected by the UML standard. The relationships between the different abstraction levels and aspects are described in terms of homomorphisms between graphs, rules, and graph transformation systems.

The next steps in this work consist in transferring the new insights to the UML specification. On the level of methodology and notation, the state transformation aspect should be discussed as one possible way of using collaboration diagrams. On the level of abstract syntax (i.e., the meta model) the pattern-occurrence relation between specification- and instance-level diagrams has to be made explicit, e.g., by additional meta associations. (In fact, this has been partly accomplished in the most recent draft of the standard [25].) On the semantic level, a representation of the causal dependencies in a collaboration diagram is desirable which captures also the data flow between actions.

It remains to state more precisely the relation between collaboration diagrams as defined by the standard and our extended version. Obviously, although our collaboration diagrams are syntactically legal, the collaboration diagrams that are semantically meaningful according to the UML standard form a strict subset of our high-level diagrams based on graph matching. An implementation of this matching by explicit navigation (as it is given, for example, in [14] as part of a code generation in JAVA) provides a translation back to the original low-level style. The formal properties of this construction have not been investigate yet.

References

1. Action Semantics Consortium. Precise action semantics for the Unified Modeling Language, August 2000. `http://www.kc.com/as_site/`.
2. P. Bottoni, M. Koch, F. Parisi Presicce, and G. Taentzer. Consistency checking and visualization of OCL constraints. In Evans et al. [13], pages 294–308.
3. D. Coleman, P. Arnold, S. Bodof, C. Dollin, H. Gilchrist, F. Hayes, and P. Jeremes. *Object Oriented Development, The Fusion Method*. Prentice Hall, 1994.
4. A. Corradini, U. Montanari, and F. Rossi. Graph processes. *Fundamenta Informaticae*, 26(3,4):241–266, 1996.
5. A. Corradini, U. Montanari, F. Rossi, H. Ehrig, R. Heckel, and M. Löwe. Algebraic approaches to graph transformation, Part I: Basic concepts and double pushout approach. In G. Rozenberg, editor, *Handbook of Graph Grammars and Computing by Graph Transformation, Volume 1: Foundations*, pages 163–245. World Scientific, 1997.
6. D. D'Souza and A. Wills. *Components and Frameworks with UML: The Catalysis Approach*. Addison-Wesley, 1998.
7. H. Ehrig, G. Engels, H.-J. Kreowski, and G. Rozenberg, editors. *Handbook of Graph Grammars and Computing by Graph Transformation, Volume 2: Applications, Languages, and Tools*. World Scientific, 1999.
8. H. Ehrig, G. Engels, H.-J. Kreowski, and G. Rozenberg, editors. *Proc. 6th Int. Workshop on Theory and Application of Graph Transformation (TAGT'98), Paderborn, November 1998*, volume 1764 of *LNCS*. Springer-Verlag, 2000.
9. H. Ehrig, H.-J. Kreowski, U. Montanari, and G. Rozenberg, editors. *Handbook of Graph Grammars and Computing by Graph Transformation, Volume 3: Concurrency and Distribution*. World Scientific, 1999.

10. H. Ehrig, M. Pfender, and H.J. Schneider. Graph grammars: an algebraic approach. In *14th Annual IEEE Symposium on Switching and Automata Theory*, pages 167–180. IEEE, 1973.
11. G. Engels, J.H. Hausmann, R. Heckel, and St. Sauer. Dynamic meta modeling: A graphical approach to the operational semantics of behavioral diagrams in UML. In Evans et al. [13], pages 323–337.
12. G. Engels, R. Hücking, St. Sauer, and A. Wagner. UML collaboration diagrams and their transformation to Java. In France and Rumpe [15], pages 473–488.
13. A. Evans, S. Kent, and B. Selic, editors. *Proc. UML 2000 – Advancing the Standard*, volume 1939 of *LNCS*. Springer-Verlag, 2000.
14. T. Fischer, J. Niere, L. Torunski, and A. Zündorf. Story diagrams: A new graph transformation language based on UML and Java. In Ehrig et al. [8].
15. R. France and B. Rumpe, editors. *Proc. UML'99 – Beyond the Standard*, volume 1723 of *LNCS*. Springer-Verlag, 1999.
16. M. Gogolla. Graph transformations on the UML metamodel. In J. D. P. Rolim et al., editors, *Proc. ICALP Workshops 2000, Geneva, Switzerland*, pages 359–371. Carleton Scientific, 2000.
17. M. Große-Rhode, F. Parisi Presicce, and M. Simeoni. Refinement of graph transformation systems via rule expressions. In Ehrig et al. [8], pages 368–382.
18. R. Heckel, A. Corradini, H. Ehrig, and M. Löwe. Horizontal and vertical structuring of typed graph transformation systems. *Math. Struc. in Comp. Science*, 6(6):613–648, 1996.
19. R. Heckel, H. Ehrig, U. Wolter, and A. Corradini. Double-pullback transitions and coalgebraic loose semantics for graph transformation systems. *Applied Categorical Structures*, 9(1), January 2001.
20. R. Heckel and St. Sauer. Strengthening the semantics of UML collaboration diagrams. In G. Reggio, A. Knapp, B. Rumpe, B. Selic, and R. Wieringa, editors, *UML'2000 Workshop on Dynamic Behavior in UML Models: Semantic Questions*, pages 63–69. October 2000. Tech. Report no. 0006, Ludwig-Maximilians-University Munich, Germany.
21. R. Heckel and A. Zündorf. How to specify a graph transformation approach: A meta model for FUJABA. In H. Ehrig and J. Padberg, editors, *Uniform Approaches to Graphical Process Specification Techniques, satellite workshop of ETAPS 2001, Genova, Italy*, 2001. To appear.
22. A. Knapp. A formal semantics of UML interactions. In France and Rumpe [15], pages 116–130.
23. M. Merro and D. Sangiorgi. On asynchrony in name-passing calculi. In *Proc. ICALP'98*, volume 1443 of *LNCS*, pages 856–867. Springer-Verlag, 1998.
24. Object Management Group. UML specification version 1.3, June 1999. `http://www.omg.org`.
25. Object Management Group. UML specification version 1.4beta R1, November 2000. `http://www.celigent.com/omg/umlrtf/`.
26. G. Övergaard. A formal approach to collaborations in the Unified Modeling Language. In France and Rumpe [15], pages 99–115.
27. V. Pratt. Modeling concurrency with partial orders. *Int. Journal. of Parallel Programming*, 15(1):33–71, February 1986.
28. W. Reisig. *Petri Nets*, volume 4 of *EATCS Monographs on Theoretical Computer Science*. Springer-Verlag, 1985.
29. A. Schürr, A.J. Winter, and A. Zündorf. The PROGRES approach: Language and environment. In Ehrig et al. [7], pages 487–550.

Dynamic Meta Modeling: A Graphical Approach to the Operational Semantics of Behavioral Diagrams in UML

Gregor Engels, Jan Hendrik Hausmann, Reiko Heckel, Stefan Sauer

University of Paderborn, Dept. of Mathematics and Computer Science
D-33098 Paderborn, Germany
engels|corvette|reiko|sauer@uni-paderborn.de

Abstract. In this paper, dynamic meta modeling is proposed as a new approach to the operational semantics of behavioral UML diagrams. The dynamic meta model extends the well-known static meta model by a specification of the system's dynamics by means of collaboration diagrams. In this way, it is possible to define the behavior of UML diagrams within UML.
The conceptual idea is inherited from Plotkin's structured operational semantics (SOS) paradigm, a style of semantics specification for concurrent programming languages and process calculi: Collaboration diagrams are used as deduction rules to specify a goal-oriented interpreter for the language. The approach is exemplified using a fragment of UML statechart and object diagrams.
Formally, collaboration diagrams are interpreted as graph transformation rules. In this way, dynamic UML semantics can be both mathematically rigorous so as to enable formal specifications and proofs and, due to the use of UML notation, understandable without prior knowledge of heavy mathematic machinery. Thus, it can be used as a reference by tool developers, teachers, and advanced users.

Keywords: UML meta model, statechart diagrams, precise behavioral semantics, graph transformation

1 Introduction

The UML specification [20] defines the abstract syntax and static semantics of UML diagrams by means of (meta) class diagrams and OCL formulas. The dynamic (operational) semantics of behavioral diagrams is only described informally in natural language. However, when using UML models for communication between development teams, for project documentation, or as a contract between developers and customers, it is important that all partners agree on a common interpretation of the language. This requires a semantics specification which captures, in a precise way, both the structural and the dynamic features of the language.

Another fundamental requirement for the specification of a modeling language is that it should be readable (at least) by tool developers, teachers, and advanced users. Only in this way, a common understanding of the semantics of the language can be developed among its users.

Presently, most approaches to dynamic UML semantics focus on the implementation and simulation of models, or on automatic verification and reasoning. Reggio et al. [23], for example, use algebraic specification techniques to define the operational semantics of UML state machines. Lillius and Paltor [17] formalize UML state machines in PROMELA, the language of the SPIN model checker. Knapp uses temporal logic [15] for formalizing UML interactions. Övergaard [21] presents a formal meta modeling approach which extends static meta modeling with a specification of dynamics by means of a simple object-oriented programming language that is semantically based on the π-calculus. The formalisms used in the cited approaches provide established technologies for abstract reasoning, automatic verification, execution, or simulation of models, but they are not especially suited for explaining the semantics to non-experts.

In contrast, the technique of meta modeling has been successful, because it does not require familiarity with formal notations to read the semantics specification. Our approach to UML semantics extends the static meta model based on class diagrams [20] by a dynamic model which is specified using a simple form of UML collaboration diagrams. The basic intuition is that collaboration diagrams specify the operations of a goal-driven interpreter. For instance, in order to fire a transition in a statechart diagram, the interpreter has to make sure to be in the source state of the transition, and it might have to ask for the occurrence of a certain trigger event. This trigger event may in turn depend on the existence of a link mediating a method call, invoked by the firing of a transition in another statechart diagram, etc. Conceptually, this may be compared to the behavior of a Prolog interpreter trying to find a proof for a given goal.

Despite the graphical notation, the specification is mathematically rigorous since collaboration diagrams are given a formal interpretation based on graph transformation rules (see, e.g., [24,6,7] for a recent collection of surveys and [1] for an introductory text) within our approach. In particular, they can be considered as a special form of graphical operational semantics (GOS) rules [4], a generalization of Plotkin's structured operational semantics (SOS) paradigm for the definition of (textual) programming languages [22] towards graphs.

The paper is organized as follows: The approach to dynamic meta modeling is exemplified using an important fragment of UML statechart and object diagrams which is introduced along with a sample model in Sect. 2. In Sect. 3, we introduce the structural part of our meta model, a fragment of the standard meta model with meta classes extended by meta operations. The semantics specification in terms of collaboration diagrams is presented in Sect. 4, and in Sect. 5 it is shown how this specification can be used to compute the behavior of the sample model introduced in Sect. 2. Finally, in Sect. 6 we summarize and outline some future perspectives.

2 Statechart and Object Diagrams

Our approach to dynamic meta modeling shall be exemplified by the operational semantics of UML statechart and object diagrams. Statechart diagrams are used to specify the local behavior of objects (of a certain class) during their lifetime. Similarly to an event-condition-action rule, a transition consists of a triggering event, an activation condition, and a list of actions. Additionally, we regard the invocation of operations on an object as well as the calls to operations of other objects by the object under consideration as particularly relevant for this purpose. Therefore, we restrict our specification to transitions with call events and/or call actions. Conditions, other kinds of events and actions, composite and pseudo states, as well as more advanced structural concepts like inheritance and composition of classes are not considered.

The considered model extract refers to a problem of general importance, since the life cycle description of objects in a statechart diagram has to be related to the messaging mechanisms between interacting objects and the invocation of methods on such objects. A recent solution [3] suggests to model dynamic behavior by state machines and to view methods as private virtual objects to allow for concurrent execution by delegation. In contrast, we propose dynamic meta modeling as a basis for an integration of events, messages, and method invocation. In the following, we present an example that will allow us to demonstrate the application of our approach.

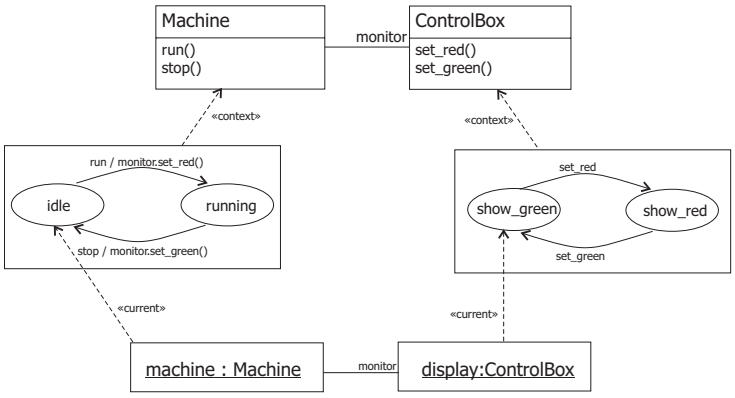


Fig. 1. A sample model (initial configuration)

Figure 1 shows a model consisting of two classes Machine and ControlBox related by an association stating that objects of class Machine may be monitored by objects of class ControlBox. In the Machine statechart diagram, transitions are labeled with combined event/action expressions like run/monitor.set_red(). That means, in order for the transition to fire, a call event for the operation run()

has to occur, and by firing the transition the method set_red() shall be called on the ControlBox object at the opposite end of the monitor link. As a result, the ControlBox object should change its state from show_green to show_red. No further actions are issued by the ControlBox statechart diagram. Notice that we do not model the implementation of operations. Therefore, the relevant interaction between objects (like switching the display by the machine) is described using call actions on the statechart level (rather than implementing it in the method run()).

The initial configuration of the system is given by an object diagram together with a specification of the control state of each object. In our example, machine is in state idle and display is in state show_green as shown in Fig. 1 by the stereotyped ≪current≫ relationships.

After presenting the static meta model and the firing rules of UML statechart diagrams in the next sections, we shall examine part of the life cycle of the objects introduced above.

3 Meta Classes and Meta Operations

In the UML semantics specification [20], the abstract syntax of statechart diagrams is specified by a meta class diagram. In order to define the structural model of an interpreter for this languages, this model has to be extended by state information, for example to represent the current control state of an object.

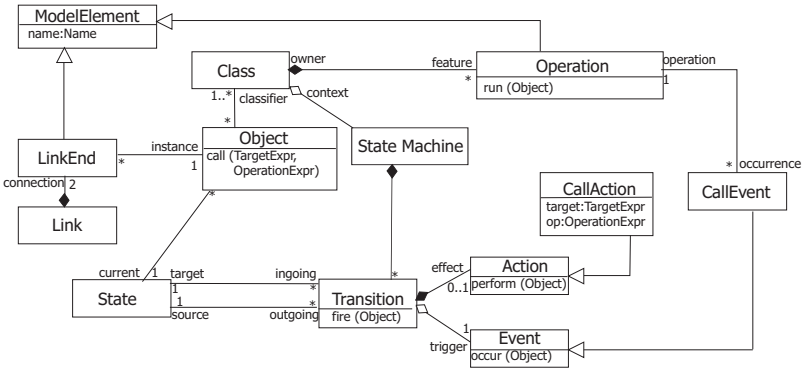


Fig. 2. Meta class diagram

Figure 2 shows the classes from the UML meta model that are relevant for the subclass of statechart diagrams we are considering (partly simplified by flattening the meta class hierarchy). A statechart diagram, represented by an instance of meta class StateMachine, controls the behavior of the objects of the class it is associated with. For this purpose, we extend the meta model by an association

current which designates the current control state of an object within the state diagram. States and transitions are represented by instances of the corresponding meta classes, and transitions are equipped with a trigger CallEvent (like run in our scenario) and an effect CallAction (like control.set_red()). A CallEvent carries a link to the local operation which is called. Unlike in the standard meta model, a CallAction is not directly associated with an operation, as this would result in static binding. Instead, an attribute OperationExpr is provided.

The state space of the diagrammatic language consists of all instance graphs conforming to the meta class diagram. Each instance graph represents the state of an interpreter given by the "programs" (e.g., statechart diagrams) to be executed, the problem domain objects with their respective data states (given, e.g., by the values of attributes and links), and their control states.

The relation between class and instance diagrams can be formally captured by the concept of *type* and *instance graphs* [5].[1] Given a type graph TG, representing a class diagram, a *TG-typed instance graph* consists of a graph G together with a typing homomorphism $g : G \to TG$ associating to each vertex and edge x of G its type $g(x) = t$ in TG. For example, the instance graph of the meta class diagram in Fig. 2 that represents the abstract syntax of the model in Fig. 1 is shown in Fig. 3.

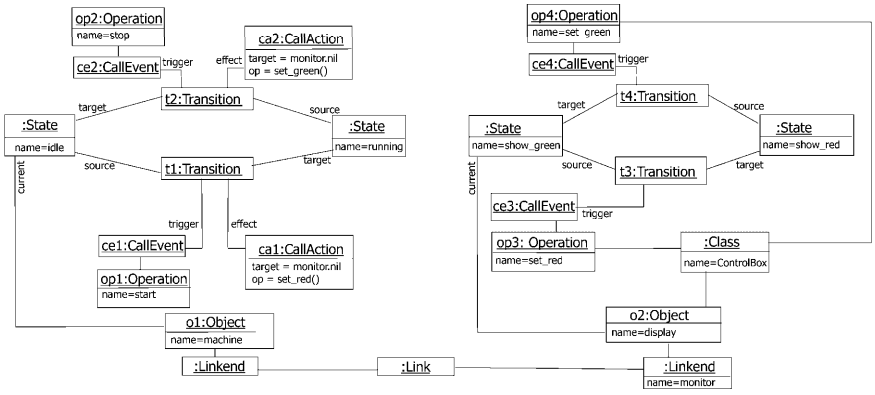


Fig. 3. Abstract syntax of sample model

The class diagram in Fig. 2 does not only contain meta classes and associations, but also *meta operations* like perform(Object) of class Action. They are the operations of our interpreter for statechart diagrams. Given the type graph

[1] By *graphs* we mean directed unlabeled graphs $G = \langle G_V, G_E, src^G, tar^G \rangle$ with set of vertices G_V, set of edges G_E, and functions $src^G : G_E \to G_V$ and $tar^G : G_E \to G_V$ associating to each edge its source and target vertex. A graph homomorphism $f : G \to H$ is a pair of functions $\langle f_V : G_V \to H_V, f_E : G_E \to H_E \rangle$ compatible with source and target.

TG representing the structural part of the class diagram, the meta operations form a family of sets $M = (MOP_w)_{w \in TG_V^+}$ indexed by non-empty sequences $w = v_1 \dots v_n$ of parameter class names $v_i \in TG_V$. By convention, the first parameter v_1 of each meta operation represents the class to which the operation belongs (thus there has to be at least one argument type). For example, the meta operation perform(Object) of class Action is formally represented as $\mathsf{perform} \in MOP_{\mathsf{Action,Object}}$.

After having described the abstract syntax of our model in terms of meta classes and meta operations, the implementation of the meta operations shall be specified using collaboration diagrams in the next section.

4 Meta Modeling with Collaboration Diagrams

The static meta model of the UML defines the abstract syntax of the language by means of meta class diagrams. Seen as a system specification, these class diagrams represent the structural model of an UML editor or interpreter. In this section, we shall extend this analogy to the dynamic part of a model, i.e., we are going to specify the dynamics of an interpreter for statechart and object diagrams. Interaction diagrams and, in particular, collaboration diagrams are designed to specify object interaction, creation, and deletion in a system model. Dynamic meta modeling applies the same language concepts to the meta model level to specify the interaction and dynamics of model elements of the UML.

The specification is based on the intuition of an interpreter which has to demonstrate the existence of a certain behavior in the model. Guided by a recursive set of rules stating the conditions for the execution of a certain meta operation, the interpreter works its way from a goal (e.g., the firing of a transition) towards its assumptions (e.g., the occurrence of a trigger event). The behavioral rules are specified by collaboration diagrams consisting of two compartments. The head of the diagram contains the meta operation which is specified by the diagram. The body specifies the assumptions for the execution of the meta operation, its effect on the object configuration, and other meta operations required.

For example, the conditions for a transition to fire and its effect on the configuration are specified in the collaboration diagram of Fig. 4: An object o may fire a transition if that object is in the corresponding source state, the (call) event triggering the transition occurs, and the operation associated with this call event is invoked by the meta operation run(o). In this case, the object o changes to the target state of the transition, which is modeled by the deletion and re-creation of the current link.

Thus, in order to be able to continue, the interpreter looks for a call event triggering the transition. This call event can be raised if the associated operation is called on the object o as specified in Fig. 5 using the meta operation call. The signature of this meta operation of meta class Object contains two parameters: The first one holds a path expression to direct the call to its target object (it equals nil when the target object is reached), and the second one specifies the

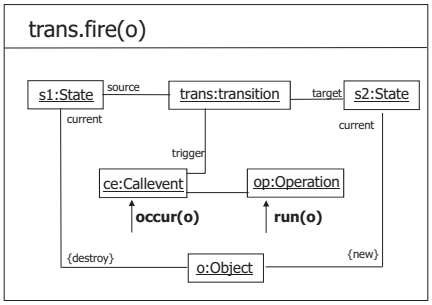


Fig. 4. The firing of a transition by an object

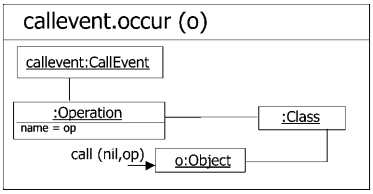


Fig. 5. Issuing a CallEvent

name of the operation to be called (and possibly further parameters). The name of the operation op has to match the operation expression transmitted by call.

Note that this does not guarantee the execution of the body of the called operation. In fact, no rule for meta operation run of meta class Operation is provided. The specification of the structure and dynamics of method implementations is the objective of *action semantics* as described by the corresponding request for proposals [18] by the Object Management Group. So far, UML provides only "uninterpreted strings" to capture the implementation of methods. We believe that our approach is extensible towards a dynamic semantics of actions once this is precisely defined.

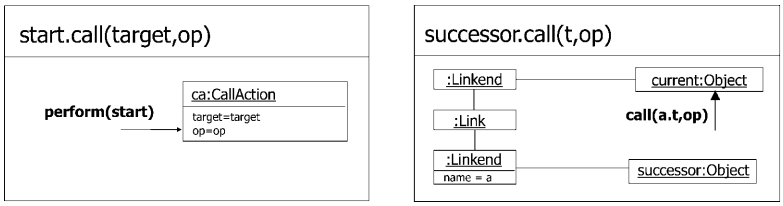


Fig. 6. Evaluating the target expression

An operation call like o.call(nil, op) in Fig. 5 originates from a call action which specifies by means of a path expression the target of the call. Thus, in order to find out whether a call is pending for a given object o, our interpreter has to check two alternatives: Either a call action is performed on o directly with target = nil, or there is a call at a nearby object with a target expression pointing towards o. These two cases are specified by the two collaboration diagrams for meta operation call in Fig. 7. The left diagram specifies the invocation of the meta operation by a CallAction on an object start. (The object is not depicted since it is given by the parameter of the premise.) Notice that the values of the meta attributes target and op have to match the parameters of meta operation call.

If the meta operation is not directly invoked by a call action, an iterative search is triggered as specified by the right diagram: To invoke the meta operation call(t,op) on an object successor which is connected to object current via a link, whose link end named a is attached to the successor object, the meta operation call(a.t,op) has to be invoked on current with the identical operation parameter op and the extended path expression a.t. (We assume target to be in a Java-like path syntax where the names of the links to be followed form a dot-separated list.)

Notice, that the right rule in Fig. 7 is potentially non-deterministic: In a state where the successor object has more than one incoming a link, different instantiations for the current object are possible. In this case, the link to be followed would be chosen non-deterministically.

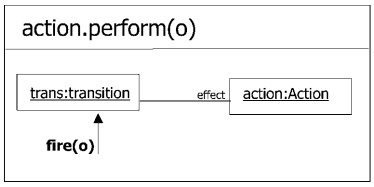


Fig. 7. The performing of an action by an object

Figure 7 presents the rule for performing an action. In our scenario this should be a CallAction initiating a call to another object, but the rule is also applicable to other kinds of actions. An action is the (optional) effect of firing a transition, i.e., the invocation of meta operation perform of meta class Action depends on the firing of the associated transition. Thus, the rule in Fig. 4 has to be applied again in order to derive the firing of the transition at the calling object.

As already mentioned in the introduction, this goal-oriented style of semantics specification is conceptually related to the proof search of a Prolog interpreter. This intuition is made precise by the paradigm of graphical operational semantics (GOS) [4], a graph-based generalization of the structured operational

semantics (SOS) paradigm [22], for the specification of diagram languages. In the GOS approach, deduction rules on graph transformations are introduced in order to to formalize the derivation of the behavior of models from a set of meta-level diagrams, which is implicitly present in this section. In the next section, we describe a simplified form of this approach especially tailored for collaboration diagrams.

5 Computing with Collaboration Diagrams

In the previous section, collaboration diagrams have been used to specify the firing rules of statechart transitions and the transmission of calls between objects. Now, concrete computations shall be modeled as collaboration diagrams on the instance level. This allows us to represent changes to the object structure together with the operations causing these changes. Moreover, even incomplete computations can be modeled, where some of the method calls are still unresolved. This is important if we want to give semantics to incomplete models like the one in Sect. 2 which requires external activation in order to produce any activity.

The transition from semantic rules to computations is based on a formal interpretation of collaboration diagrams as graph transformation rules. A rule representing the collaboration diagram for operation trans.fire(o) in Fig. 4 is shown in Fig. 8. It consists of two graphs L and R representing, respectively, the pre- and the post-condition of the operation. In general, both L and R are instances of the type graph TG representing the class diagram, and both are subgraphs of a common graph C that we may think of as the object graph of the collaboration diagram. Then, the pre-condition L contains all objects and links which have to be present before the operation, i.e., all elements of C except for those marked as {new} or {transient}. Analogously, the post-condition contains all elements of C not marked as {transient} or {destroy}. In the example of Fig. 8, graph C is just the union $L \cup R$ since there are no transient objects in the diagram of Fig. 4.

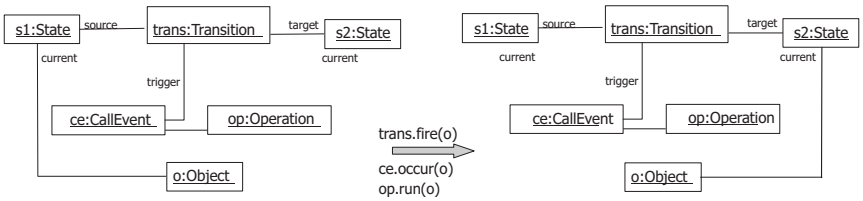


Fig. 8. Collaboration diagram as a labeled graph transformation rule

Besides structural modifications, the collaboration diagram describes calls to meta operations ce.occur(o) and op.run(o), and it is labeled by the operation trans.fire(o), the implementation of which it specifies. This information is

recorded in the rule-based presentation in Fig. 8 by means of additional labels above and below the arrow. Abstractly, a collaboration diagram is denoted as

$$C : L \xrightarrow[b]{a} R$$

where C is the object graph of the diagram, L and R are the pre- and post-conditions, a is the label representing the operation specified by the diagram, and b represents the sequential and/or concurrent composition of operations referred to (that is, called) within in the diagram. The expression ce.occur(o) × op.run(o) in Fig. 8, for example, represents the concurrent invocation of two operations.

We shall use the rule-based interpretation of collaboration diagrams in order to derive the behavior of the sample model introduced in Sect. 2. The idea is to combine the specification-level diagrams by means of two operators of *sequential composition* and *method invocation*. The sequential composition of two diagrams

$$C_1 : L_1 \xrightarrow[b_1]{a_1} R_1 \text{ and } C_2 : L_2 \xrightarrow[b_2]{a_2} R_2$$

is defined if the post-condition R_1 of the first equals the pre-condition L_2 of the second. The composed diagram is given by

$$C_1 \cup_{L_2=R_1} C_2 : L_1 \xrightarrow[b_1 ; b_2]{a_1 ; a_2} R_2$$

where $C_1 \cup_{L_2=R_1} C_2$ denotes the disjoint union of the graphs C_1 and C_2, sharing only $L_2 = R_1$. The second operator on diagrams models the invocation of a method from within the implementation of another method. This is realized by substituting the method call by the implementation of the called method, thus diminishing the hierarchy of method calls. Assume two rules

$$C : L \xrightarrow[b[c]]{a} R \text{ and } C' : L' \xrightarrow[d]{c} R'$$

where the call expression $b[c]$ of the first rule contains a reference to the operation c specified by the second rule. (In the rule of Fig. 8, $b[c]$ corresponds to ce.occur(o) × op.run(o), and c could be instantiated with either ce.occur(o) or op.run(o).) Then, the composed rule is given by

$$C \cup_c C' : L \cup_c L' \xrightarrow[b[d]]{a} R \cup_c R'.$$

The call to c is substituted by the expression d specifying the methods called within c. By $C \cup_c C'$ we denote the union of graphs C and C' sharing the *self* and parameter objects of the operation c.[2] In the same way, the pre- and post-conditions of the called operation are imported inside the calling operation.

[2] Notice that, in order to ensure that the resulting diagram is consistent with the cardinality constraints of the meta class diagram, it might be necessary to identify further elements of C and C' with each other (besides the ones identified by c). For instance, when identifying two transitions, we also have to identify the corresponding source and target states. Formally, this effect is achieved by defining the union as a pushout construction in a restricted category of graphs (see, e.g., [16]).

In Fig. 9 it is outlined how these two composition operators are used to build a collaboration diagram representing a possible run of our sample model. The given diagrams are depicted in iconized form with sequential composition and invocation as horizontal and vertical juxtaposition, respectively. This presentation is inspired by the *tile model* [12], a generalization of the SOS paradigm [22] towards open (e.g., incomplete) systems. In fact, in our example, such a semantics is required since the model in Fig. 1 is incomplete, i.e., it does not specify the source of the call events run and stop needed in order to trigger the machine's transitions.

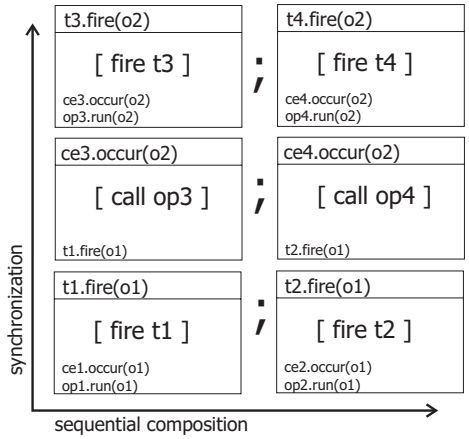


Fig. 9. Composing a run of the sample model

Figure 10 shows an expanded version of the iconized diagram *[fire t3]* in the top left of Fig. 9. It originates from an application of the operation trans.fire(o) in the context of an additional transition.[3] The diagrams *[fire t4]* to the right of *[fire t3]* as well as *[fire t1]* and *[fire t2]* in the bottom are expanded analogously.

Figure 11 details the icon labeled *[call op3]*. It shows the invocation of several operations realizing the navigation of the method call along the monitor link as specified by the target expression, and the issuing of the call event. A similar diagram could be drawn for *[call op4]*.

Finally, in Fig. 12 the composite computation is shown covering the complete scenario depicted in Fig. 1. It can be derived from the components in Fig. 9 in two different ways: by first synchronizing the single transitions (vertical dimension)

[3] In general, contextualization of rules has to be specified explicitly in our model (in this we follow the philosophy of the SOS and the tile framework [22,12]). In the present specification, however, we can safely allow to add any context but for the current links which ensure the coordinated behavior of the different statechart diagrams.

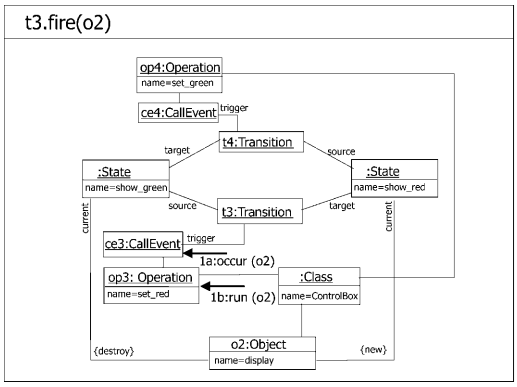


Fig. 10. Operation trans.fire(o) in context

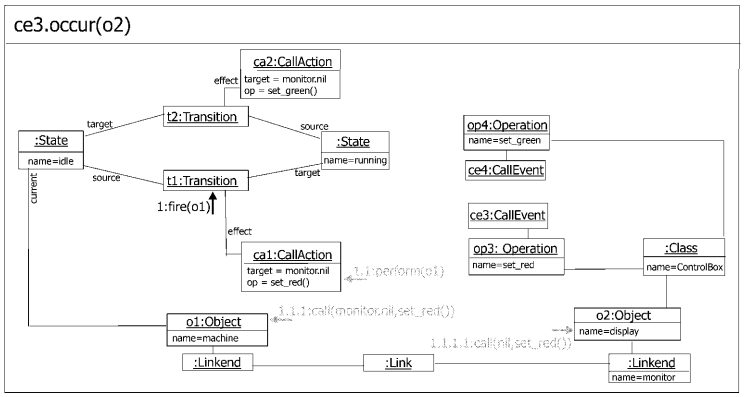


Fig. 11. Navigation of the method call

and then sequentially composing the two steps (horizontal dimension), or first building local two-step sequences (horizontal dimension) and then synchronizing them (vertical dimension).

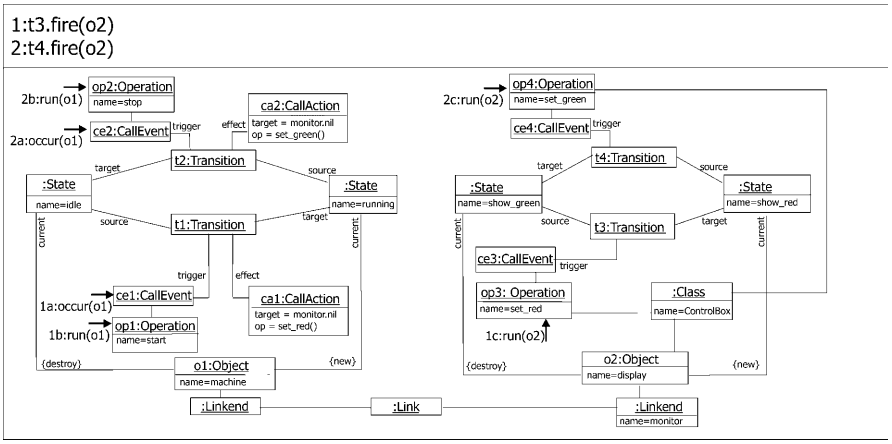


Fig. 12. Composite rule for the scenario in Fig. 1

6 Conclusion

In this paper, we have proposed the use of collaboration diagrams formalized as graph transformation rules for specifying the operational semantics of diagram languages. The concepts have been exemplified by a fragment of a dynamic meta model for UML statechart and object diagrams.

The fragment should be extended to cover a semantically complete kernel of the language which can be used to define more specific, derived modeling concepts. This approach is advocated by the *pUML* group (see e.g., [9]). Concrete examples how to define such a mapping of concepts include the flattening of statecharts by means of graph transformation rules [13] and the simplification of class diagrams [14] by implementing inheritance in terms of associations.

Our experience with specifying a small fragment of UML shows that tool support is required for testing and animating the specification. While the implementation of flat collaboration diagrams is reasonably well understood (see, e.g., [8,10]), the animation of the results of an execution on the level of concrete syntax is still under investigation. It requires a well-defined mapping between the concrete and the abstract syntax of the modeling language. One possible solution is to complement the graph representing the abstract syntax by a *spatial relationship graph*, and to realize the mapping by a graphical parser specified by a graph grammar [2].

A related problem is the integration of model execution and animation in existing UML tools. Rather than hard-coding the semantics into the tools, our approach provides the opportunity to allow for *user-defined semantics*, e.g., in the context of domain-specific profiles. Such a profile, which extends the UML standard by stereotypes, tagged values, and constraints [19], could also be used to implement the extensions to the static meta model that are necessary in order to define the operational semantics (e.g., the current links specifying the control states of objects could be realized as tagged values).

On the more theoretical side, the connection of dynamic meta modeling with proof-oriented semantics following the SOS paradigm allows the transfer of concepts of the theory of concurrent languages, like bisimulation, action refinement, type systems, etc. Like in the GOS approach [4], the theory of graph transformation can provide the necessary formal technology for transferring these concepts from textual to diagram languages.

References

1. 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. *Science of Computer Programming*, 34:1–54, 1999. 324
2. R. Bardohl, G. Taentzer, M. Minas, and A. Schürr. Application of graph transformation to visual languages. In Ehrig et al. [6], pages 105–180. 335
3. R. Breu and R. Grosu. Relating events, messages, and methods of multiple threaded objects. *JOOP*, pages 8–14, January 2000. 325
4. A. Corradini, R. Heckel, and U. Montanari. Graphical operational semantics. In A. Corradini and R. Heckel, editors, *Proc. ICALP2000 Workshop on Graph Transformation and Visual Modelling Techniques, Geneva, Switzerland*, Geneva, July 2000. Carleton Scientific. 324, 330, 336
5. A. Corradini, U. Montanari, and F. Rossi. Graph processes. *Fundamenta Informaticae*, 26(3,4):241–266, 1996. 327
6. H. Ehrig, G. Engels, H.-J. Kreowski, and G. Rozenberg, editors. *Handbook of Graph Grammars and Computing by Graph Transformation, Volume 2: Applications, Languages, and Tools*. World Scientific, 1999. 324, 336
7. H. Ehrig, H.-J. Kreowski, U. Montanari, and G. Rozenberg, editors. *Handbook of Graph Grammars and Computing by Graph Transformation, Volume 3: Concurrency and Distribution*. World Scientific, 1999. 324
8. G. Engels, R. Hücking, St. Sauer, and A. Wagner. UML collaboration diagrams and their transformation to Java. In R. France and B. Rumpe, editors, *Proc. UML'99 Int. Conference, Fort Collins, CO, USA*, volume 1723 of *LNCS*, pages 473–488. Springer Verlag, October 1999. 335
9. A. Evans and S. Kent. Core meta modelling semantics of UML: The pUML approach. In France and Rumpe [11], pages 140–155. 335
10. 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, editors, *Proc. 6th Int. Workshop on Theory and Application of Graph Transformation (TAGT'98), Paderborn, November 1998*, volume 1764 of *LNCS*. Springer Verlag, 2000. 335

11. R. France and B. Rumpe, editors. *Proc. UML'99 – Beyond the Standard*, volume 1723 of *LNCS*. Springer Verlag, 1999. 336, 337, 337
12. F. Gadducci and U. Montanari. The tile model. In G. Plotkin, C. Stirling, and M. Tofte, editors, *Proof, Language and Interaction: Essays in Honour of Robin Milner*. MIT Press, 1999. 333, 333
13. M. Gogolla and F. Parisi-Presicce. State diagrams in UML – a formal semantics using graph transformation. In *ICSE'98 Workshop on Precise Semantics of Modelling Techniques*, 1998. Tech. Rep. TUM-I9803, TU München. 335
14. M. Gogolla and M. Richters. Equivalence rules for UML class diagrams. In P.-A. Muller and J. Bezivin, editors, *Proc. UML'98 Workshop*, pages 86–97. Universite de Haute-Alsace, Mulhouse, 1998. 335
15. A. Knapp. A formal semantics of UML interactions. In France and Rumpe [11], pages 116–130. 324
16. M. Korff. Single pushout transformations of equationally defined graph structures with applications to actor systems. In *Proc. Graph Grammar Workshop, Dagstuhl, 1993*, volume 776 of *LNCS*, pages 234–247. Springer Verlag, 1994. 332
17. J. Lillius and I. Paltor. Formalising UML state machines for model checking. In France and Rumpe [11], pages 430–445. 324
18. Object Management Group. Action semantics for the UML, November 1998. `http://www.omg.org/pub/docs/ad/98-11-01.pdf`. 329
19. Object Management Group. Analysis and design platform task force – white paper on the profile mechanism, April 1999. `http://www.omg.org/pub/docs/ad/99-04-07.pdf`. 336
20. Object Management Group. UML specification version 1.3, June 1999. `http://www.omg.org`. 323, 324, 326
21. G. Övergaard. Formal specification of object-oriented meta-modelling. In T. Maibaum, editor, *Fundamental Approaches to Software Engineering (FASE'00), Berlin, Germany*, number 1783 in LNCS, pages 193–207. Springer Verlag, March/ April 2000. 324
22. G. Plotkin. A structural approach to operational semantics. Technical Report DAIMI FN-19, Aarhus University, Computer Science Department, 1981. 324, 331, 333, 333
23. G. Reggio, E. Astesiano, C. Choppy, and H. Hussmann. Analysing UML active classes and associated state machines – a lightweight formal approach. In T. Maibaum, editor, *Fundamental Approaches to Software Engineering (FASE'00), Berlin, Germany*, number 1783 in LNCS, pages 127–146. Springer Verlag, March/ April 2000. 324
24. G. Rozenberg, editor. *Handbook of Graph Grammars and Computing by Graph Transformation, Volume 1: Foundations*. World Scientific, 1997. 324

Dynamic Meta Modeling with time: Specifying the semantics of multimedia sequence diagrams

Jan Hendrik Hausmann, Reiko Heckel, Stefan Sauer

University of Paderborn, Institute for Computer Science, D 33095 Paderborn, Germany
E-mail: {hausmann,reiko,sauer}@upb.de

Received: 7 February 2003/Accepted: 14 May 2003
Published online: 1 April 2004 –

Abstract. The Unified Modeling Langugage (UML) offers different diagram types to model the behavior of software systems. In some domains like embedded real-time systems or multimedia systems, it is necessary to include specifications of time in behavioral models since the correctness of these applications depends on the fulfillment of temporal requirements in addition to functional requirements. UML thus already incorporates language features to model time and temporal constraints. Such model elements must have an equivalent in the semantic domain.

We have proposed Dynamic Meta Modeling (DMM), an approach based on graph transformation, as a means for specifying operational semantics of dynamic UML diagrams. In this article, we extend this approach to also account for time by extending the semantic domain to timed graph transformation. This enables us to define the operational semantics of UML diagrams with time specifications. As an example, we provide semantics for special sequence diagrams from the domain of multimedia application modeling.

Keywords: Formal semantics – Meta modeling – UML extensions – Graph transformation – Time – Multimedia – Sequence diagram

1 Introduction

The key objective of modeling is to create a representation of reality or ideas that abstracts from unnecessary details and concentrates on the main concepts. When designing software systems, time aspects are often (although sometimes unreasonably) considered a minor requirement and are thus not represented in the models. Consequently, the core diagrams of UML [18] – the standard language for building visual models of software systems – focus on structure, function, and dynamics of systems, but not on temporal aspects.

While this approach is adequate for example in the construction of business software (where temporal requirements typically concern efficiency, which mostly depends on the underlying hardware, system software, and database systems), temporal behavior is a key feature in other domains, and it has to be represented in the model in a precise way. Embedded real-time systems and multimedia applications are the most prominent among these domains. Real-time systems require that the results of a computational task are available within a limited period of time. In multimedia applications, timing requirements especially refer to the processing and synchronization of continuous media objects and the related quality of service (QoS).

The UML already provides some syntactic elements to express temporal behavior, like send and receive times of messages and duration of intervals on sequence diagrams or firing times for transitions in statecharts. These elements may be used to formulate timing constraints, i.e., time expressions on stimuli, message, or transition names. Further elements are being introduced by UML profiles like the *UML Profile for Schedulability, Performance, and Time* [17], which is motivated by the domain of embedded real-time systems. Its temporal modeling more fundamentally deals with time and time values, time-related events and stimuli, timing mechanisms like timers and clocks, and timing services.

Yet, while the semantics of the frequently used core elements of the UML is only partly understood, the interpretation of time-related modeling concepts in UML is even more ambiguous. The real-time profile [17] adds some detail in this regard, but still lacks a precise and formal semantics.

We can thus identify both a strong need for the precise specification of temporal behavior and a lack of concepts in the UML to meet this demand.

Existing approaches for real-time system specification are mainly motivated by the need to analyze (i.e., test or verify) systems with respect to their fulfillment of temporal properties. These approaches generally use time constraints to prescribe temporal requirements for a system. These constraints can be modeled in UML sequence diagrams. The operational execution of a system is rather described by an automata-based model, e.g. a state machine with timed events. It can then be tested or verified whether the state machine model or an implementation conforms to the timing constraints specified in the sequence diagram. Examples for this can be found in [13] where statecharts are extended by information of worst-case execution times derived from an actual implementation. Other approaches check whether an implementation satisfying all constraints may actually exist by using model checkers [1, 5] or systems of linear inequalities [14]. [1] defines an operational semantics of a real-time extended subset of UML statecharts by translating them to UPPAAL timed automata [15] and model-checking them.

In contrast to the aforementionend translation approaches, we present an approach to the operational semantics of UML diagrams in this article that resides on a higher level of abstraction, disregarding the need to be familiar with mathematical formalisms or model-checker languages. It incorporates a notion of time, thus enabling precise interpretation of models with temporal information. The approach consists of specifying an abstract interpreter for the behavioral diagrams of interest. For this purpose, diagrams are represented as instances of a meta model (i.e., as object diagrams, formally regarded as attributed graphs) which extends the UML meta model by representations of runtime state information. The steps of the interpreter are specified by graph transformation rules which manipulate the runtime state information to model the execution of the diagram. This approach to Dynamic Meta Modeling (DMM) has been successfully applied to statechart and sequence diagrams [2, 9]. Note that the use of graph transformation rules is different from the graph-grammar based transformation presented in [19]. There, UML models that are annotated with performance information (like execution time) are translated into a stochastic performance model by means of graph-grammar productions.

In order to account for the time aspect, we extend the formal foundation of the DMM approach from attributed to timed graph transformation systems [7]. Consequently, the approach is called *Dynamic Meta Modeling with time* (*DMM+t*). Following a formal introduction to the approach in Sect. 2, Sect. 3 presents – as a case study for DMM+t – multimedia sequence diagrams. They are a specialization of UML sequence diagrams for modeling multimedia applications and have been proposed as part of the OMMMA approach towards object-oriented modeling of multimedia applications based on UML (see [4, 20] for details). The operational semantics of these diagrams is defined using DMM+t in Sect. 4, where we also show how the formal semantics can be employed to gain additional information on the example under consideration.

A careful review of the achieved results in Sect. 5 reveals that in special cases the semantics definition yields results that do not correspond to the intuitive concepts. Two different kinds of strengthening the semantics are introduced that can be used to eliminate these unwanted phenomena. Section 6 concludes the presentation and points out possibilities for future work.

A preliminary version of this work has been presented at the International Workshop on Graph Transformation and Visual Modeling Techniques (GTVMT 2002) in Barcelona [10].

2 Dynamic Meta Modeling with time

While textual programming languages are defined by means of grammars and abstractly represented by terms or trees, the UML is defined by its meta model [18], i.e., a class diagram augmented with constraints, whose instances represent individual UML models. Interpreting these instances as attributed graphs, it is natural to use graph transformations to specify the manipulation and execution of diagrams. The approach of Dynamic Meta Modeling (DMM) [2, 9] uses rule-based graph transformations, denoted as UML collaborations, to specify abstract interpreters for dynamic sub-languages of the UML, and thus provides an operational semantics.

In order to provide semantics to modeling techniques with time, like multimedia sequence diagrams, the representation of time in graph transformation systems has been studied in [7]. Instead of introducing time as a separate semantic concept, the approach models time by means of time-valued attributes representing logical clocks. This bears the advantage that different aspects and properties of time can be modeled, depending on the strategies according to which time values are assigned and updated.

2.1 Graph transformation

Dynamic Meta Modeling with time (DMM+t) is based on typed and attributed graphs which are represented as UML class and object diagrams. Graph transformation is defined according to the algebraic approach, which is given a set-theoretic description in [7]. We denote rules by $p : L \to R$ and transformation steps by $G \overset{p(o)}{\Longrightarrow} H$, where p is the rule and o its occurrence, and consider sequences $G_0 \overset{p_1(o_1)}{\Longrightarrow} \cdots \overset{p_n(o_n)}{\Longrightarrow} G_n$ of transformations up to permutation of independent steps.

To be more precise, a notion of equivalence is defined on transformation sequences which considers two

sequences as equivalent if they can be obtained from each other by repeatedly swapping independent transformation steps. This equivalence has been formalized by the notion of *shift-equivalence* [12], and it is based on the notion of *independence* of graph transformations: two transformations $G \overset{p_1(o_1)}{\Longrightarrow} X \overset{p_2(o_2)}{\Longrightarrow} H$ are *sequentially independent* if the occurrences $o_1(R_1)$ of the right hand side of p_1 and $o_2(L_2)$ of the left hand side of p_2 do only overlap in objects of X that are preserved by both steps, formally $o_1(R_1) \cap o_2(L_2) \subseteq o_1(L_1 \cap R_1) \cap o_2(L_2 \cap R_2)$. Otherwise, there exists a *causal dependency* between the two steps forcing their application in the given order: either the match $o_2(L_2)$ of the second step contains vertices or edges created by the first step, or the second step removes vertices or edges that have been part of the match $o_1(L_1)$ of the first.

Two alternative transformations $G \overset{p_1(o_1)}{\Longrightarrow} H_1$ and $G \overset{p_2(o_2)}{\Longrightarrow} H_2$ are *parallel independent* if the occurrence $o_1(L_1)$ of the left hand side of p_1 in G is preserved by the application of p_2, and vice versa. Otherwise the two steps are *in conflict*.

While sequential independence allows consecutive transformations to be swapped, parallel independence allows alternative transformations to be scheduled in any order with the same result. The semantic idea behind these notions is expressed in the local Church–Rosser Theorem [3].

2.2 Graph transformation with time

Next we review the basic concepts of graph transformation with time, following [7]. To incorporate time into graph transformation with attributes, we generalize the approach of TER nets [8]. TER nets are high-level Petri nets that model time as a token attribute. Therefore, a time data type is required as a domain for time-valued attributes.

We model time by means of logical clocks represented by special attributes of a *time data type* $T = \langle D_{time}, +, 0, \geq \rangle$, i.e., an algebraic structure where $\geq$ is a partial order with 0 as its least element, $\langle +, 0 \rangle$ forms a monoid (that is, $+$ is associative with neutral element 0), and $+$ is monotonic wrt. $\geq$. Obvious examples include natural or real numbers with the usual interpretation of the operations, but not dates in the YY:MM:DD format (due to the Y2K problem).

A *graph with time* over a given time data type T is a graph in which all vertices are attributed with a special attribute *chronos* of type T. This attribute represents the state of the local clock of the object. Graph transformation rules with time $p : L \to R$ are just pairs of graphs with time as introduced above that respect the particular properties of time. This is expressed in the following axioms.

1. Local monotonicity: for all vertices $x \in L$ and $y \in R$: $x.chronos \leq y.chronos$, and
2. Uniform time stamps: for all vertices $x, y \in R$: $x.chronos = y.chronos$.

These axioms ensure a behavior of time which can be described informally as follows: according to axiom 1 an operation or step specified by a rule cannot take negative time, i.e., it cannot decrease the chronos values of the nodes it is applied to. It is, however, permitted to take zero time. If this option seems too idealistic, the zero case can be excluded without affecting the results of this article.

Axiom 2 states an assumption about atomicity of rule application, that is, all effects specified in the right hand side are observed at the same time, called the *firing time* of the rule application. Hence, it is guaranteed that the chronos value of each object always represents the last point in time when the object took part in a rule application (thus the last time it had an externally visible behavior).

In this case, one can show in analogy with TER nets that for each transformation sequence s using only rules that satisfy the above two conditions, there exists an equivalent sequence s' such that s' is time-ordered, that is, time is monotonically non-decreasing as the sequence advances. Thereby we obtain the behavior of a fully synchronized system with global time by strictly local means.

Theorem 1 (global monotonicity [7]**).** *For every transformation sequence s using only rules that satisfy axioms 1 and 2 above, there exists an equivalent sequence $s' = G_0 \overset{p_1(o_1),t_1}{\Longrightarrow} \dots \overset{p_n(o_n),t_n}{\Longrightarrow} G_n$ such that s' is time-ordered, that is, $t_i \leq t_{i+1}$ for all $i \in \{1, \dots, n-1\}$.*

The abstract interpreter specified in this manner by a set of graph transformation rules with time is mathematically represented as a rewrite relation over instance graphs $G \overset{p(o),t}{\Longrightarrow} H$ labeled with occurrences of rules $p(o)$ and their firing times t. In addition, we specify a set of *terminal instance graphs* to distinguish successful termination from deadlock. Then, a *trace of the interpreter* is a sequence of transformation steps ending in a terminal state corresponding to a terminal instance graph. Since we do not want to distinguish different interleavings of concurrent actions, we consider such traces up to *shift-equivalence*. Theorem 1 ensures that one time-ordered representative exists in every equivalence class of traces.

The rewrite relation defined above also induces a notion of equivalence on graphs: two graphs are equivalent if they are reducible to the same sets of terminal graphs. (Note that there may be more than one terminal instance graph reachable from a given graph because the rewrite relation is, in general, non-deterministic.) This equivalence can be used to define a notion of semantic equivalence on UML diagrams, even if they are syntactically different.

We use this model in Sect. 4 to specify an abstract interpreter for multimedia sequence diagrams. Given a set of individual diagrams as input to this interpreter, we can

test under which conditions a scenario executes successfully (by reaching a terminal state) and whether two given scenarios are equivalent (if they always produce equivalent traces or end up in the same terminal states).

3 Specifying time in multimedia with UML

Temporal relationships between elements of media presentations are the key characteristics of multimedia applications. The behavioral model of an interactive multimedia application has to account for both the timed and synchronized rendering of predefined scenes and the alteration of the course of presentation caused by user interaction. In the OMMMA approach, we deploy *multimedia sequence diagrams* (in the following: MM sequence diagrams), which are extended UML sequence diagrams, to model the former and UML statecharts to model the latter (the details of these modeling views and their integration can be found in [20]). Within this article, we concentrate on the representation of time in MM sequence diagrams to explain the DMM+t approach to formal semantics of UML with time.

We choose an example from a cinema application to illustrate our approach. In a cinema, typically there is a break to sell ice cream before the feature movie starts. We model this situation as a scene using an MM sequence diagram (see Fig. 1). While the ice cream is being sold, an advertisement slide is presented for 200 seconds. A sound clip announcing new products or special offers (Intro) followed by some "appetite-inducing" music is being played while the vendors sell the ice cream. The intro does not have a fixed length since it is subject to frequent change; the background music can be played indefinitely. The music is stopped when the movie is about to start.

The special elements used in this kind of multimedia modeling can be explained in natural language:

- All objects appearing at the top of a MM sequence diagram are *application objects*. They have the ability to render the content of some kind of media object. Every application object contains the methods start and stop which control the rendering of the media. Additional methods for pausing and re-synchronization purposes are not used in this article. Application objects are independent in their timekeeping, following the principles of distributed multimedia systems [16]. That means neither a central controller nor a global clock can be assumed.
- The boxes on the lifelines of the application objects are *presentations*. These elements replace the UML construct *activation*. A presentation represents the rendering of a media element by an application object. The name of the media object is given inside of the presentation's box. A presentation may furthermore define constraints on the minimal and maximal length of the media rendering. This means that shorter media elements would remain visible/audible even though their duration is over (e.g. static media elements like the slide have a duration of 0, but need to be shown for some time) and that long (possibly infinite) media objects (e.g. streams) can be limited. These features are needed because a scene (as described by the MM sequence diagram) does not require all media elements to have a known and fixed duration. MM sequence diagrams can thus be compared to higher-level and role-based interaction diagrams that are typically provided in the analysis phase of a software development.
- Attached to a presentation are incoming and outgoing messages. Incoming messages aligned with the top of a presentation box are messages calling the predefined method start (startmessages), incoming messages aligned with the bottom of a presentation box are stopmessages. Outgoing messages start or stop other presentations in synchronization with the current presentation. Outgoing messages can be synchronized either with the start of a presentation (starting at the top of the sending presentation box), the end of the presentation (starting at the bottom of the box), or they are sent out with a certain delay after the start of the presentation (starting at the side of the box with the specification of the delay attached). For instance, the message to start the audio playback in the example is timed to happen 5 seconds after the presentation of the slide started.

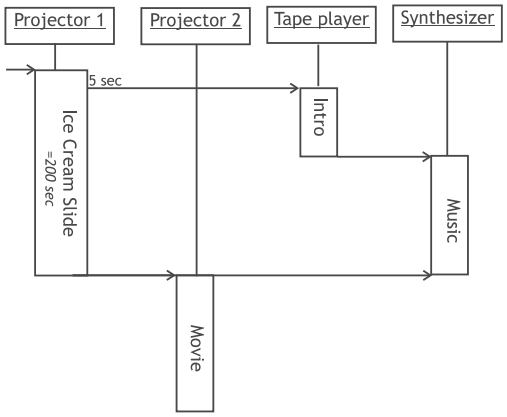


Fig. 1. Example scene as a multimedia sequence diagram

In Sect. 4.2 these concepts will be formalized using DMM+t rules. Like every semantics definition they are based on the abstract rather than the concrete syntax of the language. Therefore, the meta model defining the new language elements for MM sequence diagrams is given in Fig. 2.

The most prominent new feature in the meta model is the data type Time. This data type may contain positive integer values and the special value unlim denoting an unlimited time. It is used throughout the meta model to specify timepoints (e.g. Message.timestamp or Presentation.endtime) or the length of time intervals

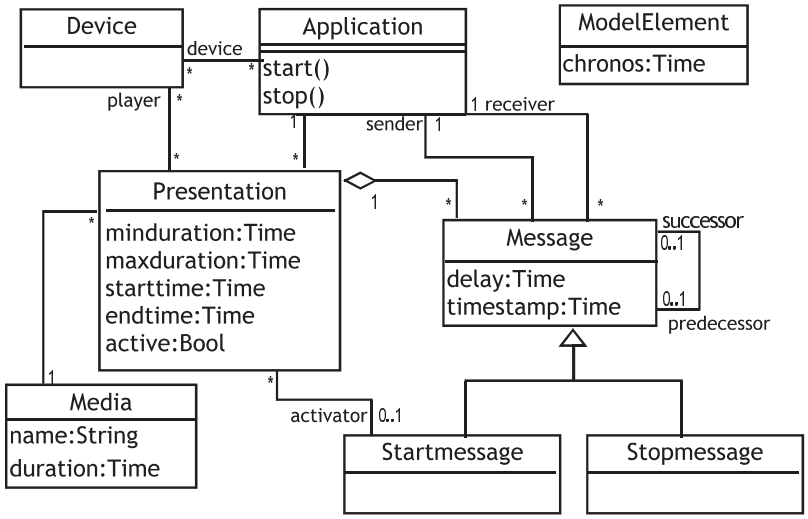
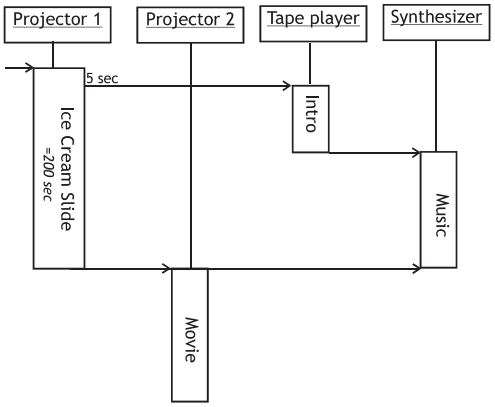


Fig. 2. Meta model of the MM sequence diagram



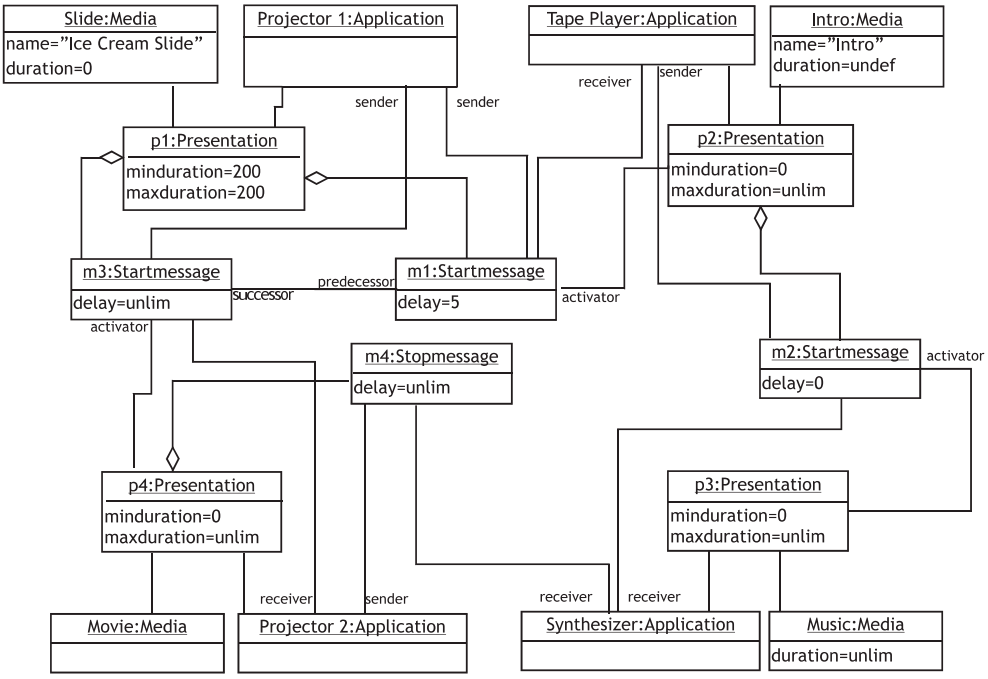


Fig. 3. Abstract syntax of the example diagram

(e.g. Media.duration). As every model element must have a chronos attribute (to comply with the axioms), it was added to the topmost class of the UML meta model hierarchy ModelElement. While a presentation uses the symbol of the UML element activation (which has no representation in the abstract syntax), it corresponds to the meta model element Presentation that contains the attributes minduration and maxduration. The associations between the incoming and outgoing messages in standard UML have consequently been adapted to this new element. The activator is now the message that starts a presentation, and all resulting outgoing messages are owned by the presentation. The order of these messages is still determined by the successor/predecessor relation. If message delays are specified, they must not contradict this order. Attributes like presentation.active or message.timestamp represent runtime information necessary for the interpretation of the operational semantics rules. A Device is the representation of a physical rendering facility suitable for the media type (this is not elaborated here).

The representation of the example diagram of Fig. 1 according to this meta model is shown in Fig. 3. Additional information in the meta model representation is the duration of the media objects (formerly only given in the text) and the assumption of default values for constraints of presentations: minduration is by default 0, maxduration is by default unlim. Everything else is just a representation of the information present in the concrete syntax diagram of Fig. 1.

Next we show how the techniques formally introduced in Sect. 2 can be used to formalize the semantics of MM sequence diagrams.

4 Semantics of multimedia sequence diagrams

This section contains the DMM+t rules for the MM sequence diagram example. The section consists of three parts: first, we introduce the rule notation as it appears in the figures, then we define the set of DMM+t rules for MM sequence diagrams, and finally we show how these rules support a precise interpretation of the model.

4.1 Format of the DMM+t rules

DMM+t rules as introduced in Sect. 2 are represented as pairs of UML collaboration diagrams on the level of classifier roles (in contrast to the instance level) where the role name following the slash symbol is optional and only shown if it is needed for binding and the name of the base classifier follows the colon symbol in the name string (see [18, pp. 3–124ff]). The collaboration diagrams are extended by attribute conditions (given in OCL). The left hand side diagram denotes the pre- and the right hand side diagram the post-conditions of the transformation. Since all rules here should conform to the axioms defined in Sect. 2, we introduce a special variable firingtime. This variable represents the chronos value that is associated with the rule's invocation. Its possible values are defined in the preconditions section of the rule. The execution mechanism of the rule has to ensure that

- the chronos attributes of all objects on the left hand side of the rule have a lower or equal value to that of firingtime before applying the rule, and
- the chronos attributes of all objects on the right hand side of the rule are updated to the value of firingtime after the rule has been executed.

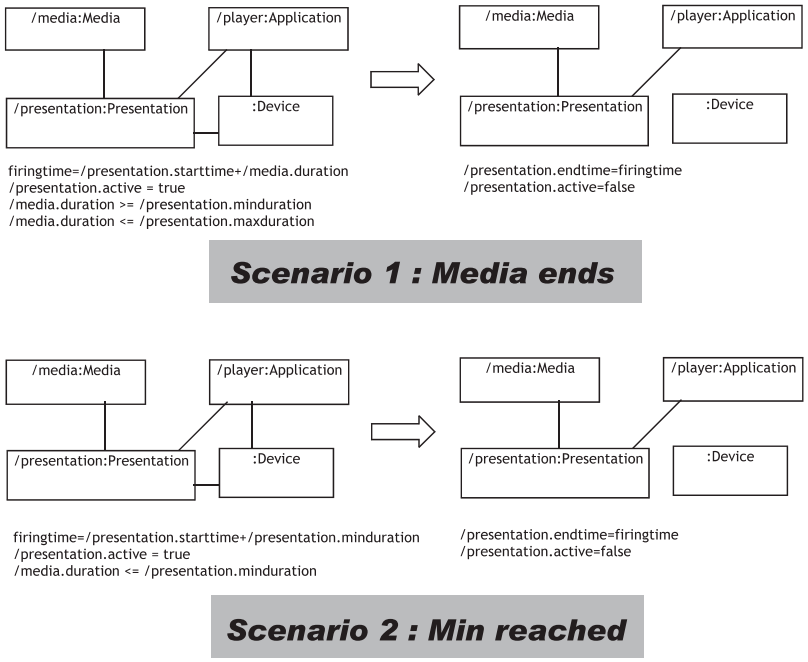


Fig. 4. DMM+t rule for ending a presentation due to the media duration

Specifying the firing time in this fashion is a convenient way to ensure that all resulting rules are correct with respect to the axioms. The value of firingtime may also be used in other conditions to set timestamps etc.

4.2 DMM+t rules for multimedia sequence diagrams

The DMM+t rules defining the semantics of MM sequence diagrams can be distinguished in three groups: rules that describe the end of a presentation (Figs. 4 to 6), rules that describe the reception of a message (Figs. 7 and 8), and rules that describe the sending of messages (Figs. 9 to 11).

One of the features of MM sequence diagrams that requires a specification in a formal time-based semantic domain is the end of a presentation. This end may occur in a number of ways. Either the media duration is inside the specified parameters and the presentation stops "naturally" (i.e., the presentation runs just as long as the media's duration), or the minduration or maxduration constraints force the end of the rendering to occur at a certain point in time. A further possibility is the reception of a stopmessage. Figures 4 to 7 specify these alternatives. The simplest one is the normal end of a media object. The corresponding rule is represented in Fig. 4. The preconditions state that this rule is only applicable if media.duration fulfills the specified constraints, i.e., if it falls inside the specified interval. Note that the association (association role, to be precise) to the :Device role is only present on the left hand side of the rule, i.e., it is deleted when applying the rule since the device is deallocated. This is the same for all rules ending a presentation. The two rules in Figs. 5 and 6 specify the end of the presentation due to the minduration or maxduration con-

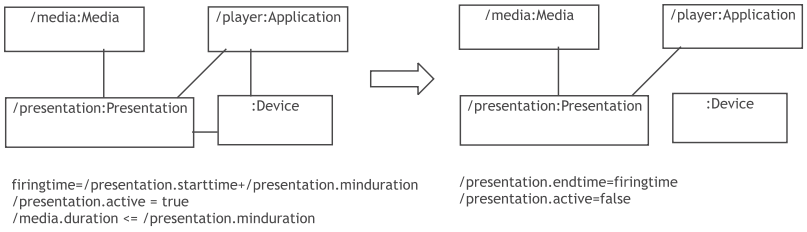


Fig. 5. DMM+t rule for extending a presentation up to minduration

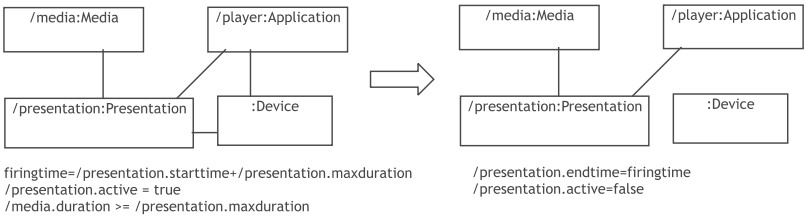


Fig. 6. DMM+t rule for cutting a presentation at maxduration

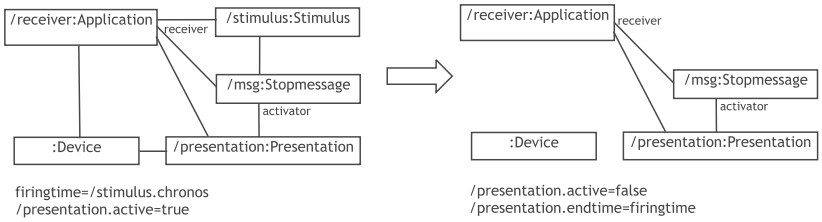


Fig. 7. DMM+t for receiving a stopmessage

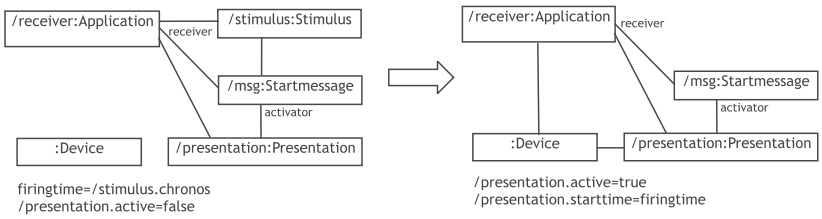


Fig. 8. DMM+t rule for receiving a startmessage

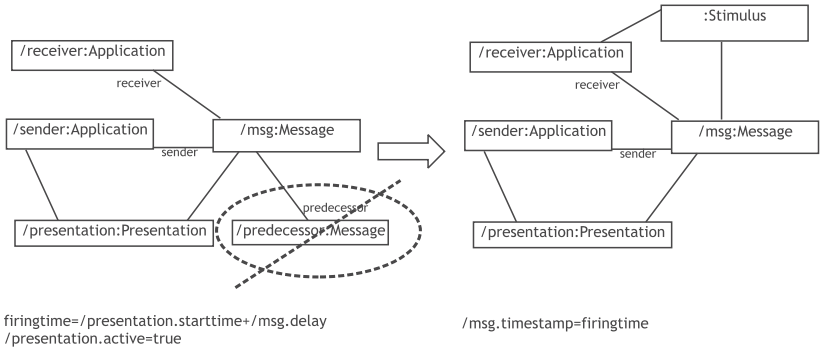


Fig. 9. DMM+t rule describing the sending of the first message in an order

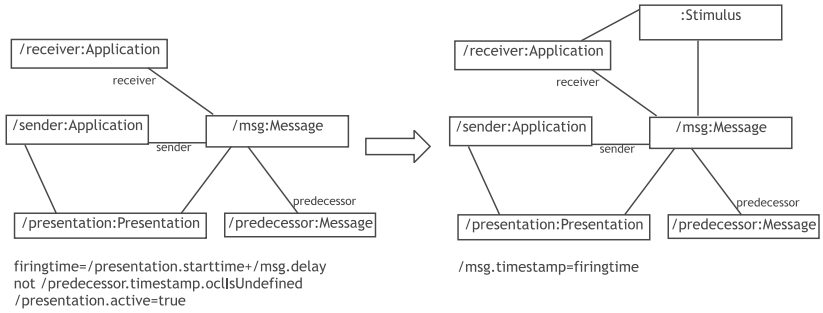


Fig. 10. DMM+t rule describing the sending of subsequent messages in an order

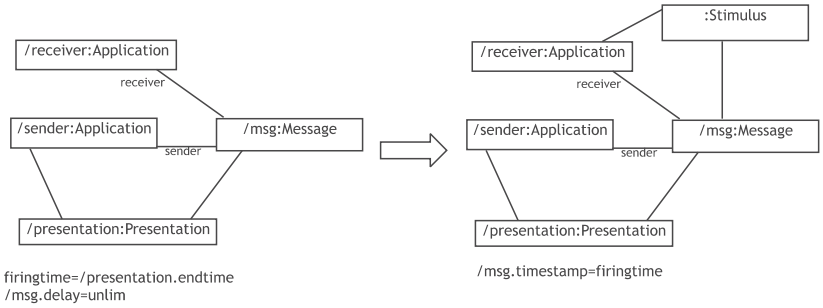


Fig. 11. DMM+t rule defining the sending of end-synchronized messages

straints. Only their preconditions differ, the effect (the end of the presentation) is the same for all three rules.

In contrast, the reception of a stopmessage requires the presence of a stimulus for the message as shown in Fig. 7. A stimulus is the UML instance of a message specification.[1] Whenever a message is sent, a stimulus is created and attached to the receiving object. In this way, sending and reception are decoupled. Thus one might e.g. formulate rules including protocol-based reception mechanisms (as specified by statecharts) as an alternative to direct reception. Since we focus on MM sequence diagrams, we provide direct reception rules for startmessages and stopmessages. The reception of a startmessage is specified in Fig. 8. The stimulus that indicates the pending message is consumed in the course of the rule application, and a device for rendering the media element is allocated. Note that as the firing time of both message reception rules is given as stimulus.chronos (the reception time of the stimulus), we assume a communication without observable delays. It would also be possible to model fixed delays, delays within a bounded interval or unbounded message delays by modifying this condition.

The third set of rules describes the sending of specified messages (Figs. 9 to 11). We have to distinguish between messages that have a numerically specified delay (either 0, indicating a synchronization with the start of the owning presentation, or a fixed time after that) and messages that have a delay value of unlim, indicating a synchronization with the end of the presentation. According to the UML specification, we also have to account for messages being in predecessor/successor relationship. The semantics specification results in three different rules describing the sending of specified messages.

[1] The precise definition of a Stimulus is not repeated here, it can be found in the UML specification [18, pp. 2–103ff].

Figure 9 displays the rule for the first message in a sequence that has a limited delay. Since the attribute msg.delay is present in the definition of the firing time, this could never be true for the value unlim (firing time can never be unlim). To interdict the application of the rule to any message which has a predecessor (and is thus not the first one), a negative application condition is used. A negative application condition (NAC) as defined in [11] contains a subgraph that must not be present in the context of a rule occurrence. It is indicated by enclosing the elements of the NAC in a dashed and canceled area. Therefore, this rule could never match the role /msg to any message in a given hostgraph that has a predecessor. The effects of the rule are the creation of a Stimulus at the receiving object and the specification of a timestamp for the message, thus indicating successful sending of the message.

The rule in Fig. 10 can be applied to any subsequent message in the order specified by the predecessor relationship. It requires the previous message to be executed (timestamp is set) and also requires a finite value for its delay. Note that if all messages would be required to carry a delay value, the way of processing messages could be simplified by determining the order based on their delay values. Combining the two ordering mechanisms can cause contradictions in the specifications (i.e., the specified order of messages and their delays do not correspond), but gives flexibility to combine timed specifications as used in our example with standard features of UML sequence diagrams. One could e.g. combine MM sequence diagrams with messages that are sent in reaction to some external event rather than after a fixed delay. Those messages would then have an unspecified delay, but could be placed in an order with the timed messages.

The remaining case that has to be specified leads to the resolution of an (intentional) ambiguity in the description of MM sequence diagrams in Sect. 3. The description in natural language does not yield any information on the situation that a presentation ends before all its messages are executed. The trivial options are either to send all remaining messages immediately (disregarding their delay values) or to discard them. When discarding the messages, we can furthermore distinguish whether end-synchronized messages should still be sent (disregarding the fact that some of their predecessors have been discarded) or whether they should also be discarded. Note that the two previous rules do not yield any information on this as they only apply in the case of active presentations. The rule in Fig. 11 does clarify these ambiguities. It states that at the end of a presentation the end-synchronized messages will still be sent. No condition requires the execution of previous messages, thus end-synchronization overrides the predecessor/successor order. There is furthermore no rule to process remaining messages that are not end-synchronized, thus they are discarded. This example shows how a formal specification strengthens and clarifies concepts presented in natural language although different semantic decisions could be taken.

4.3 Interpreting the semantics

One of the motivations for deploying formal semantics is the need to gain additional information on the model under consideration. Since specifications can be quite complex, it is not always obvious whether the specification has a meaningful interpretation or if it complies with the intention of the modeler. The definition of DMM+t rules facilitates an interpretation of a given model. The interpretation can be used for a visualization or a test of a given situation. This is especially important if certain elements of the model are underspecified, e.g. the duration of media elements is not given. Here, a modeler has to ensure that at least a few chosen test cases produce the intended behavior.

For such a test, an initial configuration (test case) has to be provided. In addition to the specification given in Fig. 3, we will assume the model to be in a state where all chronos values are initially 0, the timestamps are undefined, and all application objects are inactive. A trigger for the whole scenario is created by assuming that presentation p1 is to be started, i.e., p1.starttime is 0, p1 is active. We additionally assume a length of 50 seconds for Intro. A fragment of this configuration is depicted in Fig. 12.

The rule that can be applied on this initial configuration is the one shown in Fig. 9. The roles match as follows: /receiver on Tape Player, /sender on Projector 1, /presentation on p1, and /msg on m1. According to the OCL constraints of this rule the firing time will be at choronos=5 and a Stimulus will be created. Applying the rule of Fig. 8 will consume the stimulus and activate p2.

Abstracting from the actual details of these graph matchings, Fig. 13 shows an overview of the possible configurations and the transitions between them. In this figure, configurations are characterized by the chronos values of the four presentation objects and their application objects. Presentations that are currently active are shaded in grey. The names of the configurations are given as roman numerals. Labeled transitions between the configurations indicate the rules applied as well as the chronos value of the rule applications (the firing time). Rule applications with identical firing times have been combined into one transition.

The alternative paths between the configurations II and IV illustrate the effect of the global monotonicity theorem introduced in Sect. 2. Although some elements of configuration IIIa have already reached a chronos value of 200, performing operations on independent elements with lower chronos values (in the past) still remains possible. Whether an actual interpreter would compute the diagram by the path via configuration IIIa or configuration IIIb is non-deterministic. The theorem guarantees for each (successful) path of rule applications that an alternative path exists that is ordered with respect to the chronos values of the rule application (in the example the path via IIIb). Therefore, the result of the interpre-

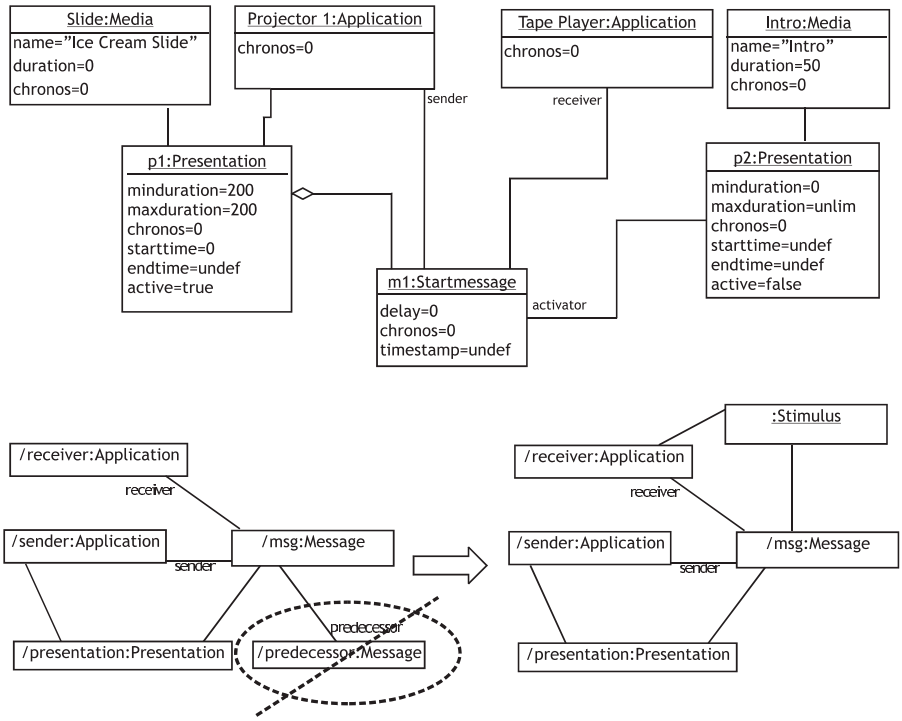


Fig. 12. Excerpt from the initial configuration

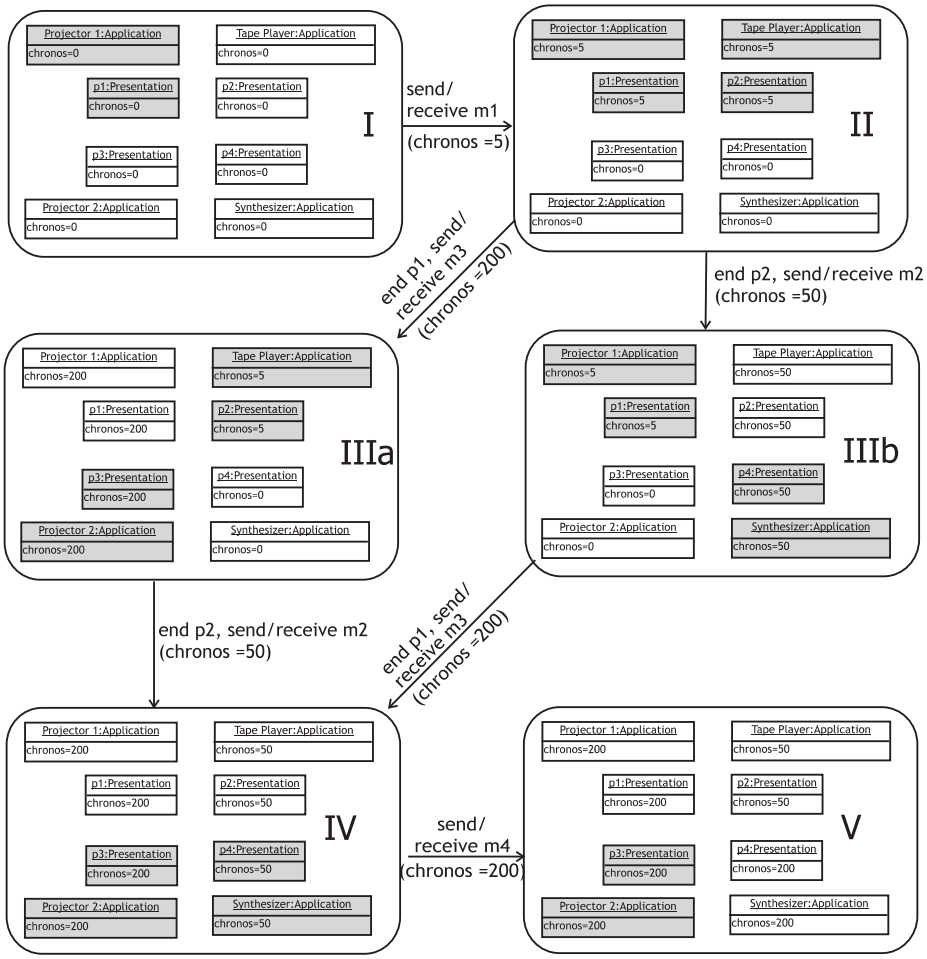


Fig. 13. Trace of the execution of the test scenario

tation is independent of the ordering of rule applications in an actual interpretation. The terminal configuration, i.e., the terminal instance graph, is not shown here; it is reached when the movie ends.

In general, a terminal state is reached when all presentation objects have been presented, that is, their endtime attributes are set. This intuitively corresponds to the completion of the scenario at the bottom of the sequence diagram.

More information on the application's behavior can be gained from possibly defective test cases, because the system's reaction to unexpected situations cannot easily be predicted. If we assume the length of the Intro to be 230 seconds, it is not intuitively clear how this situation is handled. If we apply our semantic rules to this scenario, we find that the scenario is invalid under the rules we specified. At the end of the slide presentation, a Stopmessage is sent to the tape player, but the resulting Stimulus cannot be consumed since the object is not active. Thus the Stimulus stays attached. At firingtime=235 the Intro ends and sends a Startmessage to the tape player. This can now be processed, the player starts. A conflict with the still lingering stimulus of the stopmessage does not occur as the conditions for processing the stopmessage are neither met before the startmessage is being processed (presentation not active) nor afterwards (chronos has been advanced). Thus the music never stops and no terminal state can be reached.

5 Strengthening the semantics

In the previous section we demonstrated how DMM+t rules can be used to specify semantics for UML diagrams and how these specifications can be put to use. Yet some details of the presented approach need to undergo a further investigation to guarantee that DMM+t rules are indeed able to express all concepts of MM sequence diagrams (and other dynamic UML diagrams with temporal aspects).

The first and most general observation is that building on graph transformations as the basis of the semantic domain gives a high degree of freedom. All graph transformation rules can be executed on any part of the graph provided their application conditions are satisfied. While this allows for a non-deterministic and concurrent execution of a specification, it has to be ensured that no unintended effects arise from this flexibility. The axioms given in Sect. 2 already restrict the handling of the chronos attributes to firing sequences that conform to the general concept of passing time. These axioms have been enforced by introducing the variable firingtime in the DMM+t rules and interpreting it accordingly. Thus the distributed presentation of media elements can be modeled using this semantic domain.

MM sequence diagrams provide some other notions that have to be properly represented. Outgoing messages from a presentation have an order (specified by their delays and/or the predecessor relation). To ensure that this order is preserved in the semantic domain, the rules for message processing are formulated to create a dependency between them. Each of the message-processing rules sets the timestamp of the message it executes and thus creates the context for the processing of the next message. Another intuitive order is embedded in the messaging mechanism. A message cannot be received before it is sent. Again, this is enforced by creating a dependency between the rules, based on the existence of the stimulus object.

But there still exists a degree of freedom in the semantic domain that yields non-intuitive results. Consider the MM sequence diagram given in Fig. 14 and assume that the media element assigned to presentation p1 has a duration of 80 (for the moment we will disregard p2 and p3 completely). If the presentation p1 starts at chronos=0, the rules for sending the message (at firingtime=100) and for reaching the end of the media object can both apply. If the message is sent, an awkward situation occurs. Not only does the sending of the message contradict the concepts specified in Sect. 3 and refined in Sect. 4. But also, by sending the message, the chronos value of the presentation object is advanced to a value of 100, thus the rule of Fig. 4 can never apply and end the presentation. Obviously, this is an unwanted behavior.

In terms of the formalization in Sect. 2 we have a conflict between two rules, which is unintended, because we want only the rule for ending the presentation to fire. There are two ways to rectify this situation. The first would be to introduce dependencies between all rules conflicting in this way. Those pairs of conflicting rules can be found e.g. employing the tool AGG [22]. This would result in a lot of artificially created elements embodying the dependencies. These would clog the specification, making it hard to understand and change, especially as each change might create new conflicts. Thus a more general solution has to be found.

The second and in this case much more elegant way to resolve this conflict is to introduce a general notion of priority. The human intuition for a resolution of this conflict would be that whichever rule would be "earlier", i.e., at a lower firing time, should be applied. We can compare the firing times since the rules are in

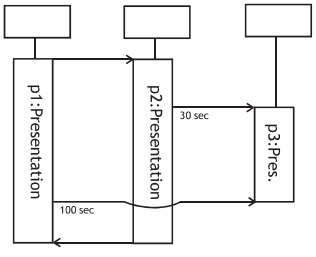


Fig. 14. Example of a complex MM sequence diagram

conflict and are thus influencing the local time of at least one common object (and all objects synchronized with it by the rule). Thus we state that an occurrence of a rule may only happen if there is no conflicting rule that might occur at a lower firing time. We call the semantics conforming to this restriction the *locally strong semantics*. By requiring this kind of priority it is guaranteed that the local order of rule occurrences does indeed conform to the intuitively expected order of events.

This order is still enforced only locally. It does not prevent "unintuitive" firing sequences due to the lack of synchronization of local clocks. Refer again to Fig. 14. If we assume the media of p1 to have a duration of 80 and p2 to have a duration of 70, we might expect that the end-synchronized stopmessage sent by p2 will stop p1 before it reaches its normal end. But once the presentations p1 and p2 are started, both the rule for ending p1 due to the end of the media (it has a lower firing time than the conflicting rule for sending the message at firingtime=100) and the rule for sending the message from p2 to p3 at firingtime=30 can occur (it is not in conflict with the rule for ending p1 as there are no common elements). Thus, non-deterministically, we might end p1 and thereby advance its chronos to 80. Then at firingtime=70, p2 would end, but the rule for emitting the stopmessage to p1 would not be applicable since the chronos value of p1 would already be advanced to 80. Note that this is not a faulty behavior if we regard the different media objects as unsynchronized entities with their own local clocks. Under this assumption, the sequence described above could yield valuable information for the case of a very "slow" componen rendering p2.

But not every timed UML diagram assumes this kind of local clocks. In most cases it is much simpler (and justifiable from the appliation domain) to assume perfectly synchronized components, i.e., a global clock. In this case, yet another restriction can be placed upon the semantics: we can require an interpreter to always choose from all possible rule occurrences one with minimal firing time. This leads to the automatic creation of a globally time-ordered firing sequence in which no object can "overtake" another. In [8] this concept is called the *strong semantics*. This kind of semantics places a heavy restriction on the non-deterministic nature of the underlying formalism, as it only allows for a non-deterministic choice between possible rule occurrences that happen at the same firing time. This interpretation of the rules is closest to the intuitive human interpretation.

We imagine that a tool could provide both possible intrepretation mechanisms. In that way, a modeler could first interpret his specification under the assumption of perfect time and then move to an interpretation taking distributed clocks into account (if this is required by the application domain). We believe that both interpretations yield interesting new and valuable information on the model.

6 Conclusion

In this paper, we have extended the approach of Dynamic Meta Modeling [2, 9] to specify the semantics of time-dependent dynamic behavior of UML models. As a case study, we have applied this approach to multimedia sequence diagrams, a variant of UML sequence diagrams for modeling the control of multimedia presentations.

We have taken a high-level point of view in two different respects. First, the proposed semantics is at the requirements level, that is, we have assumed zero duration for the operations like the start or termination of a presentation or the transmission of a message. Furthermore, we did not consider failures and delays due to imperfect infrastructure (e.g. lack of resource availability), that may appear for instance in distributed Web-based multimedia applications. To capture those aspects, an extension of both the language of multimedia sequence diagrams and its semantic rules is required.

Second, the rules themselves are high-level because they assume an execution mechanism based on global pattern (that is, graph) matching which provides non-determinism or backtracking to search for a successfully terminating sequence. Although there are tools supporting these paradigms [21], this is quite different from standard object-oriented concepts. It is a topic of future research how to map abstract semantic rules to object-oriented implementations.

References

1. David A, Möller MO, Yi W (2002) Formal verification of UML statecharts with real-time extensions. In: Proc. 5th International Conference on Fundamental Approaches to Software Engineering (FASE 2002), Lecture Notes in Computer Science, vol 2306. http://www.springer.de/comp/lncs. Springer, pp 218–232
2. Engels G, Hausmann JH, Heckel R, Sauer S (2000) Dynamic Meta Modeling: A graphical approach to the operational semantics of behavioral diagrams in UML. In: Evans A, Kent S, Selic B (eds) Proc. UML 2000, York, UK, Lecture Notes in Computer Science, vol 1939. http://www.springer.de/comp/lncs. Springer-Verlag, pp 323–337
3. Ehrig H, Pfender M, Schneider HJ (1973) Graph grammars: An algebraic approach. In: 14th Annual IEEE Symposium on Switching and Automata Theory. IEEE, pp 167–180
4. Engels G, Sauer S (2002) Object-oriented modeling of multimedia applications. In: Chang SK (ed) Handbook of Software Engineering and Knowledge Engineering, vol 2. World Scientific, Singapore, pp 21–52
5. Firley T, Huhn M, Diethers K, Gehrke T, Goltz U (1999) Timed sequence diagrams and tool-based analysis – A case study. In: France R, Rumpe B (eds) Proc. UML '99, Lecture Notes in Computer Science, vol 1723. Springer-Verlag, pp 645–660
6. Gyapay S, Heckel R, Varro D (2002) Graph transformation with time: Causality and logical clocks. In: Corradini A, Ehrig H, Kreowski H-J, Rozenberg G (eds) Proc. 1st International Conference on Graph Transformation (ICGT 02), Barcelona, Spain, Lecture Notes in Computer Science, vol 2505. Springer-Verlag, pp 120–134
7. Gyapay S, Heckel R, Varro D (2003) Graph transformation with time In: Fundamenta Informaticae 58(1):1–22

8. Ghezzi C, Mandrioli D, Morasca, Pezzè, S (1991) A unified high-level Petri net formalism for time-critical systems. IEEE Transactions on Software Engineering 17(2):160–172
9. Hausmann JH, Heckel R, Sauer S (2001) Towards dynamic meta modeling of UML extensions: An extensible semantics for UML sequence diagrams. In: Proc. IEEE Symposia on Human-Centric Computing Languages and Environments (HCC'01), pp 80–87
10. Hausmann JH, Heckel R, Sauer S (2002) Dynamic Meta Modeling with time: Specifying the semantics of multimedia sequence diagrams. In: Bottoni P, Minas M (eds) Proc. International Workshop on Graph Transformation and Visual Modeling Techniques (GTVMT 2002), Barcelona, Spain, Electronic Notes in Theoretical Computer Science 72(3). `http://www.elsevier.nl/locate/entcs`. Elsevier Science
11. Heckel R, Wagner A (1995) Ensuring consistency of conditional graph grammars – A constructive approach. In: Proc. of SEGRAGRA'95 "Graph Rewriting and Computation", Electronic Notes in Theoretical Computer Science, vol 2. `http://www.elsevier.nl/locate/entcs`. Elsevier Science
12. Kreowski H-J (1977) Manipulation von Graphmanipulationen. PhD thesis, Technical University of Berlin, Department of Computer Science
13. Küster JM, Stroop J (2001) Consistent design of embedded real-time systems with UML-RT. In: Proc. IEEE Symposium on Object-Oriented Real-Time Distributed Computing (ISORC 2001), pp 31–40
14. Li X, Lilius J (1999) Timing analysis of UML sequence diagrams. In: France R, Rumpe B (eds) Proc. UML '99, Lecture Notes in Computer Science 1723. Springer, pp 661–674
15. Guldstrand Larsen K, Pettersson P, Yi W (1997) UPPAAL in a nutshell. International Journal on Software Tools for Technology Transfer 1(1–2):134–152
16. Mühlhäuser M, Gecsei J (1996) Services, frameworks, paradigms for distributed multimedia applications. IEEE Multimedia 3(3):48–61
17. Object Management Group (2002) UML Profile for Schedulability, Performance, and Time, OMG adopted specification
18. Object Management Group (2003) OMG Unified Modeling Language Specification, version 1.5
19. Petriu DC, Shen H (2002) Applying the UML performance profile: Graph grammar-based derivation of LQN models from UML specifications. In: Computer Performance Evaluation, Modelling Techniques and Tools (Proceedings of TOOLS 2002), Lecture Notes in Computer Science, vol 2324. Springer-Verlag, pp 159–177
20. Sauer S, Engels G (2001) UML-based behavior specification of interactive multimedia applications. In: Proc. IEEE Symposia on Human-Centric Computing Languages and Environments (HCC'01), pp 248–255
21. Schürr A, Winter AJ, Zündorf A (1997) PROGRES: Language and environment. In: Ehrig H, Engels G, Kreowski H-J, Rozenberg G (eds) Handbook on Graph Grammars and Computing by Graph Transformation: Applications, Languages, and Tools. World Scientific, Singapore, pp 487–550
22. Taentzer G (1999) AGG: A tool environment for algebraic graph transformation. In: Proc. International Workshop on Applications of Graph Transformations with Industrial Relevance (AGTIVE 1999), pp 481–488

Jan Hendrik Hausmann is a PhD student at the University of Paderborn, Germany. His research topics include object-oriented modelling with the UML, graph transformations and their application in software engineering, meta modelling, and eLearning systems.

Reiko Heckel is assistant professor at the University of Paderborn, Germany, since 1998. His research interest is the use of graph transformation in software engineering, including the development of relevant theory like structuring and modularity concepts, concurrency theory, graph-based temporal logic, etc. and the application of this theory to the modelling of object-oriented and agent-based systems, and to the semantics of visual modelling languages.

Stefan Sauer works as a research and teaching associate at the University of Paderborn in the database and information systems group. The focus of his research is on object-oriented modeling with the UML, semantics of visual languages, meta modeling, extensions of the UML for interactive multimedia applications, and multimedia software engineering.

OBJECT-ORIENTED MODELING OF MULTIMEDIA APPLICATIONS *

GREGOR ENGELS and STEFAN SAUER
University of Paderborn
Mathematics and Computer Science Department
D–33095 Paderborn, Germany
E-Mail: {engels|sauer}@upb.de

The field of multimedia software engineering is still in an inmature state. Significant research and development has been dedicated towards multimedia services and systems technology such as networking or database systems. Multimedia document formats have been standardized. But when it comes to multimedia application development, the development process is truncated to an implement-and-test method. Either specialized multimedia authoring systems or multimedia frameworks or toolkits complementing programming languages or system software are directly used for implementation. No preceding modeling phases for requirements specification, analysis, or design of the system to build are enforced. The development of sophisticated multimedia process models and established, usable graphical notations tailored to the specification of multimedia systems is still underway.

In order to fill this gap, it is the purpose of this chapter to show current achievements in object-oriented modeling of multimedia applications. Based on an analysis of the state of the art in multimedia application development, we shortly present approaches to object-oriented hypermedia modeling and extensions of the Unified Modeling Language (UML) for hypermedia and interactive systems. The main part of the chapter is dedicated towards a recent approach to the Object-oriented Modeling of MultiMedia Applications (OMMMA).

Keywords: Multimedia software engineering, object-oriented modeling, integrated modeling, Unified Modeling Language.

*chapter in S. K. Chang (editor), *Handbook of Software Engineering and Knowledge Engineering*, vol. 2, pp. 21-53, World Scientific, Singapore, 2002. ISBN 981-02-4974-8. http://www.wspc.com/books/compsci/4603.html

1 Introduction

Multimedia applications can be defined as interactive software systems combining and presenting a set of independent media objects of diverse types that have spatio-temporal relationships and synchronization dependencies, may be organized in structured composition hierarchies, and can be coupled with an event-based action model for interaction and navigation. Media objects contribute essentially to presentation and interaction via the user interface. Two major categories of media types can be distinguished:

- *static media types*: time-independent media that do not show any temporal expansion and whose presentation is invariant over time, e.g. text, image, and graphics; and
- *temporal media types*: time-dependent media that possess time-dynamic behavior and whose presentation varies over time, e.g. animation, video, and audio.

Interactive multimedia applications are becoming a widely-used kind of software systems. By integrating multimedia elements, application programs can be made more comprehensible. For instance, it is better to provide auditive examples of musical pieces within an encyclopedia of classical composers or to show a video on the deployment and functioning of a computer tomograph than to simply provide this information textually or with single pictures. Therefore, multimedia is especially useful if the information to present is itself inherently multimedia. Additionally, multimedia can also make user interaction more intuitive by reproducing natural forms of interaction, e.g. with simulated laboratory instruments like rotary switches or analogue indicators, or by speech and gestures [9]. We expect that typical multimedia application domains will not be restricted to purely presentational objectives, like web pages, educational courseware, interactive entertainment, or multimedia catalogues, but conventional business, technical, and information systems will be extended with multimedia features.

State of the art in multimedia software development is that multimedia applications are directly built either by deploying multimedia authoring systems or by coding them using multimedia frameworks or toolkits adding multimedia features to (object-oriented) programming languages or system software. In both cases, no preceding modeling phase as part of a sophisticated software engineering process is carried out prior to implementation. But the importance of such an activity in the course of multimedia software development has been advocated (see e.g. [3] or [8]).

From a software engineering perspective, the problem with the current state of multimedia application development is not only the absence of *sophisticated, yet practical multimedia software development process models*, but also the lack of *usable (visual) notations to enable an integrated specification* of the system on different *levels of abstraction* and from different *perspectives*. Such languages must be understandable by the different people and stakeholders involved in the multimedia development process.

While support by authoring systems, multimedia technologies, services, and standard formats eases the implementation and exchange of multimedia software, they must be complemented by more abstract, yet precise modeling or specification methods for these systems. What is especially missing is the integrated specification of the multiple aspects of multimedia systems like real-time behavior, user interaction, and application logic. This becomes a key issue when multimedia applications continue to evolve towards higher complexity, and multimedia features become prominent in almost any application domain in order to make information more conceivable as well as user interaction more sophisticated and lively. Effective management and maintenance of such applications requires testing and quality assurance, adaption and reuse of its parts, as well as readable and structured documentation. These requirements are supported by a modeling activity within the development process. Precise specification techniques add to this the capability of validation other than testing by model analysis. Instead of authoring systems supporting the instance-level assembling of interactive multimedia presentations, multimedia CASE tools are needed that support an integrated development process. The implement-and-test paradigm used during multimedia authoring resembles the state of software development before leading to the software crisis of the 1980s.

From this observation, we claim that the development process of multimedia applications should include a *modeling* activity, predominantly on analysis and design levels, like they are essential in conventional software engineering methods. Especially the design phase is required to achieve a clearly structured and error-free implementation. Concepts, languages, methods, and tools must be developed that take the specific requirements of multimedia systems into account and support a methodical multimedia software process. As regards software specification prior to implementation, we consider an *integrated visual modeling language* with capabilities to describe all relevant aspects of the system in an interrelated fashion as the most promising approach. Since the object paradigm has proven many intrinsic properties to support software product and development process quality – like encapsulation, modularity, extensibility, adaptability, portability, integrated specification of structure and behavior, or seamless integration by using a uniform paradigm throughout the process – we regard an *object-oriented modeling language for multimedia* that enables an integrated specification as being well-suited for this task. Object-oriented modeling eases the transition to the implementation of a multimedia application since implementation technologies as well as multimedia databases are mostly based on the object paradigm, too.

A number of models have been proposed for multimedia applications. One striking disadvantage of most models is that they only support partial models for individual aspects of the system, but not a holistic model for the integration of these modeling dimensions. Primarily, they focus on modeling of temporal relations and synchronization of multimedia presentations (e.g. [41, 61]; consult [6] for a general classification). Some more elaborated models also account for interactivity (e.g. [31]). Others concentrate on logical structure and navigational concepts for hypermedia (e.g. [33, 51]). Another problem with many of

these models is that they are not intended to be directly used by a multimedia software engineer during a development project. Instead, they form the conceptual basis of (proprietary) authoring systems. Thus, they are not suited for directly supporting modeling activities in a multimedia software development process.

In traditional software engineering, the Unified Modeling Language (UML [47, 11])) has become the *de facto* standard notation for software development from the early stages of requirements specification up to detailed design. It has been adopted as the standard modeling language by the Object Management Group (OMG) and has been submitted for standardization to the International Standardization Organization (ISO).

UML is a family of visual, diagrammatic modeling languages that facilitate the specification of both software systems and process models for their development. The diverse language elements of its constituent sub-languages enable modeling of all relevant aspects of a software system, and methods and pragmatics can be defined how these aspects can be consistently integrated.

Unfortunately, UML does not support all aspects of multimedia applications in an adequate and intuitive manner (see Sect. 4.1). Especially, language features for modeling time, synchronization, and user interface aspects are not explicitly provided. Other concepts of UML are not mature enough or less vivid and thus aggravate multimedia modeling unnecessarily.

But UML comes with built-in extension mechanisms and a so-called profiling mechanism [47] in order to adapt and extend the general-purpose modeling language for specific development processes and application domains.

These fundamental concepts of the UML form the basis for the object-oriented modeling approach for multimedia applications that is presented as the main content of this chapter. Based on the characteristics of multimedia applications, we have developed extensions of the UML to specify all relevant aspects of multimedia applications in a single and coherent model.

The aim of this contribution is therefore to give an overview of existing approaches to object-oriented multimedia software modeling and, in particular, to introduce the UML-based, visual multimedia modeling language OMMMA-L (**O**bject-oriented **M**odeling of **M**ulti**M**edia **A**pplications - the **L**anguage). In this intention, we first identify the characteristic dimensions of multimedia software engineering and their relevance for an object-oriented modeling approach for multimedia applications in Sect. 2. We then give an overview of the state of the art in (object-oriented) multimedia software development. The UML-based modeling approach OMMMA is presented in Sect. 4 where we point out the essential elements of OMMMA-L for the specification of the different aspects of multimedia applications. Section 5 sketches some future directions of research and practice in the area of multimedia software engineering influenced by object-oriented modeling, and Sect. 6 concludes this chapter by summarizing the current achievements.

2 Dimensions of Multimedia Software Engineering

Software engineering can be approached from both the process and the system perspective. Within these perspectives, a wide range of dimensions exist that need to be considered. Each dimension captures a particular aspect of software engineering. But these dimensions do not exist in isolation, they have interdependecies with each other. Dimensions can be represented by partial models of software engineering that must be integrated. A formal, graph-based model for integrated software engineering has been presented in the GRIDS approach by Zamperoni (see [63]). The instantiation of this generic (meta-) model is exemplified by a three-dimensional model of software engineering 3D-M. This model identifies software processes, architectures, and views of the system as the three fundamental dimensions of software engineering and addresses the integration of the corresponding partial models.

The *process* dimension distinguishes different development phases (or activities) such as requirements specification, analysis, design, implementation, and testing. Modeling and implementation phases generally correlate to different levels of abstraction. Coordinates of the *architecture* dimension are different system components, e.g. interfaces with the user or external systems, control, processes, or a repository, whereas the *views* on the system can be distinguished in e.g. structure, function, dynamic behavior etc. Each view captures specific aspects of the system. (Note that architecture can also be regarded as a high-level internal view of the system to be built.) Software development can thus be understood as a (multi-) path through the multi-dimensional space of this integrated software engineering model, where the trajectory is defined by a process model, and suited software engineering technologies on the levels of concepts, languages, methods, and tools are applied.

If we look at the characteristics of multimedia applications and their development processes, we easily observe the necessity to tailor and extend these general dimensions. Multimedia specific aspects introduce new process activities, new requirements regarding the architectural structure, and new views on the system.

The *process* dimension must be extended to account for specific life-cycles of multimedia applications (see e.g. [52]) and to include activities of content and media production. Traditional development activities must be adapted, e.g. through new kinds of testing.

The *architecture* dimension needs to be refined for multimedia systems. *Multimedia systems* require complex architectures combining a multitude of hardware and software components. Thus, architectural design resembles software-hardware co-design of embedded systems. Architectural extensions are needed to account for different notions of media that co-exist in a multimedia system such as perception, presentation, representation, storage, and transport media (see [36]), or for media-related processors like filter, converter, and renderer components. Software components of multimedia systems can be categorized

into system, service, and application components. *Multimedia applications* are interactive and have a strong emphasis on a multimedia user interface. Thus, software architectures from the field of interactive system modeling, e.g. Model-View-Controller (MVC [39]) or Presentation-Abstraction-Control (PAC [16]) can be considered as a basis for the architecture of multimedia applications. (A similar approach was followed in the development of the MET++ framework [1].) Since multimedia applications in general allow for multimodal interaction, architectures for interactive systems have to be specialized by adapting the control component to incorporate support for modes of interaction.

Views need to be added or reconsidered: the structure view has to account not only for domain objects, but also for media objects of static and temporal media types. Multimedia-specific application frameworks and patterns may be deployed. A presentation view, i.e., the design of the user interface with respect to spatial (and temporal) relations of its constituents becomes fundamental – more than in traditional interactive systems. Although space and time are mainly perceivable at the user interface, other system entities, such as media or domain entities, are subject to spatial and temporal relations (or constraints), too. In fact, the most important new aspect of multimedia systems is that of space and time, i.e., whether parts of the system have a temporal expansion relative to some time axis or a spatial within some two or three-dimensional coordinate space. Thus, a temporal and a spatial view or an integrated spatio-temporal view are mandatory. Temporal behavior must also account for real-time features and synchronization. Behavior can be predefined (algorithmic) or interactive, i.e., non-deterministic at specification time, because the system has to react to (run-time) events that are unpredictable in time. Multimedia-specific events, e.g. regarding the processing of temporal media data or specific media system services, need to be incorporated.

As has been stated above, the dimensions for architecture and views on the system are interrelated with each other. The relations between different views are partly influenced by architectural structures of the system and, vice versa, views on particular application aspects influence the architectural separation of concerns. For example, the interactive behavior view has to account for degree and modes of interaction (encapsulated in user interface components). Due to multimodal inputs resulting in the same application events, the coupling of interactive control behavior and system function needs to be revised. The coupling of these views can, for example, be accomplished by the architectural separation of physical user actions in user interface components – wrapping input devices – from application behavior in process components, mediated by different levels of technical and logical events exchanged and handled by the respective architectural components.

In case of hypermedia systems, navigation views can be understood as a further refinement of both the structure and the function views. Other characteristics that can be represented by dedicated views are security, media presentation adaption, quality of service (QoS), or presentation environment (compare [55]).

Since the primary focus of this contribution is not on the multimedia software process dimension, but on the multi-dimensional modeling of multimedia

applications, the dimensions of the system perspective, i.e., architecture and views, are mainly relevant in the following. What remains as a requirement for a modeling language from the process dimension is the necessity to support models with different levels of abstraction. From the aforementioned views, the following can be identified as being fundamental (see [50] for a discussion):

- media and application *structure*,
- spatio-temporal *presentation*,
- *temporal behavior* (function),
- interactive *control*.

Together with the architecture dimension, these views will be used in the remainder to characterize the different presented approaches.

3 State of the Art in Multimedia Software Development

Researchers and practitioners agree that multimedia systems pose new requirements on software engineering (see e.g. [12] or [45] for an overview of topics).

Research and development projects in the multimedia field have been mainly focused on either multimedia enabling system technologies and services, e.g. networking or multimedia database systems, or the use of scripting-based authoring systems as well as XML-based or object-oriented programming technologies for the implementation of multimedia applications. Multimedia software engineering in the sense of specialized multimedia software development process models and usable multimedia specification languages and methods has only drawn minor attention.

In recent time, three different kinds of approaching multimedia software programming have become dominant.

- Most multimedia applications and documents are nowadays developed using multimedia authoring systems. Multimedia authoring systems are visual, interactive programming tools for the development of multimedia applications that can be used by developers with different degrees of expertise.
- With the advent of more complex multimedia software that needs to be integrated with other sophisticated application features, frameworks and application programming interfaces complementing standard (object-oriented) programming languages or system software were introduced.
- With the growing importance of web-based multimedia applications that are portable between multiple platforms, XML-based approaches to multimedia document authoring like the Synchronized Multimedia Integration Language (SMIL [62]; see [32]) – uptaking the SGML-based HyTime

approach – are being implemented aside from proprietary web-enabled formats like Flash (by Macromedia) or Apple's Quicktime.

Since the focus of this chapter is on object-oriented modeling of multimedia applications, we refer to these implementation technologies only in their relations to object-orientation and/or modeling in the following. In particular, authoring systems are discussed in the perspective of integration with modeling, and object-oriented frameworks are presented that show fundamental concepts based on which object-oriented modeling can be easier understood. Nevertheless, we stress that implementation of object-oriented models of multimedia applications can be done with any implementation technology, regardless of its degree of object-orientation. The architecture and views of object-oriented models can, for example, be easily mapped to the main concepts of most authoring systems. Technology-specific decisions would only influence the modeling on the level of detailed architecture and design.

3.1 Multimedia Authoring Systems and Modeling

Authoring systems are (partly) visual programming environments that support *ad hoc* implementation and rapid prototyping of multimedia applications based on direct-manipulative graphical user interfaces and intuitive media production and processing metaphors. They are in general supplemented by a scripting language containing simple language constructs to program extended functionality by hand. Pre-existing media and user interface objects are coupled with built-in mechanisms for dynamic execution, e.g. event handling. Extensibility of this functionality is rather limited.

Architecturally, authoring systems integrate four main functional components: media object tools, composition and structure tools, interpreters, and generators. Multimedia authoring systems can be classified according to the media metaphors they deploy as their main abstractions for design and content provision:

- *Screen or card-based* authoring systems, e.g. HyperCard [2] or ToolBook [13], place media objects on cards, slides, or pages, and navigational interaction allows users to switch between these cards.
- In *icon or flowchart-based* systems like Authorware [42], media objects have iconic representations that are used as nodes in a navigational flow graph.
- Macromedia Director [43] is an example for a *timeline-based* authoring system where media objects are positioned along a time axis and navigational interactions lead to jumps on this axis.

Regarding behavior, both card- and timeline-based categories rely in general on scripting-based event handlers that are associated with user interface elements, while dialogues in flowchart-based systems can be graphically specified.

Authoring systems are object-based rather than object-oriented, i.e., developers work with objects and component instances instead of classes and component types. Reuse in the form of composition of existing artifacts is possible only on the level of instances and scripts specifying object behavior. Inheritance is, if at all supported, restricted to the instance level.

Other disadvantages of most authoring systems are scripting languages built from primitive programming constructs; weak support for structuring, modularization, and reuse, especially in timeline-based systems; unsufficient support for team development; lack of user-defined types; limited documentation generation; and limited semantic foundation for analysis, test design, and maintenance. A striking disadvantage of most authoring systems is that they do not offer open and standardized interfaces for individual extensions or adaptions. Furthermore, the authored applications generally are not platform-independent bcause they use proprietary formats and languages.

The wide use of authoring systems and the lack of sophisticated and practically approved multimedia software process technology are responsible for a multimedia development practice that is *de facto* truncated to implementation and testing phases. This leads to problems well-known in traditional software engineering like missing conceptualization and documentation. Although the multimedia software engineering process is not the topic of this chapter, we will shortly reference two approaches for an integration of UML-based modeling and tool-supported authoring. We hereby intend to show how the current practice of development can be maintained and extended by modeling activities.

Boles et al. deploy UML for modeling and Director [43] for implementing a virtual genetics lab (see [10]). They use class, sequence, and statechart diagrams of UML to specify the application structure and behavior, i.e., possible user interactions as well as internal course of control, of simulated lab experiments. The Model-View-Controller (MVC [39]) paradigm is used as a pattern for implementation, it is not explicitly represented in the model. To transform the model to an executable application, they propose a translation approach into Lingo, the scripting language of Director, bypassing the visual programming capabilities of the authoring system. Events that trigger transitions in statechart diagrams are implemented as methods of the corresponding classes. Afterwards, view and controller classes for the direct-manipulative user interface are added. In this case, modeling is restricted to the application internals, user interface aspects are instantly coded.

To overcome the limitations regarding software engineering principles, we have proposed a process model for improving the development of multimedia applications from a software engineering perspective in [17]. It combines object-oriented modeling in analysis and design phases with an implementation based on a commercial authoring system, exemplified with Director [43]. The main idea is to transform a framework-based analysis model of the application, that is independent of the technology used for implementation, into a program. Key feature of this transformation is a conceptual programming model of the authoring system that bridges the gap between analysis model and implementation. This authoring system model is used during design to map an object model

instantiating the analysis-level class model to an object model instantiating the authoring system model. The resulting design-level instance model can then be implemented in a straightforward manner. This approach has so far only covered the structural aspects of the multimedia application, it has not yet integrated a behavioral or dynamic view.

Depending on their main metaphors, authoring systems more or less explicitly support the different views on an application identified in Sect. 2. While the presentation view is generally well supported, complex application structures and behavior require sophisticated programming. The underlying architecture is transparent to the developer. Since authoring systems are visual programming environments, different levels of abstraction are not supported. From the above discussion we conclude that both multimedia authoring and object-oriented modeling can benefit from each other when they are used as complementary techniques within a multimedia software development process. Which role object-orientation plays in the current practice of multimedia software development will be presented in the next sections.

3.2 Object-orientation in the Development of Multimedia Applications

The important role of object-orientation for multimedia has been stated by many research contributions proposing object-oriented models as a conceptual basis for multimedia. Also, standardization efforts in the multimedia domain have discovered the advantages of object models. The family of MHEG standards [36] for the specifiaction of interoperable interactive multimedia documents (see [19] for an overview of MHEG-5 and its complementary parts) is based on an object-oriented model with abstractions for applications and scenes as well as links, streams, and other basic elements (so-called ingredients). Its focus is on coding and exchange of hypermedia documents. PREMO (Presentation Environment for Multimedia Objects [34]) is directed towards a standardization of the presentation aspects of multimedia applications. It incorporates temporal and event-based synchronization objects (see e.g. [29, 30, 28]). Its object model originates from the OMG object model for distributed objects. PREMO contains an abstract component for modeling, presentation, and interaction that combines media control with aspects of modeling and geometry. For the definition of the MPEG-4 [35] standard, objects have been discovered as a potential source for compression of video data instead of data reduction based on image properties.

But object-orientation is also promoted by the existence of object-oriented class libraries, toolkits, and frameworks that support the programming of multimedia applications. In the following, we summarize some characteristics of the latter since some of these approaches are accompanied by graphic authoring systems that enable visual programming based on an object-oriented conceptual model.

3.3 Object-oriented Multimedia Frameworks

Besides authoring systems, several object-oriented toolkits and frameworks have been proposed to support a programming-based development of multimedia applications. They reify architectural structures and fundamental abstractions of multimedia systems based either on extensions to object-oriented programming languages or on conceptual object-oriented languages that abstract from concrete programming environments. An important characteristic feature of object-oriented multimedia frameworks is their open and extensible architecture. It supports portability and adaptability, e.g. in the advent of new media types that can be integrated. Some developments in this field are, beyond others, the Media Editor Toolkit (MET++ [1]), MultiMedia Extensions (MME [18]), the Berkeley Continuous Media Toolkit (CMT [54]), and Nsync [4]. A recent development is the Java Media API [57] consisting of several components such as the Java 2D and 3D APIs or the Java Media Framework (JMF [58]; see e.g. [25]) for continuous media. Microsoft's DirectX [44] is a multimedia extension on the operating system level.

The main objective of object-oriented multimedia frameworks is to supply a multimedia developer with a software abstraction for multimedia programming. The framework should comply with the fundamental object types and operations that appear in multimedia applications. Conceptually, a framework consists of interrelated abstract classes that have to be implemented by concrete classes for different multimedia platforms. Therefore, one can distinguish at least two layers within such a framework. On the higher level, the abstract framework classes build an application programming interface that can be used by a multimedia developer independently of the target platform when implementing a multimedia application. On the lower level, concrete classes realize a platform-dependent implementation of the abstract concepts on the higer level. To achieve such an implementation, the framework classes make use of a system programming interface. The framework classes are organized in a generalization hierarchy where the abstract superclasses specify interfaces that are realized by their specialized, concrete subclasses.

Requirements for a multimedia framework are openess, robustness, ability to be queried, scalability, support for architectural structuring, availability of general, high-level concepts and interfaces for spatio-temporal media composition and synchronization, hardware control, database integration, and concurrency.

We will shortly refer to some representative frameworks in the following.

Framework by Gibbs and Tsichritzis. A prototypical object-oriented multimedia framework has been presented by Gibbs and Tsichritzis [24]. It specifies an abstract application programming interface (API) that serves as a homogeneous interface to heterogeneous platforms. It combines two fundamental concepts, a media hierarchy that encapsulates media values and a hierarchy of components. Transform and format class hierarchies are used as supporting concepts.

IMD. Another prominent approach that has been widely recognized is the modeling method for interactive multimedia documents (IMD) by Vazirgiannis [59]. It integrates the temporal and the spatial domain of multimedia documents in a common event-based framework. It comprises an object-oriented event model where elementary events of different, hierarchically specialized event types are combined by algebraic and spatio-temporal operators. Events are conceived as representatives of actions that generate these events, e.g. start or end of an action, parameterized by their subject (triggering object) and object (reactive object) and a spatio-temporal signature. Composite objects are combined from basic media objects by temporal and spatial operators. Based on these concepts, an authoring systeml is provided that allows developers to specify scenarios as a set of autonomous functions, so-called scenario tuples, to which start and end events, action lists and synchronization events, raised at the begin or end of the tuple, can be assigned.

MET++. MET++ [1] is an application framework in that multimedia presentations are modeled as hierarchical compositions of temporal objects. The modeled compositions are automatically transformed in temporal layout mechanisms and propagated. Real-time behavior and user interaction are integrated in the controller part of an extended MVC model. The MET++ class hierarchy includes compositional time-layout objects providing synchronization behavior for temporal relations and time-dependent media objects. Dynamic behavior of temporal media objects is specified by time-related functions that are themselves specializations of complex temporal objects.

Java Media API. The Java Media API [57] contains classes for the integration of animation, imaging, two and three-dimensional graphics, speech, and telephony. The Java Media Framework (JMF [58]), which is part of the Java Media API, offers an interface for accessing and controlling continuous media objects. It does not come with a general time concept that would enable the integrated synchronization of temporal and static media, only synchronization of the former is supported. Furthermore, no sophisticated mechanism for temporal composition is built in.

The presented frameworks and their inherent structuring can be used as sources for the architectural structuring of multimedia systems and applications, and partly for modeling language design. Additionally, their implementations are promising technologies for the implementation of object-oriented models of multimedia applications, especially for complex applications. But all the different views on multimedia applications must be mapped to basic object-oriented programming principles such as objects, messages, and events. Only in cases where the frameworks are themselves accompanied by graphical development (authoring) tools, like MET++, different views are explicitly supported on a higher (visual) level of abstraction.

We now step from object-oriented implementation technologies to modeling techniques. We first direct our attention to object-oriented modeling approaches for hypermedia and interactive systems that can contribute to a holistic multimedia modeling, before we return to the object-oriented modeling of multimedia applications in Sect. 4.

3.4 Hypermedia Modeling

Hypermedia software development has been addressed by different modeling approaches. Because these models focus mainly on *hyperlinked* media, emphasis is put on the design of navigational structures. Conceptual models of a hypermedia application are accompanied by some form of navigation model and sometimes by an abstract user interface model. Other aspects of multimedia, especially temporal behavior of continuous media and synchronization, are underrepresented.

The Object-Oriented Hypermedia Design Model (OOHDM [51]) starts modeling with a conceptual model of the semantics for the application domain that is complemented by a navigational model in a second step. This part of the model is based on an extended Entity-Relationship model. As a third activity, abstract user interface design completes the model. The browsing semantics of OOHDM is based on the static navigation structure specified in the navigational model. OOHDM is an object-oriented extension of HDM [23] and comprises the same basic modeling activities.

The Relationship Management Methodology (RMM [33]) is a method for design and implementation of hypermedia systems. In contrast to OOHDM, navigational structures are modeled within the domain model. Furthermore, RMM enables the generation of HTML documents or code for authoring systems from the model.

HyDev [48] is another proposal for decomposing hypermedia application models in different partial models. The domain model is accompanied by an instance model whose objects are instances of the domain model. The instance model is regarded important since the behavior of multimedia applications often relies on the characteristics of individual objects, and thus multimedia software development has to deal with both type and instance level views. Relevant features of object presentation and navigation between objects are captured in a so-called representation model that abstracts from media formats and concrete media objects.

HyperProp [55] comprises a conceptual model that represents authoring requirements by spatio-temporal constraints, and a formatting algorithm for runtime presentation adaption in reaction to occurrences of specified events. HyperProp is based on a logical document model for the composition of hypermedia artifacts. Architecturally, it distinguishes three layers for representation objects, data objects, and storage objects, respectively. The authoring system prototype contains a graphical editor for constraint specification.

From the perspective of modeling multimedia applications by extensions of the Unified Modeling Language, the work by Baumeister, Hennicker, Koch, and

Mandel is of particular interest. In [5], they propose extensions of the UML to specify hypermedia based on the concepts of OOHDM. In [27], these extensions are used for a further underpinning of the process associated with their language extensions. There, they give a set of guidelines how to (semi-) automatically derive information for a model view from previous models. They start with a conceptual model of the application domain from which they derive a navigation space model for the hypermedia application based on views on conceptual classes and additional navigation associations. From the navigation space model, they build a navigational structure model by incorporating navigational elements such as index, guided tour, query, and menu. Finally, they use the navigational structure model to construct an abstract presentation model focusing on the structural organization of the presentation rather than on the physical presentation on the user interface. The disadvantage of this approach is that interaction is restricted to the navigation via predefined links within the application.

In summary, the main shortcoming of hypermedia approaches is their limited capability of modeling behavior that is in most cases restricted to hyperlink navigation. Other forms of user-initiated control and temporal behavior are only partly considered. Additionally, application structure is often restricted to trees with hyperlinked nodes. Because multimedia applications are highly user-interactive, we look at user interface modeling next.

3.5 User Interface Modeling based on UML

In the UML field, there have also been some research contributions on how to extend the general purpose modeling language UML towards a better representation of user interface modeling dimensions. These are important for multimedia modeling since we have identified the multimedia user interface presentation and interaction as key views of multimedia systems in Sect. 2. Human-computer interaction has to deal with representations of user roles; of user behavior when performing tasks; of abstract conceptual and concrete physical user interfaces. To capture user roles and user requirements, use case diagrams of UML can be deployed. This approach is described in detail by Constantine and Lockwood [15]. In [60], so-called user interaction diagrams (UID) are introduced to detail use cases for requirements specification. For the modeling of behavior from the user's perspective, Kovacevic [38] proposes a UML extension for task analysis.

From a software analysis and design perspective, the UML profile for interaction design suggested by Nunes and Cunha [46] is of importance. Here, analysis and design models for the specification of user interfaces are presented. The analysis classes from the UML built-in profile for software development processes [47] are further refined to architecturally distinguish between interfaces to external systems and for user interaction. On the design level, a dialogue model for structuring dialogues and a presentation model for capturing navigation between different interaction spaces (contexts) are introduced.

The problem with all these models is that they are either on a high level of abstraction such as use cases or other requirement gathering approaches or they

mostly focus on the architectural dimension or structural rather than behavioral aspects (views). An exception is the UML-based approach described in [53] that addresses dynamics of abstract user interfaces. Extended UML activity diagrams are employed to detail the interaction for realizing a use case. But since activity diagrams are still rather high-level behavior descriptions, they are not well-suited to describe detailed behavior of a concrete user interface.

None of the presented approaches of user interface modeling based on UML accounts for specific characteristics of multimedia applications and their implications on user interface modeling or the integration of multimedia applications with the proposed user interface models. Multimedia-specific architectures or system views beyond interaction are hardly supported. In the following section, we show how the OMMMA approach attempts to integratedly specifiy the different aspects of multimedia applications that have been identified in Sect. 2.

4 OMMMA — Object-oriented Modeling of Multimedia Applications

In this section, we introduce the modeling language OMMMA-L [50]. We show how this language extends the standard object-oriented modeling language UML appropriately and allows all aspects of a multimedia application to be modeled in an integral and coherent form.

4.1 UML and its Extensibility Towards Multimedia

The Unified Modeling Language (UML [47], see [11, 49] for an introduction, [21] for an overview) consists of a family of diagrammatic languages which are tailored to modeling diverse aspects of a system. Those are grouped into four categories: use case diagrams, structural diagrams, behavioral diagrams, and implementation diagrams. While use case diagrams are intended for capturing functional requirements of a system, implementation diagrams are used to describe physical system structures and runtime entities. Structural aspects are modeled in class and object diagrams. Behavioral aspects can be described using sequence, collaboration, statechart, and activity diagrams. For the modeling of multimedia applications, we have to analyze how well these diagram types are suited for modeling the architecture of multimedia applications and the four fundamental system views identified in Sect. 2: media and application structure, spatio-temporal presentation, temporal behavior (function), and interactive control.

Use case diagrams and implementation diagrams can be used to model requirements and architectural structure and components, respectively. In the following, we will focus on the fundamental system views within analysis and design models. Thus we only discuss the structural and behavioral diagrams regarding their appropriateness. As it turns out, the structure of an application can be adequately modeled in UML class (and object) diagrams, interactive control can be modeled in statechart diagrams, accompanied by a tailored dialogue

signal hierarchy in class diagrams (although some specific abstractions for dialogue and user interaction specification as they are discussed in Sect. 3.5 may be desirable), and parts of (predefined) temporal behavior can be adequately modeled with UML sequence diagrams. But the analysis of UML's features reveals that specialized and more advanced language constructs are needed to describe the temporal assembling of different objects. Additionally, UML does not offer an explicit notation for spatial modeling in order to specify e.g. the presentation view of user interface layouts intuitively. Finally, UML lacks appropriate pragmatic guidelines on how to deploy the different diagram types cooperatively to model complex multimedia applications. Such guidelines relate to both which diagram types to use for a particular view on the system and how to deploy a particular diagram on a specific level of abstraction, and how the different views and levels of abstraction relate to each other. (Note that we concentrate on a single level of abstraction herein.) These shortcomings have led to the development of an extension of UML towards multimedia entitled OMMMA-L (**O**bject-oriented **M**odeling of **M**ulti**M**edia **A**pplications – the **L**anguage) that captures the application characteristics represented in the different views and to deriving pragmatics on how to model multimedia applications with an object-oriented language based on UML.

The extensions of OMMMA-L can be integrated with UML by deploying UML's built-in extension mechanisms allowing existing model elements to be specialized by stereotypes, constraints, and tagged values (see [47]). These lightweight extensions do not influence the syntax and semantics of the UML itself, but semantics can be specialized for domain-specific extensions. The extensions can then be used to build profiles for specific application domains or kinds of applications.

OMMMA-L is presented in the next subsections, starting by introducing an example application to be modeled.

4.2 OMMMA-L Modeling Example: Automotive Information System

The UML-based modeling language OMMMA-L has been designed to model a wide range of aspects of interactive multimedia applications. We illustrate its capabilities by showing extracts from a model of a simulation application of an automotive information system.

Car cockpits nowadays evolve towards being highly-integrated multimedia information interfaces that interact with many embedded components as well as with external and distributed information systems and services. Diverse applications have to be integrated, like car audio, navigation and communication systems, travel or tourist information, and automotive system monitoring and control.

Regarding interactivity with a human user, several levels of abstraction can be distinguished: on a low level of interaction, a user has to interact with hardware input and output devices (presentation media) that are visual, haptic, or voice-enabled. Input devices produce signals that need to be transformed to

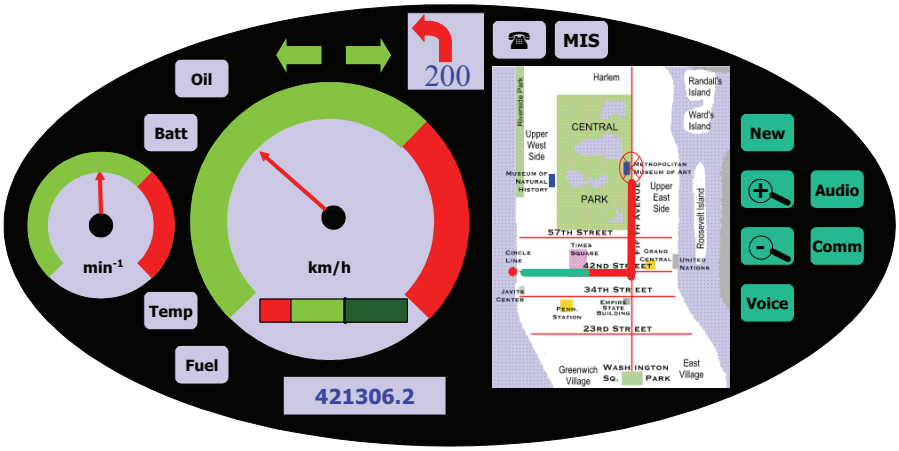


Figure 1: Display of an Automotive Information System

semantic events on application level. For examle, clicking the right mouse button on a specific point on the screen has a specific semantics for the application when it is in a particular state. Pressing a specific button in a multi-functional automotive control panel also shows a context-dependent behavior.

We return to the appropriate aspects of this application in the following subsections in order to illustrate the language concepts of OMMMA-L. The four fundamental views identified in Sect. 2 each relate to a particular diagram type in the OMMMA approach:

- media and application structure are modeled in the class diagram;
- the spatial aspect of the presentation is modeled in OMMMA-L presentation diagrams (that are related to OMMMA-L sequence diagrams for the spatio-temporal integration);
- temporal behavior (function) is modeled in OMMMA-L sequence diagrams; and
- interactive control is modeled in statechart diagrams.

Architectural considerations also appear on these diagrams, although they are not explicitly modeled within these OMMMA-L diagram types. The individual diagrams are presented in the succeeding sections, before we explain the integration of these diagrams in Sect. 4.7.

4.3 Class Diagram

Class diagrams are the core of an object-oriented application model and are used to model the static structure of the multimedia application. Essentially,

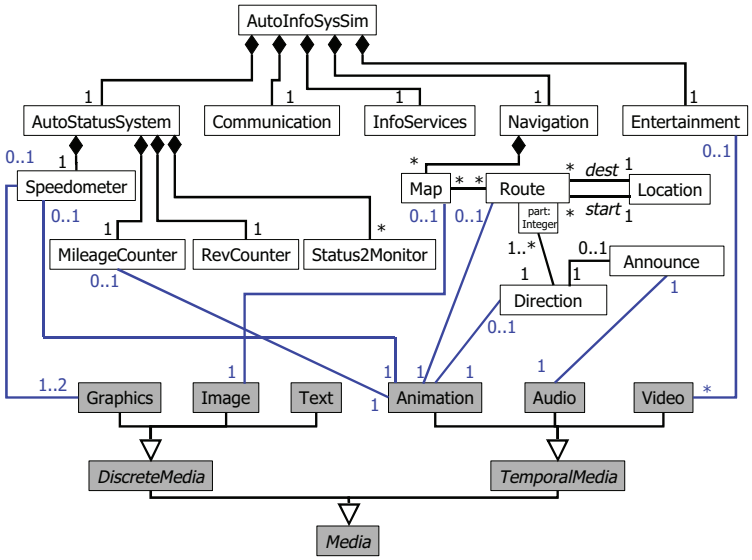


Figure 2: OMMMA-L class diagram

they consist of class and association definitions which describe the structure of objects and their possible structural interrelations. As UML's language features for defining a class diagram are expressive enough, they have been incorporated unchanged into OMMMA-L. But in order to express the two structural model aspects of application semantics and media types, each OMMMA-L class diagram consists of (at least) two closely interrelated parts:

- an *hierarchy of media type definitions*, which comprises classes for all (representation) media types; and
- the logical model of an application, which comprises classes and associations to describe application domain objects and their interrelations.

The two aspects are linked by associations which interrelate application objects with corresponding media objects. For the specification of interaction, these class hierarchies must be accompanied by a *signal hierarchy* as a basis for event-based interaction. *Presentation classes* may be deployed (possibly in a different package) in addition to model the possible composition of user interfaces as a basis for the presentation diagrams introduced in Sect. 4.5. Figure 2 shows a part of the class diagram for the sample application of a simulated automotive information system shown in Fig. 1. The lower part of the diagram depicts the media type hierarchy and the upper part the structure of the logical model. It shows that the automotive information system simulation is a complex composition of five subsystems: an automotive status system and systems for navigation, communication, information services, and entertainment. The

status and navigation systems are more detailed. The status system comprises a speedometer, a mileage counter, a revolution counter, and a set of status monitors. The navigation system contains multiple maps that can be associated with an unrestricted number of routes. In turn, routes can be related to multiple maps. For each route, there is a start and a destination location. A route relates to a set of directions that are qualified from the perspective of a route by a part number defining their position in the sequence of directions. Additionally, a direction may be accompanied by a spoken announcement of the way to drive. For some of these application classes, the associations to elements of the media class hierarchy are shown. The speedometer, for instance, is related to one or two graphics and an animation, e.g. to enable a day and night design of the background and and a moving indicator for presentation of the actual speed. Also, routes shown on the map (which is related to an image) and the directions are realized as animations. An announcement is related to an audio object whereas the (simplified) entertainment system relates to multiple video objects.

OMMMA explicitly distinguishes between application objects as regards content and media objects in order to allow an application to present one application entity by different (representation) media, e.g. accounting for distinct presentation media, such as screens, or resource availability. Thus, media objects are not specializations of application objects or vice versa. The dimension of media types is based on a generalization hierarchy of static and temporal media types as it can be found in several multimedia standards, e.g. MHEG [36], and frameworks, e.g. the framework by Gibbs and Tsichritzis [24] or MET++ [1].

4.4 Extended Sequence Diagram

UML offers various diagram types to model behavioral aspects of an application. Due to their emphasis on modeling temporal sequences (of messages), sequence diagrams are deployed in OMMMA-L to model the (predefined) *temporal behavior* of a multimedia application. But, in order to be able to model specific characteristics of a multimedia application more directly and thus more intuitively, standard UML sequence diagrams are extended by a series of features, especially regarding timing and time constraints. These are for example:

- *Refinement of the time dimension* by defining local time axes for objects supporting a notion of local time. Local time can be related to global (real) time (represented by the actor's timeline) to specify *intra-object* synchronization or to the time of other objects to specify *inter-object* synchronization. Durations and points in time can be specified by different forms of fixed, bounded or unbounded time intervals restricting their possible temporal positions. Time intervals are represented by their start and end points. (Syntactically, these timing requirements can be written as constraints using (in-)equalities or in an interval notation.)
- *Sychronization bars* (bold lines) instead of message arrows between object activations to specify the continuous inter-object synchronization between

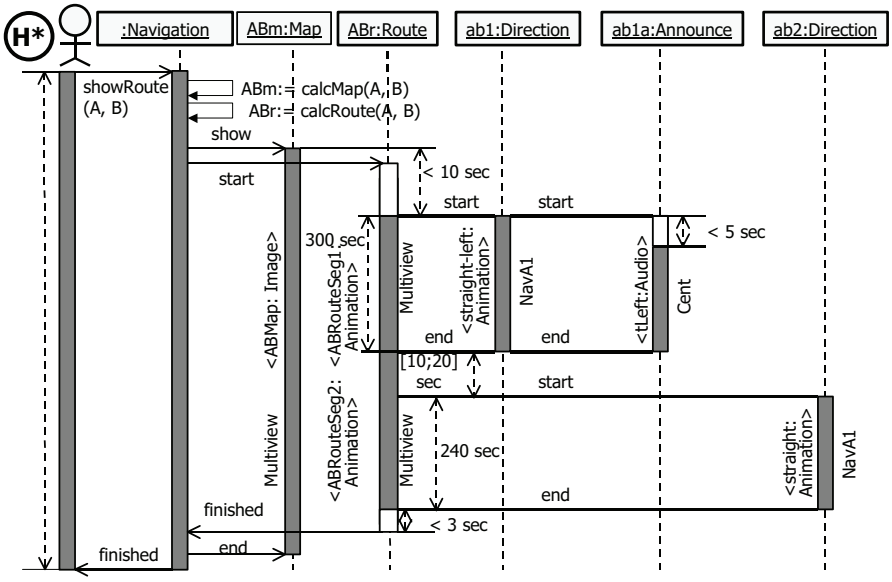


Figure 3: OMMMA-L sequence diagram

temporal media presentations that may abstract from a message direction.

- *Activation and deactivation delays* of media objects in order to model tolerated variations of synchronization relations for media objects (compare maximum start and end times in [31]).

- A notion of shallow and deep *History* on a sequence diagram that UML only provides for statechart diagrams. It allows the designer to specify whether a specified scenario can be interrupted and later returned to at the same point of virtual time where it had been interrupted. Deep history (H*) denotes that this temporal reconstruction is possible even on nested invocation levels with the semantics of pause and resume for the possibly complex presentation, whilst shallow history (H) restricts returning to the situation on the top level with the semantics that the sequence diagram can be restarted from the interruption time, but already started presentations of objects cannot be cued to their last presentation state.

- *Parallely composed activation* of media objects in order to model the simultaneous presentation of an application object across different presentation channels (or media) and/or by different media objects.

- *Sequentially composed activations* resulting in an *automatic triggering* of subsequent activations (segments), e.g. to present an animated object that is sequentially presented via different channels and/or by different media objects.

- Activations of application objects can be annotated with presentation objects, either abstractions from hardware devices (presentation media) or software user interface objects – depending on the level of use – such as audio channels or graphical objects on a screen, and/or media objects designated to represent an application object during an activation or activation segment. Associated media objects, that must conform to the types specified in the class diagram, are enclosed in ⟨⟩. The identifiers of presentation objects appear as pure strings (as they are used on presentation diagrams, cf. Sect. 4.5).

- Activations of objects may be overlayed by *media filters*, which describe time functions, e.g. the incremental increase of an audio level over time.

Each OMMMA-L sequence diagram models the temporal behavior of a predefined *scenario* of the multimedia application. The scenario specified by the sequence diagram is represented by the (initial) message sent from an actor symbol (or some user interface component) to an object within the sequence diagram that acts as the scenario controller. The message can be parameterized, e.g. by time stamps for start and end of execution of a sequence diagram, in order to support its re-use, or by parameters that may be used in guard expressions or nested message calls.

All objects in one diagram relate to the same (global) timeline to which they can be synchronized if required. The projection of a single continuous media object on the corresponding (global) timeline specifies intra-object synchronization. Concurrent scenarios with an independent timeline need to be modeled by different sequence diagrams related to parallel substates within an *and*-superstate of the corresponding statechart diagram (see Sect. 4.6), i.e., they do not have a common notion of time. Different message types between objects enable specification of synchronous or asynchronous messaging. A propagation mechanism has to ensure consistency of temporal specifications or the mapping of a local time axis to its relative temporal coordinate system.

Figure 3 gives an example of an OMMMA-L sequence diagram. It describes the execution of showRoute(Location A, Location B) which is an operation of the navigation system. All objects shown in the horizontal object dimension are semantic, i.e. application objects. Based on the parameters for start and destination (A respectively B), the Navigation object determines a route object ABr and a map object ABm that is then called to be shown. After a maximum activation delay of ten seconds relative to the start of the presentation of the map, an animation of the route from location A to B has to be presented. This animation consists of two parts of which the first one is associated with the media object ABRouteSeg1:Animation that is shown for 300 seconds and the second one is associated with the media object ABRouteSeg2:Animation that is shown for 250 up to 260 seconds. (These presentation time intervals do not necessarily coincide with the duration of the animation objects themselves.) Both animations are presented via a screen object referenced as Multiview (see Fig. 5). Parallel to the first part of the route animation, direction ab1:Direction

is presented at NavA1. It is accompanied by a spoken announcement that must be started at most five seconds after the invocation by the direction object ab1, and whose end is synchronized with the ends of both the first part of the route animation and the corresponding direction ab1. The annoucement is to be output via a center speaker denoted by Cent as specified in Fig. 4. The synchronization bars at the start and the end of these activations specify that a continuous inter-object synchronization is intended and has to be ensured by the renderer at presentation time. The end of the first part of the route animation directly triggers the execution of the second part. After a delay that is between 10 and 20 seconds, another direction animation ab2 is started. It (synchronously) co-ends with the presentation end of the second part of the route animation. After finishing its presentation, the route object has to signal to the navigation component within 3 seconds that presentation is finished.

For the specification of activation and deactivation delays, we use the UML presentation option to distinguish periods of actual computing by shading the activation segments from periods where objects are activated, but do not own the focus of control of the associated thread, by plain activation segments [47].

4.5 Presentation Diagram

Class diagrams are used to model the media and application structure view in OMMMA-L, OMMMA-L sequence diagrams model the temporal behavior (function) view. Before we continue with interactive control in the next subsection, we first explain how the spatial structure of the presentation (view) is modeled in OMMMA-L.

Due to the fact that UML does not offer a diagram type which is well-suited and appropriate for modeling this view, the new *presentation diagram* type is added to OMMMA-L. Presentation diagrams support an intuitive description of the layout, i.e., the spatial arrangement of *presentation objects* at the user interface. Spatial relationships (and constraints) can thus be graphically depicted. In addition, by incorporating the user interface design into the modeling language, consistency relations to other diagram types can be formulated and checked.

The presentation diagrams of OMMMA-L follow the idea of structuring the presentation area of the user interface by boundig boxes for presentation objects (more precisely, these can be roles as in UML collaboration diagrams that can be substituted by conforming objects) that are to be presented. Bounding boxes show geometry and size characteristics and are positioned on a virtual area relative to some specified coordinate system. Presentation objects are distinguished into visualization objects and interaction objects. *Visualization objects* are passive objects that are used to present e.g. text, pictures, graphics, video, or animation objects. *Interaction objects* enable user interactions and may raise events in the running system. Examples are scroll or menu bars, buttons, input fields or a hypertext containing links. Bounding boxes for interaction objects are indicated by bold borders (like active objects are marked in UML). The visual layout specification is accompanied by an iconic representation of audio channels beside the visual presentation area.

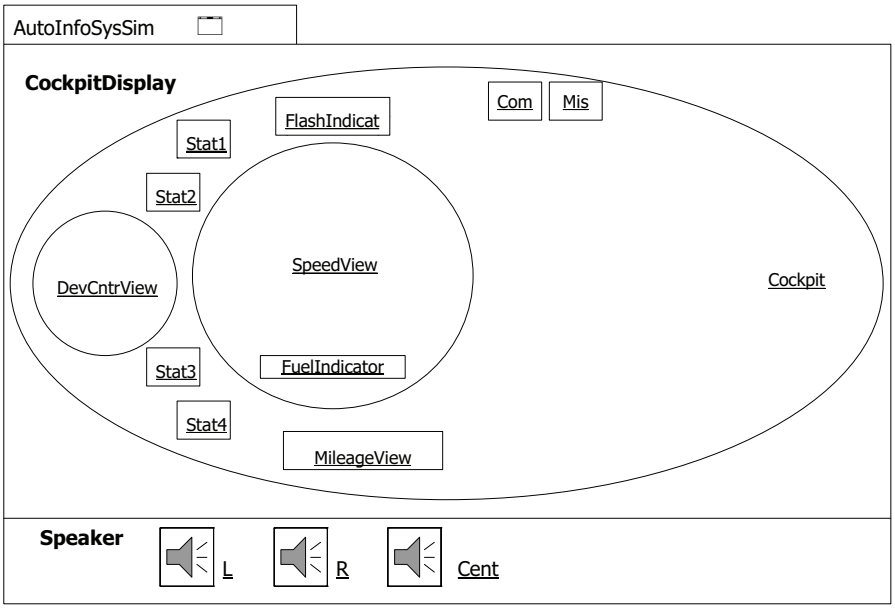


Figure 4: Application-level OMMMA-L presentation diagram

We use the notation of a stereotyped UML package to depict a presentation diagram. The presentation diagram can be further divided into different compartments representing hardware output devices such as different screens or audio channels. In Fig. 4, the presentation diagram is divided in two compartments, one for a dashboard cockpit display and one for the left, right, and center speakers as channels of an audio system (in the same way, local areas for input devices can be described).

Since, in our example, there is no direct interaction with the presentation objects like in direct-manipulative graphical user interfaces, but interaction is via specific input devices such as knobs, all bounding areas are marked as visualization objects (an area for input elements has been omitted).

The complete presentation of a certain application unit may be described by several presentation diagrams that may be composed by a layered placement on the virtual area (e.g. Figs. 4 and 5). Positioning of presentation elements is by convention relative to the directly surrounding element unless otherwise specified, e.g. by a separate hierarchical composition of presentation elements. For instance in Fig. 5, MultiView is positioned relative to Cockpit from the presentation diagram AutoInfoSysSim, given by the path expression on the diagram.

Following this description of spatial modeling of the presentation view, now the interactive control view, modeling system reaction to (user) inputs and other events, is shown.

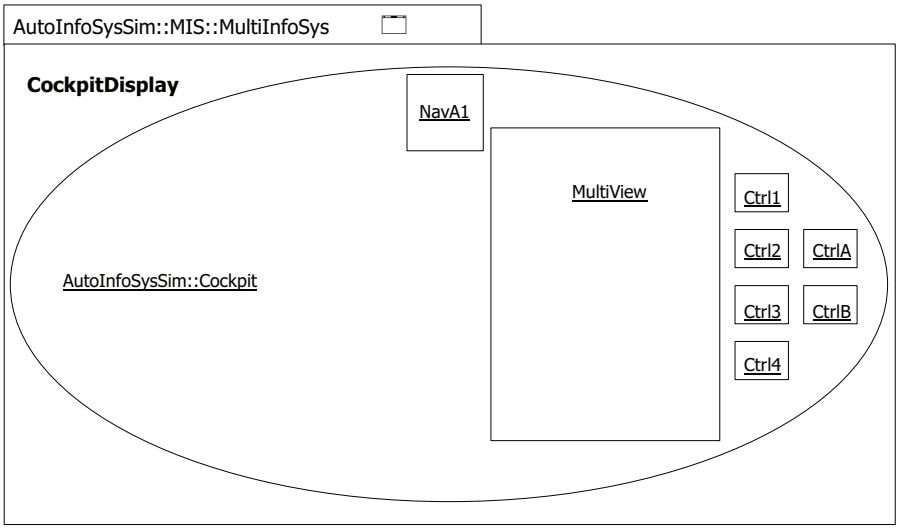


Figure 5: Lower-level OMMMA-L presentation diagram

4.6 Statechart Diagram

In UML, statecharts are assigned to classes to specify the behavior of their instances. They can be used on different levels of granularity. On a high level, they can specify the behavior of an application or substantial parts thereof. On a low level, they can be used to specify the behavior of simple classes, such as user interface element classes, to model e.g. the behavior of a button or the control state of a media object or its associated media player. The high-level statecharts are used to specify the control behavior within the context of the application since it must be specified which event regarding which particular element of the application triggers a transition from a state of the application. Examples of high-level statecharts that are partly schematic for simplification of presentation can be seen in Figs. 6 and 7. They are used on an application-semantic rather than a user interface level.

While OMMMA-L sequence diagrams are used to specify the (predefined) temporal behavior of a multimedia application, statechart diagrams are used to specify the system states as well as state transitions triggered by user interactions or other system events, i.e., the *interactive control* or dynamic behavior. OMMMA-L statechart diagrams are syntactically and semantically equal to UML statechart diagrams. This means that e.g. they may be structured by *and*- and *or*-superstates or refined by embedded statechart diagrams. An action appearing on an OMMMA-L statechart may represent a multimedia scenario (see Sect. 4.4). For example, internal entry- and exit-actions, or do-activities of states may be labeled with names of actions or action expressions.

To enable dialogue specification on an adequate level of abstraction, e.g.

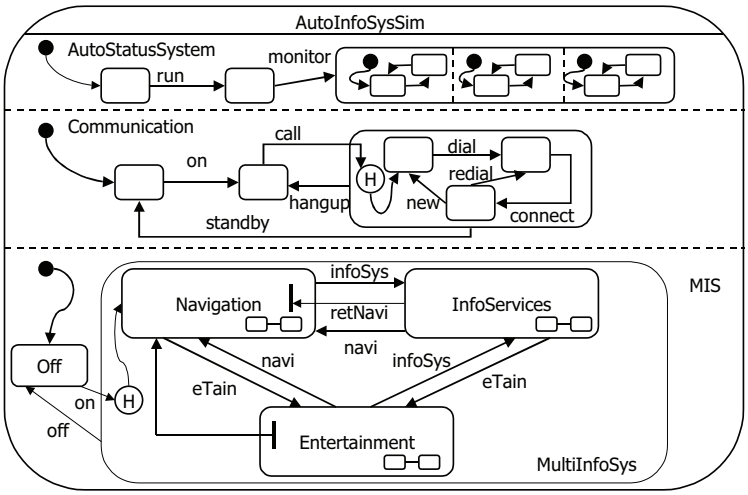


Figure 6: OMMMA-L statechart diagram for the automotive information system application

the selection of navigational alternatives or the control of media playout, the OMMMA-L application model needs to be accompanied by an appropriate signal hierarchy (see [47], cf. Sect. 4.3). This must be based on a spatio-temporal event model for user interactions on the control interface. It should, in perspective, also account for modal parameters of event instances. Events on statechart diagrams relate to such signals. Signals may be categorized according to user interaction events, application events, system events, and timer events. The event model that can be used within OMMMA-L statecharts is not restricted. It is therefore possible to integrate event algebras as they have been specified in the area of active database management systems, that have also been used as a basis for the spatio-temporal event model in [59]. Events can then be composed by algebraic and temporal operators such as and, or, or seq.

This description of the use of statechart diagrams in OMMMA-L concludes the presentation of the individual diagrams.

4.7 Integrated Modeling

Each of the above introduced OMMMA-L diagram types is used to specify a certain view of a multimedia application. What remains is the integration of these views into a coherent model of a multimedia application.

The typological fundament of the different views on the multimedia application is defined in the OMMMA-L class diagram. Other structural and behavioral views must conform to the class definitions and association specifications therein. This implies that objects on OMMMA-L sequence diagrams or – if intended – presentation objects on presentation diagrams must be typed

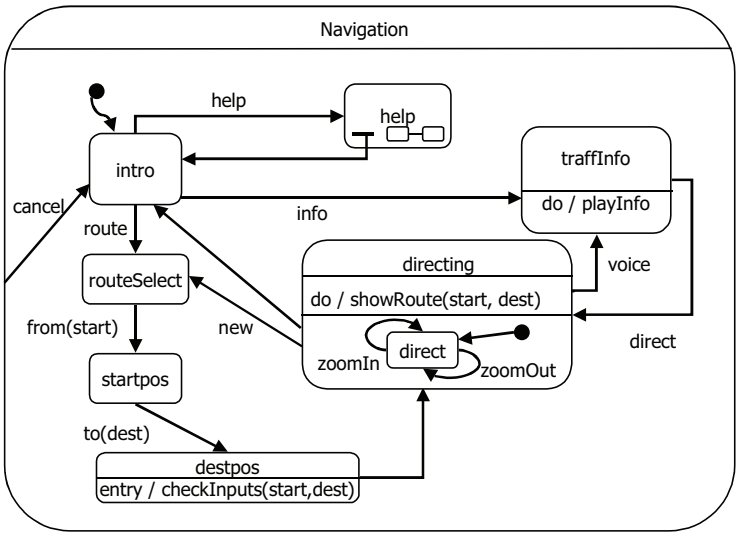


Figure 7: OMMMA-L statechart diagram for the navigation subsystem

over these classes. Statechart diagrams are assigned to these classes for the specification of interactive control or dynamic behavior.

A multimedia application may be more detailed seen as a collection of multimedia application units, so-called *scenarios* or scenes. In the OMMMA approach, each scenario corresponds to a state within an high-level statechart diagram which is associated to the class of the overall application or some class encapsulating a substantial part thereof. Furthermore, each scenario is via its associated state related to a presentation, possibly composed of different presentation diagrams.

A state associated to a scenario may be refined by a nested statechart diagram or nested states (as depicted for state directing in Fig. 7) which describe the possible interactive behavior during this scenario. (Thus, scenarios can be hierarchically composed.) For example, state directing is not left when zooming, implying the semantics that the temporal behavior of the scenario related to *showRoute* is not influenced by zoom operations.

In order to couple the interactive control with the predefined functional behavior views of a multimedia application, actions on a statechart diagram can be associated with OMMMA-L sequence diagrams that specify the scenarios, more precisely, the predefined, timed pieces of functional behavior within a scenario, corresponding to such actions or action expressions. An entry-action or do-activity means that the behavior specified by the sequence diagram is automatically triggered whenever the corresponding state is entered, exit-actions are executed before the state is left. The semantics of the (concurrently executed) do-activity is thereby to be interruptible by events triggering a transition from

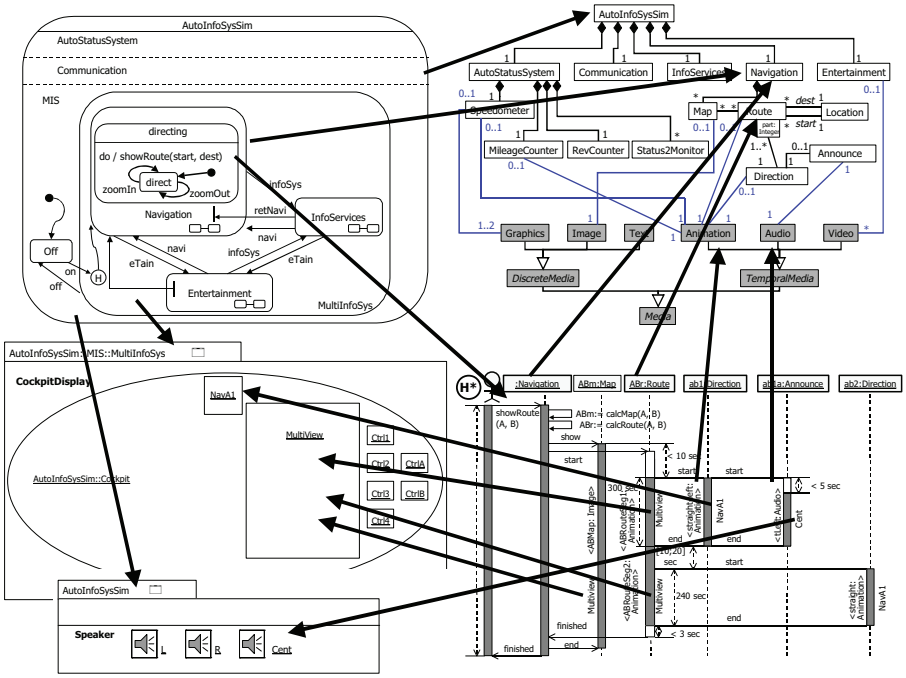


Figure 8: An integrated OMMMA-L specification

that particular state, whilst entry-actions and exit-actions are non-interruptible as specified by the UML run-to-completion semantics (see [47]). Figure 7 gives examples for an entry-action in state destpos and for two do-activities in states traffInfo and directing. The latter do-activity is specified by a sequence diagram with a (deep) history indicator denoting that the assigned scenario can be completely resumed after interruption. This construction is semantically feasible since do-activities on statechart diagrams are executed concurrently. Based on this coupling of function and control views, mutually exclusive substates of an *or*-superstate enable the specification of alternative presentation flows triggered by events on incoming transitions comparable to the timeline-tree model in [31].

A presentation diagram is associated to a state of the interactive control view in order to couple presentation and control views. Therefore, its name coincides with the name of the state to which it is assigned. Semantically, this presentation diagram will be part of the user interface presentation as long as the application or the respective part thereof is in that specific state. In Fig. 4, the presentation diagram is labeled with the name of the top-level state AutoInfSysSim, meaning that these objects are addressable for presentation as long as the application is running.

The composition of a complete presentation of a certain application unit (or state) is based on the hierarchical composition of states to which the con-

stituent presentation diagrams are assigned. Figure 5 shows a presentation diagram that is assigned to state MultiInfoSys which is a nested substate of AutoInfoSysSim (via state MIS) as can be seen from the path expression in the name compartment in the upper left. When the application is in state AutoInfoSysSim::MIS::MultiInfoSys or a substate thereof, the complete presentation is composed of (at least the given) two presentation diagrams, i.e., the general diagram for all application states and the diagram for the multi-information system being active (compare Fig. 6).

In OMMMA-L sequence diagrams, an activation of an application object can reference a media object that is being presented during this activation as well as a presentation object that may be used on a reachable presentation diagram in order to specify the spatio-temporal constraints of the presentation to the user. Reachable presentation diagrams are defined by the coupling statechart diagram. They either relate to the same state to which the action belongs that is specified by the sequence diagram – either an internal action of that state or an action on a transition within that state in case of a complex state – or to a superstate of that state.

Figure 8 gives a simplified example of the interrelations between diagrams in such a complete specification where small parts of each diagram type are depicted. Diagrams are interrelated by using the same identifier names in different diagrams. Examples are the name of a specified scenario (initial message) in a sequence diagram used as the action expression of an internal action within a state of the state diagram, or the name of a state used as the identifier of a presentation diagram, or the name of a (visual) presentation object or audio channel on a presentation diagram used within a sequence diagram in association to an activation box. Other relations (consistency constraints) between diagrams are depicted by overlayed arrows in Fig. 8. These constraints are precisely specified in a UML profile for multimedia applications that extends standard capabilities of UML for the modeling of multimedia applications according to the OMMMA approach.

5 Future Directions

The modeling approaches presented herein still have to show their feasibility in real-world applications and complex application settings. Case studies are necessary to underpin their real achievements in practice.

The extension of UML by profiles for specific application domains has been widely recognized as indicated by a diverse range of (prototypical) custom profiles, e.g. for architectures of web applications [14], and several requests for proposals issued by the OMG in order to standardize profiles for several domains, like embedded real-time applications and CORBA. A major drawback still is the absence of generally accepted, formal and precise specifications of UML semantics, and, therefore, also for most proposed extensions of UML. But with the forthcoming appearance of diverse profiles, the issues of consistency between and semantically sound integration of profiles need to be analyzed in

detail. Orthogonal aspects should be placed in isolated profiles that could then more easily be combined. For the profiles presented herein, the integration of user interface, hypermedia, and multimedia modeling extensions is an interesting challenge.

A recent trend in the specification of multimedia applications is the use of constraint-based approaches to more flexibly describe the requirements and properties of multimedia presentations (see e.g. [40, 7, 56, 26]). Some constraints can be graphically expressed in the current OMMMA-L notation, other constraints can be integrated into a model by UML's built-in constraint language OCL [47] or by using other textual or diagrammatic (for instance, constraint diagrams [37]) constraint languages.

Some interesting challenges exist regarding the integration of the proposed object-oriented modeling approach into general multimedia software development processes. Especially the transformation between the model and an implementation, based on either object-oriented frameworks or authoring systems, that does account for both structure and behavior, is an obvious task at hand to prove the feasibility of the concept.

6 Conclusion

In this chapter, we presented approaches for the object-oriented modeling of interactive, hypermedia, and multimedia applications. We also examined the current state of application development based on multimedia authoring systems and object-oriented programming. By doing this, we illuminated some important aspects of the multi-dimensional task of multimedia software engineering.

In our presentation, we focused on OMMMA-L, a visual, object-oriented modeling language for multimedia applications. OMMMA-L is based on the standard modeling language UML. New language features have been incorporated into OMMMA-L in order to support integrated modeling of all aspects of a multimedia application. Particularly, a presentation diagram type and appropriate extensions to sequence diagrams have been introduced.

The modeling language is accompanied by a method description how to deploy the language elements in a multimedia software development process and a prototype implementation. Furthermore, the language extensions are being formalized by a precise semantics specification based on graphical operational semantics (cf. [20]). As a refined conceptual basis, we intend to define a formal model for the composition within and between the different behavioral diagrams.

References

[1] P. Ackermann, *Developing Object-Oriented Multimedia Software — Based on MET++ Application Framework* (dpunkt, Heidelberg, 1996).

[2] Apple, Hypercard, *http://www.apple.com/hypercard/*

[3] T. Arndt, "The evolving role of software engineering in the production of multimedia applications", *Proceedings of the IEEE International Conference on Multimedia Computing and Systems* (*ICMCS'99*) **I** (1999) 79–84.

[4] B. Bailey, J. A. Konstan, R. Cooley and M. Dejong, "Nsync — A toolkit for building interactive multimedia presentations", *Proceedings of the 6th ACM International Conference on Multimedia'98* (ACM Press, 1998) 257–266.

[5] H. Baumeister, N. Koch and L. Mandel, "Towards a UML extension for hypermedia design, eds. R. France and B. Rumpe, *Proceedings of the «UML»'99 — The Unified Modeling Language, Beyond the Standard, 2nd International Conference*, *Lecture Notes in Computer Science 1723* (Springer, 1999) 614–629.

[6] E. Bertino and E. Ferrari, "Temporal synchronization models for multimedia data", *TKDE* **10**, no. 4 (1998) 612–631.

[7] E. Bertino, E. Ferrari and M. Stolf, "MPGS: An interactive tool for the specification and generation of multimedia presentations", *TKDE* **12**, no. 1 (2000) 102–125.

[8] A. Bianchi, P. Bottoni and P. Mussio, "Issues in design and implementation of multimedia software systems", *Proceedings of the IEEE International Conference on Multimedia Computing and Systems* (*ICMCS'99*) **I** (1999) 91–96.

[9] M. M. Blattner and E. P. Glinert, "Multimodal integration", *IEEE MultiMedia* **3**, no. 4 (1996) 14–24.

[10] D. Boles, P. Dawabi, M. Schlattmann, E. Boles, C. Trunk and F. Wigger, "Objektorientierte Multimedia-Softwareentwicklung: Vom UML-Modell zur Director-Anwendung am Beispiel virtueller naturwissenschaftlich-technischer Labore", *Proceedings of the Workshop Multimedia-Systeme, 28th Annual Conference of the German Computer Science Association* (*GI*) (1998) 33–51 (in German).

[11] G. Booch, J. Rumbaugh and I. Jacobsen, *The Unified Modeling Language User Guide* (Addison-Wesley, Reading, MA, 1998).

[12] S.-K. Chang, *Multimedia Software Engineering* (Kluwer, Boston, MA, 1999).

[13] Click2Learn, Toolbook II, *http://home.click2learn.com/*

[14] J. Conallen, *Building Web-Applications with UML* (Addison-Wesley, Reading, MA, 2000).

[15] L. L. Constantine and L. A. D. Lockwood, *Software for Use: A Practical Guide to the Models and Methods of Usage-Centered Design* (ACM Press, New York, NY, 1999).

[16] J. Coutaz, "PAC-ing the architecture of your user interface", eds. M. D. Harrison and J. C. Torres, *Design, Specification and Verification of Interactive Systems, Proceedings of the 4th Eurographics Workshop '97* (Springer, 1997) 13–27.

[17] R. Depke, G. Engels, K. Mehner, S. Sauer and A. Wagner, "Ein Vorgehensmodell für die Multimedia-Entwicklung mit Autorensystemen", *Informatik: Forschung und Entwicklung* **14** (1999) 83–94 (in German).

[18] D. Dingeldein, "Modeling multimedia objects with MME", *Proceedings of the EUROGRAPHICS Workshop on Object-Oriented Graphics* (*EOOG'94*), 1994.

[19] M. Echiffre, C. Marchisio, P. Marchisio, P. Panicciari and S. Del Rossi, "MHEG-5 — Aims, concepts, and implementation issues", *IEEE MultiMedia* **5**, no. 1 (1998) 84–91.

[20] 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 [22], pp. 323–337.

[21] G. Engels, R. Heckel and S. Sauer, "UML — A universal modeling language?" eds. M. Nielsen and D. Simpson, *Proceedings of the 21st International Conference on Application and Theory of Petri Nets* (*ICATPN 2000*), *Lecture Notes in Computer Science 1825* (Springer, 2000) 24–38.

[22] A. Evans, S. Kent and B. Selic, *Proceedings of the «UML» 2000 — The Unified Modeling Language, Advancing the Standard, 3rd Intl. Conf.*, *Lecture Notes in Computer Science 1939* (Springer, 2000).

[23] F. Garzotto, P. Paolini and D. Schwabe, "HDM - A model-based approach to hypertext application design", *TOIS* **11**, no. 1 (1993) 1–26.

[24] S. J. Gibbs and D. C. Tsichritzis, *Multimedia Programming: Objects, Environments and Frameworks* (Addison-Wesley, Wokingham, 1995).

[25] R. Gordon and S. Talley, *Essential JMF: Java Media Framework*, (Prentice-Hall, Englewood Cliffs, NJ, 1999).

[26] V. Hakkoymaz, J. Kraft and G. Özsoyoglu, "Constraint-based automation of multimedia presentation assembly", *Multimedia Systems* **7**, no. 6 (1999) 500–518.

[27] R. Hennicker and N. Koch, "A UML-based methodology for hypermedia design", in [22], pp. 410–424.

[28] I. Herman, N. Correia, D. A. Duce, D. J. Duke, G. J. Reynolds and J. van Loo, "A standard model for multimedia synchronization: PREMO synchronization objects", *Multimedia Systems* **6**, no. 2 (1998) 88–101.

[29] I. Herman, G. J. Reynolds and J. van Loo, "PREMO: An emerging standard for multimedia presentation — Part I: Overview and framework", *IEEE MultiMedia* **3**, no. 3 (1996) 83–89.

[30] I. Herman, G. J. Reynolds and J. van Loo, "PREMO: An emerging standard for multimedia presentation — Part II: Specification and applications", *IEEE Multimedia*, **3**, no. 4 (1996) 72–75.

[31] N. Hirzalla, B. Falchuk and A. Karmouch, "A temporal model for interactive multimedia scenarios", *IEEE MultiMedia* **2**, no. 3 (1995) 24–31.

[32] P. Hoschka, "An introduction to the synchronized multimedia integration language", *IEEE MultiMedia* **5**, no. 4 (1998) 84–88.

[33] T. Isakowitz, E. Stohr and P. Balasubramanian "RMM: A methodology for structured hypermedia design", *CACM* **38**, no. 8 (1995) 34–44.

[34] ISO/IEC JTC1/SC24/WG6 — Multimedia Presentation and Interchange, PREMO (ISO/IEC 14478), *http://www.iso.ch*

[35] ISO/IEC JTC1/SC29/WG11 — Moving Picture Experts Group (MPEG), MPEG-4 (ISO/IEC 14496), *http://www.iso.ch*, *http://www.cselt.it/mpeg/*

[36] ISO/IEC JTC1/SC29/WG11 — Multimedia and Hypermedia Expert Group, MHEG (ISO/IEC 13522), *http://www.iso.ch*

[37] S. Kent, "Constraint diagrams: Visualising invariants in OO modelling", *Proceedings of the OOPSLA'97* (ACM Press, 1997) 327–341.

[38] S. Kovacevic, "UML and user interface modeling", eds. J. Bézivin and P.-A. Muller, *Proceedings of the «UML» '98 — The Unified Modeling Language, Beyond the Notation, 1st International Workshop*, *Lecture Notes in Computer Science 1618* (Springer, 1998) 253–266.

[39] G. E. Krasner and S. T. Pope, "A cookbook for using the model-view-controller user interface paradigm in Smalltalk-80", *Journal of Object-Oriented Programming* **1**, no. 3 (August/September 1988) 26–49.

[40] Y.-M. Kwon, E. Ferrari and E. Bertino, "Modeling spatio-temporal constraints for multimedia objects", *Data and Knowledge Engineering* **30**, no. 3 (1999) 217–238.

[41] T. D. C. Little and A. Ghafoor, "Synchronisation and storage models for multimedia objects", *IEEE Journal on Selected Areas in Communications* **8**, no. 3 (April 1990) 413–427.

[42] Macromedia, Authorware, *http://www.macromedia.com/software/authorware/*

[43] Macromedia, Director, *http://www.macromedia.com/software/director/*

[44] Microsoft, DirectX, *http://www.microsoft.com/directx/*

[45] M. Mühlhäuser, "Issues in multimedia software development", *Proceedings of the International Workshop on Multimedia Software Development* (IEEE Computer Society, 1996) 2–9.

[46] N. J. Nunes and J. F. e Cunha, "Towards a UML profile for interaction design: The Wisdom approach", in [22], pp. 101–116.

[47] Object Management Group, *OMG Unified Modeling Language Specification*, version 1.3, June 1999, *http://www.omg.org*

[48] P. Pauen, V. Voss and H.-W. Six, "Modelling hypermedia applications with HyDev", eds. A. A. Sutcliffe, J. Ziegler and P. Johnson, *Designing Effective and Usable Multimedia Systems, Proceedings of the IFIP 13.2 Working Conference* (Kluwer, 1998).

[49] J. Rumbaugh, I. Jacobson and G. Booch, *The Unified Modeling Language Reference Manual*, *Object Technology Series* (Addison-Wesley, Reading, MA, 1999).

[50] S. Sauer and G. Engels, "Extending UML for modeling of multimedia applications", eds. M. Hirakawa and P. Mussio, *Proceedings of the IEEE Symposium on Visual Languages* (*VL'99*), (IEEE Computer Society, 1999) 80–87.

[51] D. Schwabe and G. Rossi, "An object oriented approach to web-based applications design", *Theory and Practice of Object Systems* **4**, no. 4 (1998) 207–225.

[52] T. K. Shih, S.-K. Chang, and P. Shih, "A web document development paradigm and its supporting environment", *Proceedings of the 6th International Conference on Distributed Multimedia Systems* (*DMS'99*), 1999.

[53] P. P. da Silva and N. W. Paton, "UMLi: The Unified Modeling Language for interactive applications", in [22], pp. 117–132.

[54] B. C. Smith, L. A. Rowe, J. A. Konstan and K. D. Patel, "The Berkeley Continuous Media Toolkit", *Proceedings of the 4th ACM International Conference on Multimedia'96* (ACM Press, 1996) 451–452.

[55] L. F. G. Soares, R. F. Rodrigues and D. C. Muchaluat Saade, "Modeling, authoring and formatting hypermedia documents in the HyperProp system", *Multimedia Systems* **8**, no. 2 (2000) 118–134.

[56] J. Song, G. Ramalingam, R. E. Miller and B.-K. Yi, "Interactive authoring of multimedia documents in a constraint-based authoring system", *Multimedia Systems* **7**, no. 5 (1999) 424–437.

[57] Sun Microsystems, Java Media API, *http://java.sun.com/products/java-media/*

[58] Sun Microsystems, Java Media Framework, *http://java.sun.com/products/java-media/jmf/*

[59] M. Vazirgiannis, *Interactive Multimedia Documents — Modeling, Authoring, and Implementation Experiences*, *Lecture Notes in Computer Science 1564* (Springer, Berlin, 1999).

[60] P. Vilain, D. Schwabe and C. S. de Souza, "A diagrammatic tool for representing user interaction in UML", in [22], pp. 133–147.

[61] T. Wahl and K. Rothermel, "Representing time in multimedia systems", *Proceedings of the IEEE 1st International Conference on Multimedia Computing and Systems* (*ICMCS'94*) (IEEE Computer Society, 1994) 538–543.

[62] World Wide Web Consortium (W3C), Synchronized Multimedia Integration Language (SMIL), *http://www.w3.org/AudioVideo/*

[63] A. Zamperoni, "GRIDS — Graph-based integrated development of software: Integrating different perspectives of software engineering", *Proceedings of the 18th International Conference on Software Engineering* (*ICSE*) (IEEE Computer Society, 1996) 48–59.

UML-based Behavior Specification of Interactive Multimedia Applications

Stefan Sauer and Gregor Engels
University of Paderborn, Dept. of Mathematics & Computer Science, D 33095 Paderborn, Germany
{*sauer*|*engels*}*@uni-paderborn.de*

Abstract

Availability of precise, yet usable modeling languages is essential to the construction of multimedia systems based on software engineering principles and methods. Although several languages have been proposed for the specification of isolated multimedia system aspects, there not yet exists an integrated modeling language that adequately supports multimedia software development in practice. We propose an extension of the Unified Modeling Language (UML) for the integrated specification of multimedia systems based on an object-oriented development method. Since integration of co-existing timed procedural and interactive behavior is at the heart of multimedia systems, we focus on UML-based specification of behavior in this paper. In addition, we outline how these behavioral aspects are to be integrated with media, presentation, and software architecture modeling to achieve a coherent and consistent model.

Keywords: UML, interactive multimedia, behavior specification, integrated modeling

1 Introduction

In addition to the areas of interactive games and entertainment, interactive multimedia applications are gaining increasing importance in traditional areas of software systems such as information systems as well. As an effect, many software and multimedia researchers and practitioners advocate deploying software engineering principles and methods for the construction of multimedia systems (see, e.g., [3, 9]). Essential to such approaches is the demand for semantically precise, yet syntactically usable modeling notations that support different views and levels of abstraction. Visual and diagrammatic languages are defined to exactly fulfill these requirements.

Many different aspects need to be integrated to coherently model a multimedia application. The temporal integration and synchronization of diverse media objects with different timing characteristics is the most important feature of multimedia applications in contrast to other interactive applications. Obviously, when we model application behavior, *timed procedural* behavior, i.e., behavior with predefined temporal characteristics, cannot be viewed in isolation unless applications are non-interactive and thus do not react to external events that dynamically occur in a non-predictable manner at execution time. Otherwise, integration of co-existing timed procedural and *interactive* behavior is a key feature that needs to be accounted for during the construction of multimedia software. Therefore, it is the objective of this work to present an integrated modeling technique for timed procedural and interactive behavior.

Different models and notions have been proposed for timed procedural and interactive multimedia behavior (e.g., [11, 10]). Nevertheless, the lasting disadvantage of many approaches is that they purely focus on behavior, but it is not specified how they are to be integrated with other aspects. As a consequence, object-oriented models (e.g., [5]) and (conceptual) frameworks (e.g., [1, 8]) have been proposed that enable the integrated specification of application structure and behavior. A disadvantage of these approaches is that they either require thorough knowledge of the object-oriented paradigm since they are not designed as modeling languages for the practical software development process and are thus not combined with an intuitive graphical notation, or they are directly implemented in programming frameworks and do only run in their proprietary environment. Another problem is the lack of higher level abstractions allowing to model core functionality before getting into the details of a software design.

To overcome these shortcomings, we build on and extend the Unified Modeling Language (UML; [12]) for the specific requirements of application modeling in the multimedia domain. UML allows to model diverse system aspects by a family of diagram types based on the paradigm of object-orientation. In [13] we gave an overview of the OMMMA (Object-oriented Modeling of MultiMedia Applications) approach and modeling language that constitutes the starting point for the more detailed presentation of integrated behavior modeling herein. We restrict the presentation to an analysis level of modeling, not making use of UML's complete modeling capabilities.

The contribution of this work to the field of applying visual modeling to multimedia software engineering is threefold: First, we present an integrated modeling language (method) for the specification of multiple aspects of multimedia applications. This approach was outlined in [13], but herein we clarify the details of behavior specification and the integration of interactive and timed procedural behavior. Second, the presented modeling approach does not invent a new language, but extends the standard object-oriented modeling language UML deploying UML's built-in extension mechanisms. Thus, it supports the evolution of traditional application models into multimedia application models without a paradigm or language shift as well as the migration of software developers that are familiar with object-oriented methods and UML modeling towards multimedia application development. Third, the visual modeling language is also understandable for members of multimedia developer teams other than programmers due to more intuitive notations and reflection of established metaphors and authoring concepts.

The presentation in this paper is structured in the following way: We start with an overview of the integrated modeling of multimedia applications based on UML focusing on structural and basic architectural findings. In Sect. 3, we demonstrate how to apply statechart diagrams to modeling of interactive behavior. Section 4 presents an extended form of sequence diagrams for the specification of timed procedural behavior. In Sect. 5, it is shown how the partial models for these behavioral aspects are to be integrated. In the following section, we compare the modeling approach to selected related work in this area. Finally, we summarize the presented achievements and outline future perspectives.

2 Integrated Modeling of Multimedia Applications Based on UML

UML comprises a family of sublanguages that are each tailored to modeling specific aspects of a software system. The variety of diagram languages reaches from the specification of requirements (use cases, class and activity diagrams), to the integrated modeling of structure, functionality, and dynamics (class, collaboration, sequence, and statechart diagrams), and the modeling of software and hardware architecture (component and deployment diagrams).

UML is designed as a general-purpose modeling language that is independent of a particular software development *process* (or *method*). The concrete use of the diagram languages and their interpretation must be fixed by prescribing a process during the activity of process modeling. Technically, process-specific variants of the diagram types for different modeling purposes, e.g., views and levels of abstractions, can be defined by the built-in extension mechanisms and the concept of UML profiles.

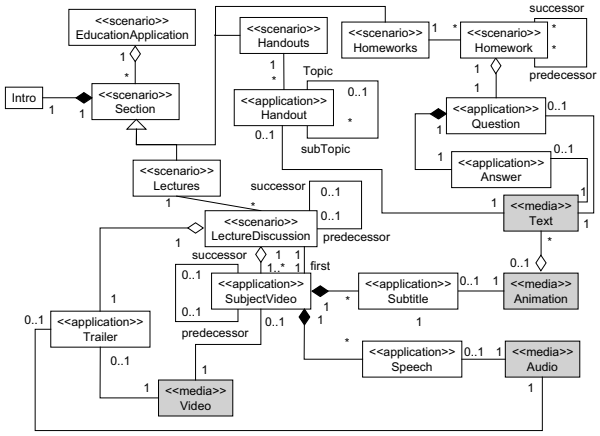


Figure 1. The class diagram of the education application shows ≪scenario≫, ≪application≫, and associated ≪media≫ classes

Analogously, UML's extension mechanisms support the definition of *domain-specific adaptions* of the language elements in order to adapt and specialize the modeling capabilities to domain-specific requirements. Within this paper, we make use of this built-in extensibility for tailoring UML to the multimedia domain, particularly the modeling of interactive multimedia application behavior (on analysis level), without stressing an overall multimedia software development process.

We demostrate how UML can be adapted towards the specification of interactive multimedia aplications by building on the OMMMA (Object-oriented Modeling of Multi-Media Applications) approach that we presented in [13].

The modeling approach is illustrated deploying an example application taken from Adali et al. [2]. They present an educational application for organizing course material that consists of three independent interactive presentations. The first presentation contains three homeworks. Each homework consists of a question and an answer. The second presentation is a sequential list of videos representing parts of lecture discussions, each referring to a specific subject. The third presentation contains a hierarchically structured set of textual handouts. In addition, they specify rudimentary temporal and spatial constraints for these presentations.

Figure 1 depicts the UML class diagram for this application. We extended the example to account for the modeling of additional features not captured in the original model. For the purpose of modeling multimedia applications, the class diagram distinguishes the semantic part of the application and the media types deployed to present the content of multimedia objects. The latter are marked by the stereotype ≪media≫. Other stereotypes are used to distinguish different types of (semantic) application objects. The stereotype

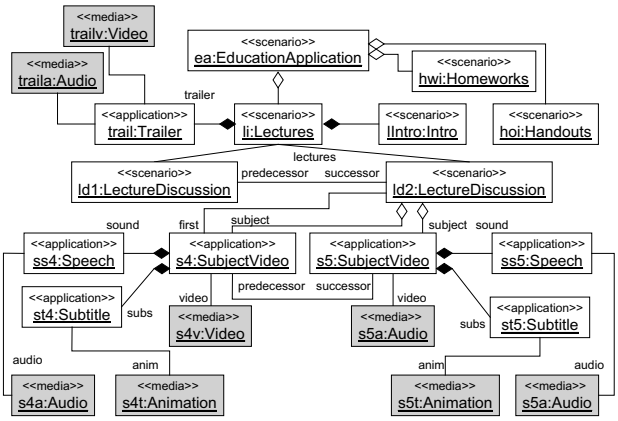


Figure 2. Object diagram showing instances for the second lecture discussion

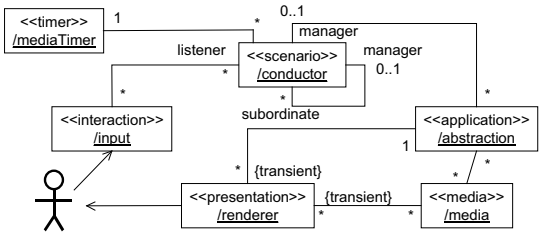


Figure 3. General architectural pattern for interactive multimedia applications

≪application≫ is used to distinguish application classes that correspond to multimedia information from general application classes, carrying no stereotype. This enables the integrated modeling of media and non-media application elements. Furthermore, the stereotype ≪scenario≫ marks classes of objects that represent complex scenarions, i.e., composite parts of the interactive multimedia application that involve several ≪application≫ objects with temporal and spatial relationships. This distinction will be used for the separation of behavioral aspects in the following sections.

Part of an object diagram in the context of this class diagram is shown in Fig. 2. It depicts the application and media objects for a part of the presentation of lecture discussions. Adali et al. structure this presentation into three subsequent parts: an introduction, a first discussion consisting of three sequential subjects, and a second discussion containing two sequential subjects. In our model, the lecture discussion presentation is embodied by object li:Lectures of stereotype ≪scenario≫. This is associated with instances of class LectureDiscussion representing the two discussions. The detailed structure of such a discussion is only depicted for the second discussion. In contrast to [2], we extend the videos that constitute the discussions by distinguishing a visual presentation, an associated audio track, and an animation for the presentation of subtitles. We thus account for the synchronization of continuous media objects. Each of the ≪application≫ objects is associated with a ≪media≫ object as it is specified by the class diagram.

After presenting the structure of the application, we now consider its behavior. The modeling approach is based on the conceptual idea that users interact with the multimedia application via specific user interface components for input (control) and output (presentation). User input events are pre-processed and forwarded to functional controllers that enact and manage (complex) scenarios of multimedia applications. The presentation of multimedia content is achieved by collaborating application and media objects and renderer components that execute the actual presentation.

The fundamental pattern of interaction within the multimedia application is depicted as a UML collaboration diagram on the specification level, i.e., containing roles instead of individually identified objects, in Fig. 3. The user of the system is depicted by a stick man, the UML symbol for an actor that interacts with the modeled system. Users can only interact with objects of stereotypes ≪interaction≫ and ≪presentation≫ for input and output, respectively. User input events are handled by ≪scenario≫ objects which act as event listeners as well as controller objects for the multimedia application behavior. Objects of stereotype ≪scenario≫ can be hierarchically structured and forward/delegate events to other ≪scenario≫ objects. The behavior described so far is inherently reactive and will be modeled within statechart diagrams (see Sect. 3).

In addition to the communication with other ≪scenario≫ objects, objects of this stereotype are also responsible for enacting (timed) procedural behavior. This is achieved by sending method calls to ≪application≫ objects in a timely fashion. The interaction between ≪scenario≫ and ≪application≫ objects is modeled within an extended version of UML sequence diagrams (see Sect. 4). To obtain timing information, the ≪scenario≫ objects are associated with a media timer of stereotype ≪timer≫.

Finally, in order to present the multimedia information to the user, an ≪application≫ object and its associated ≪media≫ objects are temporarily associated with ≪presentation≫ objects for the duration of the presentation of the media content. We abstract from this last pattern of interaction between application, media, and presentation objects within sequence diagrams on the analysis level of our method, but it may be modeled on a less abstract design model.

We thus separate reactive behavior from timed procedural behavior as two different modeling views of a multimedia application in the following. The former is modeled

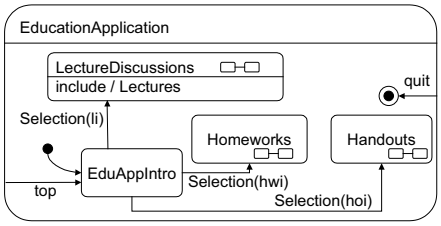


Figure 4. Statechart diagram modeling the top-level reactive behavior of the education application

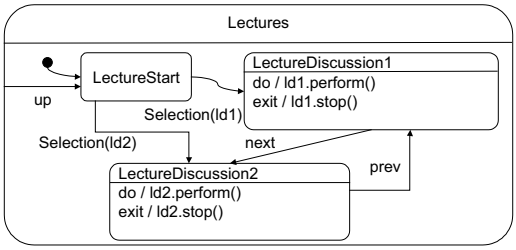


Figure 5. Statechart diagram for the lecture discussion presentation

within statechart diagrams with the usual semantics, the latter is modeled within extended sequence diagrams.

3 Modeling Interactive Behavior in Statechart Diagrams

User interaction appears asynchronously and may affect both the temporal and the spatial composition of objects at presentation time. Interactive behavior is best represented by an event-based model where a user event triggers some action within the system, possibly mediated by a cascade of events sent between objects within the system. UML contains statechart diagrams for modeling event-based reactive behavior. For the modeling of multimedia applications, three basic alternative uses of statecharts can be distinguished: modeling the overall system behavior on a high level of abstraction modeling the behavior of active objects controlling scenarios of media presentations, and modeling the internal state of media, application or presentation objects during the presentation lifecycle. These models have different purposes and appear on different levels of abstraction. For the objective of this paper, we concentrate on the first and second alternative. In these cases, statemachines are assigned to ≪scenario≫ objects.

The statechart diagrams for the example are depicted in Figs. 4–6. We show the top-level statechart of the educational application and a more detailed view of the nested statemachine corresponding to the lecture discussion presentation. In order to present the integration of interactive and procedural behavior on a detailed level, we have added more possibilities for user control to the original example. The events appearing as triggers on the state transitions correspond to signals issued by corresponding user interface, i.e., interaction components, either referring to selection or navigation actions.

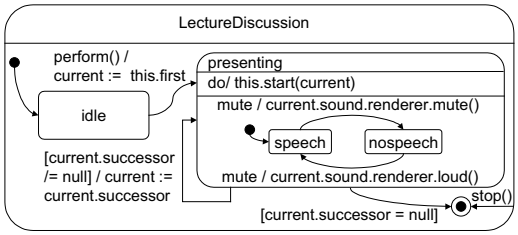


Figure 6. Statechart diagram for a video presentation in a single lecture discussion

Figure 4 shows the top-level statechart diagram for the modeled education application. States within this top-level diagram correspond to scenarios of the interactive multimedia application. In this example, four alternative scenarios that are modeled as substates of the composite top-level XOR-state can be selecteded by user input. Note that it is also possible to specify concurrent execution of multiple scenarios by deploying composite AND-states. Icons on the states *LectureDiscussions*, *Homeworks*, and *Handouts* show that these states are themselves composite states. The UML notation for referring to a nested statemachine *Lectures* from a sub-machine state is only shown for the further refined state *LectureDiscussions* (i.e., include in its internal transition compartment). The events triggering transitions within this statechart are handled by the top-level ≪scenario≫ object of class EducationApplication to which this statechart diagram is assigned by name.

We next consider the nested statemachine corresponding to the presentation of lecture discussions. The statechart diagram in Fig. 5 shows that this scenario consists of three different substates, one for the introduction and one for each contained lecture discussion. The internal activities of states *LectureDiscussion1* and *LectureDiscussion2*, labeled with the predefined action label *do*, model which activity is undertaken by the Lectures object whenever it is in one of these states. In case of LectureDiscussion2, it sends the message *perform()* to the object ld2 which is a ≪scenario≫ object of type LectureDiscussion. The reaction of ld2 is modeled in a separate statechart diagram (Fig. 6). This case thus models the coordination of activity between different (active) objects by signaling events. The

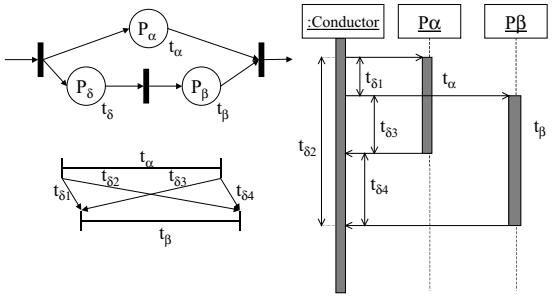


Figure 7. A comparison of generic models for temporal constraints on intervals between OCPN (upper left), interval relations (lower left), and UML sequence diagrams

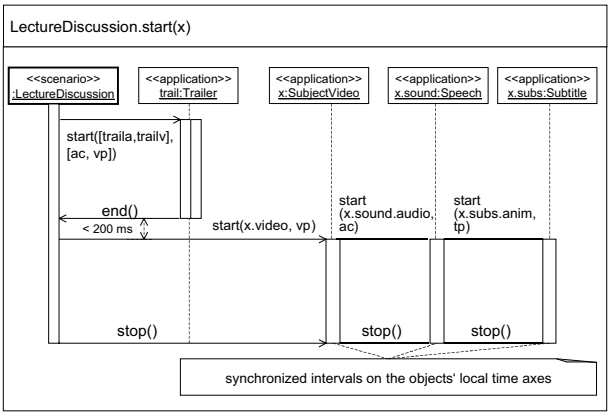


Figure 8. Sequence diagram modeling timed procedural behavior for the presentation of lecture discussion videos

semantics of the *do*-activity is that this behavior is interruptible whenever an event occurs that triggers any transition from this state. In contrast, UML also offers predefined internal action labels for *entry* and *exit*-actions that are run to completion. To ensure that the media presentation, concurrently activated by the *do*-activity, is actually stopped on interrupts, an *exit*-action is executed when such a preemptive interaction occurs. Further details of the statechart diagram for *LectureDiscussion* will be explained in Sect. 5.

4 Modeling Timed Procedural Behavior in Extended Sequence Diagrams

Sequence diagrams are used for specifying timed procedural behavior in our method. This diagram type offers a rudimentary notion for time axes, and models can be extended by timing marks and temporal constraints. Several extensions, such as activation and deactivation delays, integrated time functions or composite activations on the lifeline of a single object have been proposed in [13].

Modeling of temporal behavior can be based on either time point relations or time interval relations. Note that all these point and interval relations can be expressed in sequence diagrams. In addition to explicit textual temporal constraints, we developed extensions of the graphical syntax for temporal constraints in order to make diagrams less complex and more comprehensible. and to reduce explicit temporal constraints. We do not show the syntax here, but restrict ourselves to the presentation of a generic temporal interval model that is depicted in Fig. 7 in comparison to the well established generic models of OCPN [11] and the interval relations from [14].

On sequence diagrams, we can distinguish different dimensions of time:

- (virtual, *local*) *object time* for each (application) object depicted on the diagram, i.e., the timing for actions relative to the object's notion of time,
- *message time*, i.e., the send and receive time of a message which correlate with events on the sender's and receiver's local time axes,
- (*global*) *real time* coinciding with the user's notion and perception of time.

Each temporal event and interval on a local time axis maps to a corresponding temporal event or interval of a reference time axis. A reference time axis can be the global time axis or another local time axis.

Figure 8 depicts the timed procedural behavior of a lecture discussion. Technically, it is given as the specification of the method start of class LectureDiscussion which is denoted by the header of the surrounding box. We have modified the behavior specification from [2] in several ways in order to explain additional features of our modeling approach. The atomic video segment of their example has been further decomposed into a trailer and the successive main video. The trailer is the same for all video segments. The main video consists of three parts: video, sound, and subtitles. In contrast to the trailer, where the decomposition is only on the media level (depicted by a parallely composed activation of trail:Trailer), the structure of the video was defined as a semantic property of the application by distinguishing three specialized ≪application≫ objects. The bold bar connecting the activation symbols instead of a message arrow specifies that the related activation intervals are to be presented synchronously. Which video is to be played is specified by parameter x. This parameter is also used in combination with role names on links–as depicted on the object diagram (see Fig. 2)–to define navigational path ex-

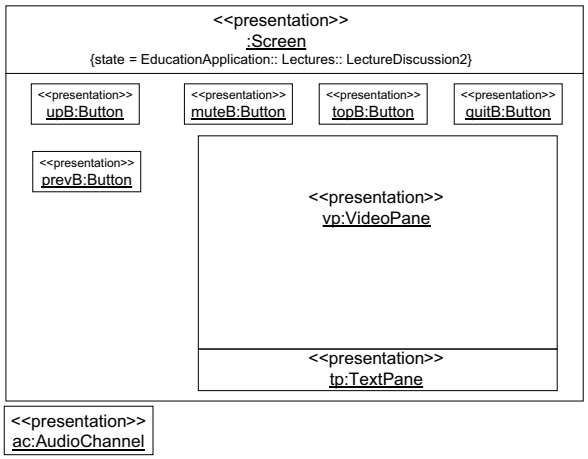


Figure 9. Presentation diagram showing presentation objects (and an audio channel) and their spatial layout for the video presentation

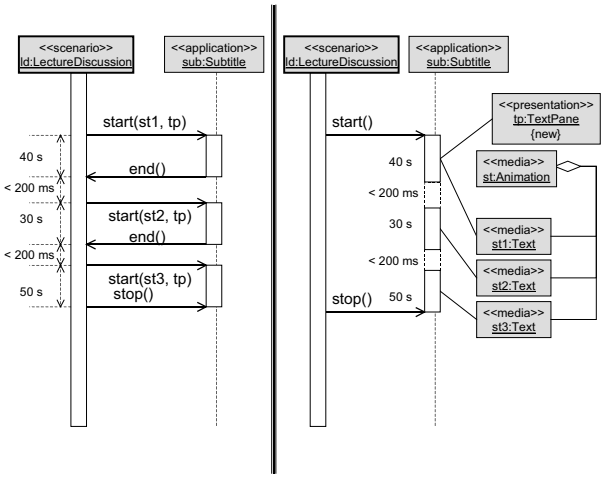


Figure 10. Alternative notations on sequence diagrams

pressions that evaluate, e.g., to the actual sound and subtitle objects (x.sound and x.subs, respectively). A temporal constraint has been added stating that the maximum delay between the end of the trailer presentation and the start of the video presention must not exceed 200 milliseconds. In contrast, the only temporal constraint defined in [2], saying that the start of the successive video must be after the end of the preceding one, was moved to the statechart in Fig. 6 where it is specified by the sequential activation of the parameterized *start* activity. We thus obtain a single sequence diagram specifying the behavior of all video presentations instead of specifying this behavior individually on the instance level.

The parameters of the *start* messages within the sequence diagram refer to the associated ≪media≫ and ≪presentation≫ objects. Since the trailer object has two different media objects associated with it, its *start* message has parameter lists with two-elements for each of both stereotypes. The ≪presentation≫ objects and their graphically depicted constraints (relative or absolute positioning) can be obtained from the presentation diagram assigned to the current scenario state (see Fig. 9; the corresponding class diagram is not shown).

While in the diagram of Fig. 8 only the synchronization bar and the parallel activation are non-standard notations, the right of Fig. 10 shows additional notations (here on an instance-level diagram) that we introduced for the modeling of multimedia applications. The respective standard syntax is given on the left. Instead of assigning media and presentation objects by parameterizing messages, they can be graphically attached to the corresponding activations on the application object's lifeline, like, e.g., the object tp:TextPane (in earlier work [13], we simply used textual references). Note that this object is marked with a constraint {new} stating that this presentation object is to be created within the scenario specified by the sequence diagram. Another simplification is the use of sequentially composed activations for animations. In this case, the intermediate *start* and *end* messages can be neglected. Objects can then be assigned to any segment of the animation activation as it were a normal activation. They will be overridden by the next occurrence of the same stereotype, i.e., in the example, object tp:TextPane is valid for the whole animation sequence, st1:Text only for the first segment. Delay intervals between segments can be distinguished by dotted borders. If temporal durations refer to intervals on an object's time axis, the construction marks of UML can be omitted.

The sequence diagram in Fig. 11 shows possibilities how to specify the synchronization of the partial view of Fig. 10 with the rest of the video presentation. It is modeled that the possible delays in the animation need to be resynchronized with the video and speech presentation as if they would not have happened. It is also exemplified how the local time of an object time axis can be related to global time by mapping intervals (or points in time) between two different time axes. Here it is modeled that 120 seconds of the presentation will be rendered within 40 seconds. Such mappings can be specified relatively, i.e., by an offset and a conversion factor, or absolutely. The same mechanisms can be applied between any two time axes. Finally, another temporal mapping is shown regarding the relation of inherent time of an associated continuous media object to the time axis of the application object. Here it is specified that the presentation of the media objects m:DigitalVideo and s:Audio

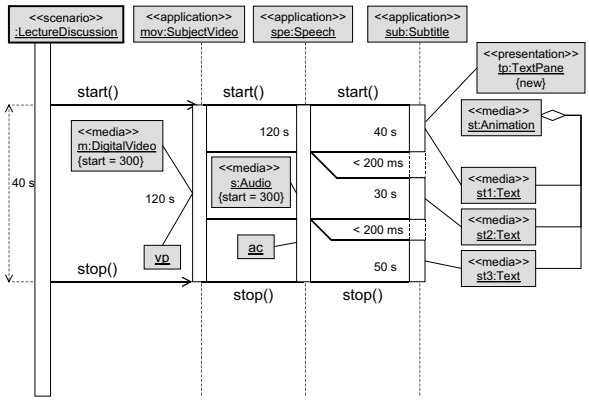


Figure 11. Synchronization and temporal mapping on sequence diagrams

starts at time 300. Since no duration or conversion factor is specified, these parameters are identical to those of the application object. In addition to temporal properties on assigned objects, spatial properties can also be modeled. By assigning changing spatial properties to associated presentation objects on an animation sequence, e.g., one can model spatio-temporal animations.

5 Integrating Timed Procedural and Interactive Behavior Modeling

After presenting the modeling of interactive behavior in statechart diagrams and the modeling of timed procedural behavior in sequence diagrams in detail, we will now show how these two different behavioral aspects are integrated in our modeling method for interactive multimedia applications. The main idea is that sequence diagrams specifiy actions appearing on transitions or within states (i.e., as internal actions or activities) of the statechart diagrams. Actions are atomic while activities can be interrupted. We have used the same mechanism already in Sect. 3 for the coupling of statemachines of different objects. Now the receiver's response to the action on the transition of the sender is not specified as a trigger on a transition of the receiver's statechart, but instead by a sequence diagram that specifies the initiated procedural behavior sequence. The statechart specifying the reactive behavior of class LectureDiscussion is shown in Fig. 6. As a reaction to the reception of a *perform* message, an object of this class enters the state *rendering*. The *do*-activity of this state specifies that the action *start* with the parameter current has to be executed. This action is specified by the sequence diagram of Fig. 8. Reactive and timed procedural behavior can thus be integrated. Figure 6 also shows that such timed procedural specifications can be assigned to complex states as well. In the example, it is possible to mute the sound during the video presentation without affecting the course of the timed procedural behavior. Only an asynchronous message is sent to the presentation object rendering the sound media object to set the output level to zero. By this mechanism, sequence diagram specifications can be integrated in any kind of dynamic specification and on any level of nesting within statechart diagrams. This removes the restriction to simple states as required in [13], allowing for interaction with running timed scenarios at execution time. The detailed semantic consequences of this integration are currently under investigation.

6 Related Work

A wide variety of models for the synchronization of temporal behavior within multimedia applications has been proposed which can be categorized into graph, Petri-net, object-oriented, and language-based models according to [4], but none of the analyzed models fulfills all requirements listed therein.

The timeline-tree model presented in [10] is similar to our approach in that it allows to execute different predefined timelines in response to user interaction. But due to the lack of modularization and hierarchical structuring concepts, models become overwhelmingly complex even for moderate size applications.

Adali et al. [2] present an algebra for interactive multimedia presentations. Within their model, a presentation consists of a tree whose nodes represent non-interactive presentations. An interaction corresponds to a transition from a parent to a child node traversing an edge of this tree. This resembles our concept of predefined scenes that are specified within sequence diagrams that are in turn assigned to actions (resp. states) of a statemachine. Transitions between theses states reify interactions with the multimedia application. Thus, states in the statemachine correspond to nodes in their multimedia presentation tree. Since statechart diagrams are in general graphs, our approach is more expressive as regards user interactive control. They also do not account for simultaneous presentation of continuous media.

In the field of visual programming languages, the visualization of temporal behavior has attracted some attention. In addition, Burnett et al. [6] investigate how spatial programming mechanisms can be applied to program temporal behavior for animated graphics. Animated graphics are a specific form of multimedia application. The main problems they have to deal with are the mismatch between the programming notation and the intuitive representation of the problem to be solved and the manipulation of speed for programming temporal interrelationships. The first objective (closeness of mapping) similarly applies to our modeling approach whereas the latter relates to the manipulation of presentation speed, i.e., the mapping of object time to real

time in our case. They develop a hierarchy of grid-based time models by relating the speed of a grid cell to normal time or the speed of another cell. The most elaborate model is capable of representing an unlimited number of speeds and temporal relationships. These models can be applied to the temporal mapping between different (local) object time axes or between object time and global time.

7 Conclusion

The presented integration of predefined temporal behavior and interactive control can be easily mapped to object-oriented implementation techniques and frameworks and the development paradigms of most multimedia authoring systems. According to the classification criteria in [4], we come to the conclusion that our integrated approach to behavior modeling of multimedia applications based on the Unified Modeling Language meets the following requirements:

- the diagrammatic notation is understandable even by non-technical members of a development team or users,
- due to the inherent structuring on the structural and behavioral levels by modularization and nested statemachines that are coupled with sequence diagrams, even large scenarios are still manageable,
- models are easily mapped to an object-oriented implementation or the concepts of many multimedia authoring tools, but still no programming language knowledge is needed for the task of modeling,
- temporal (as well as spatial) constraints can be intuitively expressed,
- although temporal and spatial constraints are notated in different diagrams, they are parts of a common integrated model,
- the object-oriented classification of different object stereotypes offers an adequate media abstraction and supports architectural decomposition,
- dedicated diagram types enable separation of concern for the different aspects of multimedia applications,
- language (and methods) are easily extensible and customizable by applying UML's built-in extension mechanisms for stereotyping, tagged-values, and constraints,
- the modeling process can thus be very flexible.

We gave an overview of the extensions to UML behavior diagrams for modeling interactive multimedia applications in this paper. The whole set of extensions together with a process defining the pragmatics for deploying this extended variant of UML is underway to be defined as a standard-compliant UML profile.

We are currently working on a formal semantics of UML behavior diagrams and the proposed extensions for the multimedia domain based on the concept of dynamic meta modeling as introduced in [7].

References

[1] P. Ackermann. *Developing Object-Oriented Multimedia Software – Based on MET++ Application Framework*. dpunkt, Heidelberg, 1996.

[2] S. Adali, M. L. Sapino, and V. S. Subrahmanian. An algebra for creating and querying multimedia presentations. *Multimedia Systems*, 8(3):212–230, 2000.

[3] T. Arndt. The evolving role of software engineering in the production of multimedia applications. In *Proc. IEEE Intl. Conf. on Multimedia Computing and Systems (ICMCS'99)*, pages 79–84.

[4] E. Bertino and E. Ferrari. Temporal synchronization models for multimedia data. *TKDE*, 10(4):612–631, 1998.

[5] E. Bertino, E. Ferrari, and Marco Stolf. MPGS: An interactive tool for the specification and generation of multimedia presentations. *TKDE*, 12(1):102–125, 2000.

[6] M. Burnett, N. Cao, and J. Atwood. Time in grid-oriented VPLS: Just another dimension? In *Proc. IEEE Symposium on Visual Languages (VL 2000)*, pages 137–144.

[7] G. Engels, J. H. Hausmann, R. Heckel, and St. Sauer. Dynamic meta modeling: A graphical approach to the operational semantics of behavioral diagrams in UML. In A. Evans, S. Kent, and B. Selic, editors, *Proc. UML 2000*, volume 1939 of *LNCS*, pages 323–337. Springer, 2000.

[8] S. J. Gibbs and D. C. Tsichritzis. *Multimedia Programming: Objects, Environments and Frameworks*. ACM Press, 1995.

[9] M. Hirakawa. Do software engineers like multimedia? In *Proc. IEEE Intl. Conf. on Multimedia Computing and Systems (ICMCS'99)*, pages 85–90.

[10] N. Hirzalla, B. Falchuk, and A. Karmouch. A temporal model for interactive multimedia scenarios. *IEEE MultiMedia*, 2(3):24–31, 1995.

[11] T. D. C. Little and A. Ghafoor. Synchronisation and storage models for multimedia objects. *IEEE Journal on Selected Areas in Communications*, 8(3):413–427, April 1990.

[12] Object Management Group. *OMG Unified Modeling Language Specification*. Version 1.3, June 1999. http://www.omg.org

[13] St. Sauer and G. Engels. Extending UML for modeling of multimedia applications. In M. Hirakawa and P. Mussio, editors, *Proc. IEEE Symposium on Visual Languages (VL'99)*, pages 80–87.

[14] T. Wahl and K. Rothermel. Representing time in multimedia systems. In *Proc. IEEE 1st Intl. Conf. on Multimedia Computing and Systems (ICMCS'94)*, pages 538–543.

Easy Model-Driven Development of Multimedia User Interfaces with GuiBuilder

Stefan Sauer and Gregor Engels

s-lab – Software Quality Lab
University of Paderborn
Warburger Str. 100
D-33098 Paderborn, Germany
{sauer,engels}@s-lab.upb.de

Abstract. GUI builder tools are widely used in practice to develop the user interface of software systems. Typically they are visual programming tools that support direct-manipulative assembling of the user interface components. We have developed the tool GuiBuilder which follows a model-driven approach to the development of graphical (multimedia) user interfaces. This allows a meta-design approach where user interface developers as well as prospective users of the system are supported in modelling the desired functionality of the GUI on a high level of abstraction that is easy to understand for all involved stakeholders. The model consists of compositional presentation diagrams to model the structure of the user interface and hierarchical statechart diagrams to model its behaviour. GuiBuilder then supports the transformation of the model to Java, i.e., the generation of a working user interface and the simulation of the modelled behaviour. Interactive sessions with the user interface can be recorded and replayed.

Keywords: Model-driven development, meta-design, user interface, prototype generation, capture-replay.

1 Introduction

Recently, *meta-design* has been proposed as a novel approach to system development where end users play an active role not only in using a software system but also in designing it. In [2], G. Fischer et al. state: "Meta-design characterizes objectives, techniques, and processes for creating new media and environments allowing 'owners of problems' (that is, end users) to act as designers. A fundamental objective of meta-design is to create socio-technical environments that empower users to engage actively in the continuous development of systems rather than being restricted to the use of existing systems."

In [1], M.F. Costabile et al. refine this approach and introduce the notion of "Software Shaping Workshops (SSW)", where groups of stakeholders focus on certain aspects of system development. They state: "We view meta-design as a technique, which provides the stakeholders in the design team with suitable languages

and tools to favour their personal and common reasoning about […].” Futhermore, they follow G. Fischer’s arguments, who characterizes end users as persons who want to be a “consumer” (i.e., user) of a software system in some situations, and in others a “designer”, who adapts the software system to her personal needs and desires.

In our approach, we exemplify these ideas by presenting a model-based development approach for graphical user interfaces (GUI). The overall idea is to provide high-level sophisticated design languages and tools, which allow end users to be involved in designing and testing graphical user interfaces of a software system.

Following the approach of model-driven development (MDD) techniques [4,9], such a platform-independent model of a GUI is automatically transformed into an executable GUI realisation in a common programming language like Java.

Graphical user interfaces of (multimedia) software applications provide users with the presentation of information and interaction capabilities with (media) content and functionality. The user interface is a complex part of the overall system and often requires software engineering effort comparable to building the application functionality itself. In addition, the user interface has to meet the user’s requirements and expectations in order to yield a high acceptance rate by future users. Thus, user interface development should be done cooperatively by software engineers and prospective end users. Due to the inherent complexity of user interfaces, model-based development processes which are nowadays well-accepted in software development should be applied for user interfaces, too. GUI builder tools that merely support visual programming of the user interface are overstrained with this task.

Model-based development of user interfaces promotes structuring of the resulting implementation and allows developers and prospective users in teamwork to prevent errors or to detect errors earlier and more easily by already analysing the model of the user interface. The models can also be used as documentation and for guiding the maintenance of the software system. *Model-driven* development even goes a step further by automatically generating from the model an executable user interface in a common programming language like Java.

The objective of this work is to develop a model-driven and tool-based development technique for graphical user interfaces (GUI). The model of the GUI combines structural and behavioural aspects. The model-driven development of the GUI is then supported by a tool called GuiBuilder. GuiBuilder provides developers and prospective users with an editor for GUI modelling and an execution environment for GUI simulation. A prototype user interface can be generated from the model, executed and tested. External tools can also connect to the simulation and are notified about the simulation progress. Simulation runs can be recorded and replayed. The simulation logs can also be used to support regression testing based on the capture-replay paradigm.

A number of model-based approaches have been proposed in past years to deal with user interface modelling at different levels of abstraction (see e.g. [10]). GuiBuilder is targeted towards concrete user interface modelling. The idea of combining statechart and presentation diagrams originally stems from the OMMMA approach [8]. Statecharts have also been used in [3] for describing GUI behaviour. UsiXML (e.g. [11]) uses graph transformations instead. It provides a variety of GUI elements which are currently not completely supported by GuiBuilder due to its early development state. In MOBI-D [7] the process of constructing a GUI is guided and

restricted by domain and task definitions, which are the building blocks of user interfaces in MOBI-D. A UML-based approach towards model-driven development of multimedia user interfaces is described in [5]. Recently, model-driven development of user interfaces has attracted wider interest in the research community [6].

In the next section, we will introduce the different models that are supported by GuiBuilder and their interplay. Section 3 presents the tool GuiBuilder. We draw conclusions and outline future perspectives in Section 4.

2 Models of GuiBuilder

The model of the multimedia user interface in GuiBuilder consists of two parts: the presentation model and the dynamics model. The *presentation model* captures the structure and layout of the user interface, the *dynamics model* uses UML statecharts to specify the behaviour of the GUI. Dynamic behaviour is enacted by user interaction or other events that cause a change of state in the user interface (and the application). Events that are caused by user interaction are modelled as signals which can be handled by the presentation elements. Signals can also be sent as the actions of triggered state transitions.

The basic concept of the compound model is to assign a presentation design to a state, which describes the structure and layout of the user interface while in that state. At any point in time, the GUI of an application is in a specific, possibly complex state. An event occurrence causes a state change and thus a change of the presentation.

The presentation model consists of presentation elements (see Fig. 1). Typically they are graphical elements that are part of the application's presentation. Such elements can e.g. be geometric shapes, widgets, or graphics elements for rendering images or video. In addition to graphical elements, audio elements can be included for playing music or sound effects. The presentation elements have properties which can be assigned with values. The properties depend on the type of presentation element and determine the presentation of the element. The types of presentation elements are organized in a class hierarchy.

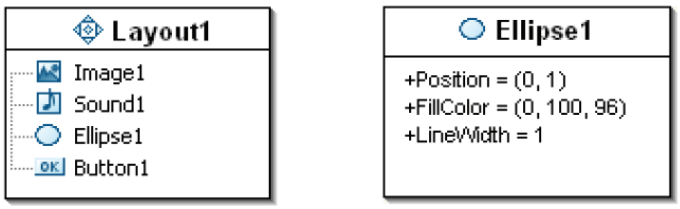


Fig. 1. A presentation consists of presentation elements (left), where each presentation element is characterized by its property values (right)

The presentation elements within one presentation diagram are ordered. The topmost element is upfront and possibly covers parts of other elements if they overlap.

If the GUI is in a simple state, the presentation is a composition of presentation elements with their property values. The presentation is completely described by the presentation diagram that is assigned to this state.

However, it is also possible to assign presentation diagrams to complex states in our model. Complex states allow us to hierarchically structure the state of the user interface. The actual presentation is then composed from the presentation diagrams that are assigned to the current simple state and all its parent states, where the complex states can even be concurrent (i.e., AND-superstates). Fig. 2 shows an example, where the presentation diagrams Layout1 and Layout2 are assigned to State1 and its substate State2, respectively.

The actual composition of the presentation is determined by the hierarchical structure of the statechart diagram. If the behaviour of a superstate is refined by substates, the assigned presentation is also refined by the presentation diagrams that are assigned to the respective substates.

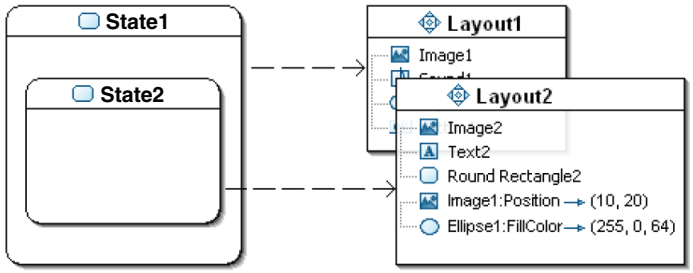


Fig. 2. Presentation diagrams can be assigned to hierarchical states, new presentation elements can be added for substates or properties of existing elements be modified

The composition of the presentation diagrams according to the state hierarchy works as follows:

First, presentation diagrams are stacked on top of each other. The order is determined by the state hierarchy: presentation diagrams of substates are put on top of presentation diagrams of their superstates. The former are intended to be the more specific. Their presentation elements override (cover) the presentation elements of the latter. For concurrent states, an order is not defined.

Secondly, since presentation diagrams can not only contain new presentation elements, but also property changes (i.e., modify the properties of presentation elements contained in presentation diagrams that are assigned to superstates), the modified value also overrides the 'inherited' value. All presentation elements that are introduced in presentation diagrams of the superstates of a state can be altered by modifying their property values. A property change thus specifies the modification of a property value of an inherited presentation element in a substate (see Fig. 3).

Consequently, a hierarchical presentation can be interpreted as a list of modifications where the instantiation of a presentation element is a specific case. This list can then be processed to construct and compose the actual presentation for a particular state: for each presentation diagram, the list of modifications is processed,

whereby the order of the lists of different presentation diagrams is determined by the hierarchical state structure from superstates to substates.

With respect to execution semantics, this means that when a state is left, the modifications of its presentation diagram to the user interface become ineffective and are replaced by the modifications of the presentation diagram of the successively entered state. Modifications of the presentation diagram of the possibly still active superstates remain unaffected, yet may be overridden.

Structured specification of a user interface is facilitated by this composition mechanism. GUIs typically contain a limited number of fundamentally different views which are then subject to a larger number of smaller (local) modifications for representing the particularities of different states within the overreaching context. Our incremental composition mechanism eases the specification of such modifications and prevents the developer from having to specify the complete presentation design for each, even simple modification. The GUI design thus requires less effort and the GUI models become easier to extend and modify even by end users, especially since redundancy is limited and controlled.

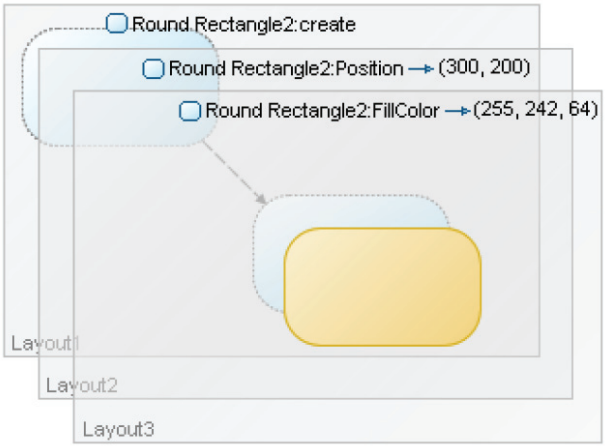


Fig. 3. Stepwise modification of a presentation element through hierarchical presentation diagrams

In addition to presentation, interaction also profits from the incremental specification. User interaction results in events which are received by presentation elements as signals. Since signals are properties of the presentation elements as well, they can be 'inherited' and modified like presentation properties. Functionality can thus be adapted in the same way by modifying the signal specification.

Thus, since we follow a clearly structured approach toward user interface construction and limit the GUI modelling language to a selected number of modelling concepts and elements, it is suited for professional software developers and end users as well. The integration of end users in the software development tasks is further promoted by the strict distinction of interactive control behaviour that is modelled here and possibly complex algorithmic computations of the system that are developed separately.

3 GuiBuilder - The Tool

GuiBuilder has been developed as a plug-in of the Eclipse tool environment and platform. We used the Plug-in Development Toolkit PDT for its implementation and the Graphical Editor Framework GEF for implementing the graphical editor of the GuiBuilder plug-in.

GuiBuilder supports user interface software developers as well as prospective users in the development of graphical (multimedia) user interfaces. Audio and video can be integrated in the presentation of the application that is developed. In the current version of GuiBuilder, executable GUIs are generated from the model and executed using Java SWT, and the Java Media API is deployed for rendering of multimedia artefacts.

The main view of GuiBuilder is the GUI editor. Additional views of GuiBuilder are the Eclipse standard views problems view, outline view, and properties view as well as the presentation view. The problems view lists the detected errors and warnings. The outline view presents an outline page for each window of the GUI editor when it is selected. The properties view shows properties of a selected model element (statechart element or presentation element). Properties can be edited directly in the properties view or in an explicit properties dialog. The presentation preview is a GuiBuilder-specific view that presents a preview of the presentation (see Fig. 6). In our development of GuiBuilder we tried to keep the tool as simple as possible—despite its diverse functionality—to be usable even by end users.

The tight integration of editing and simulation tools allows users to dynamically switch between the roles of software developers who design the structure and behaviour of the interactive graphical user interface by the use of design models and users who interact with the application that is being designed.

3.1 Editor

The editor of GuiBuilder is a graphical tool that supports the direct-manipulative construction of dynamics and presentation diagrams. The GUI editor is a multi-page editor that can manage windows of two different types for statecharts and presentation layouts, respectively.

3.2 Model Validation

The GUI editor calls the validator to validate the correctness of the edited model. The diagrams of the dynamics model have to be valid UML statechart diagrams where only a limited subset of modelling elements is used to control the model's complexity. In addition, we require that the specified behaviour is deterministic. Thus, the statechart diagrams are validated before code is generated from them by the generator function and the simulation can be started. To effectively support the developer as well as prospective user, we provide syntax-directed editing to prevent from fundamental syntactic errors and static model analysis to detect more complex and context-sensitive problems. For example, missing start events or non-deterministic transitions are identified by our model analysis. Two categories of problems, errors and warnings are recorded and presented to the user of the editor in the central

problems view, in the outline view (see Fig. 4), and directly at the relevant modelling elements in the editor view. The identified problems are accompanied by correction procedures (i.e., quick fixes). These features are altogether intended to support users of the editor as much as possible in detecting and correcting defects. Only after all errors have been resolved, the generation can be enacted. Warnings need not to be resolved; however, they should not be ignored since they mark weaknesses of concept or style within the model. Thus, the static analysis supports both the syntactic correctness of the model and its quality in accordance to modelling guidelines.

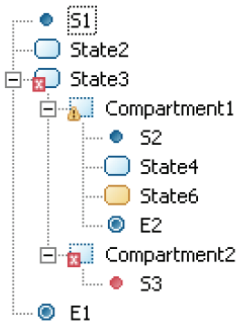


Fig. 4. Problems are marked at the causing model elements

Since the static analysis is the powerful core of the validation module, the syntax-directed editing restrictions can be kept low, not to unnecessarily hinder the flexibility of model editing. For example, inconsistencies or incorrectness can be temporarily tolerated as long as the developer does not want to start the prototype generation process.

Despite the wide range of checks in the static analysis, some problems can still only be detected during dynamic analysis. Dynamic analysis is integrated with the simulation and executed at runtime. Dynamic errors that are detected then are for example infinite loops or non-deterministic behaviour. Such errors cause the termination of the simulation run.

3.3 Generation and Simulation

The simulator view shows the simulated GUI. The GUI editor passes the GUI model via a generator function to the simulator view. The generator function flexibly implements the transformation rules to build a prototype GUI from the GUI model. It can be replaced for generating a different target language or for tailoring of the generation results.

The simulator view starts the simulation in the simulator and registers the GUI editor with the simulator. The simulator then notifies the GUI editor about state changes.

Simulation of the user interface is accomplished by interpreting the model. The GUI simulator uses a statechart simulator which interprets the statecharts of the dynamics model. Connected objects are notified by the statechart simulator about

state transitions and triggered actions (signals). The GUI simulator constructs the composite presentation view for the active state configuration and passes it to the simulator view of GuiBuilder. The simulator view renders the current GUI view. The user can then start the simulation in the simulation view, and the simulator executes the generated GUI. Events can be raised either by interacting with the simulated GUI elements directly or by using the 'remote control' that we implemented as an external plug-in. It can be operated remotely to generate the required signals.

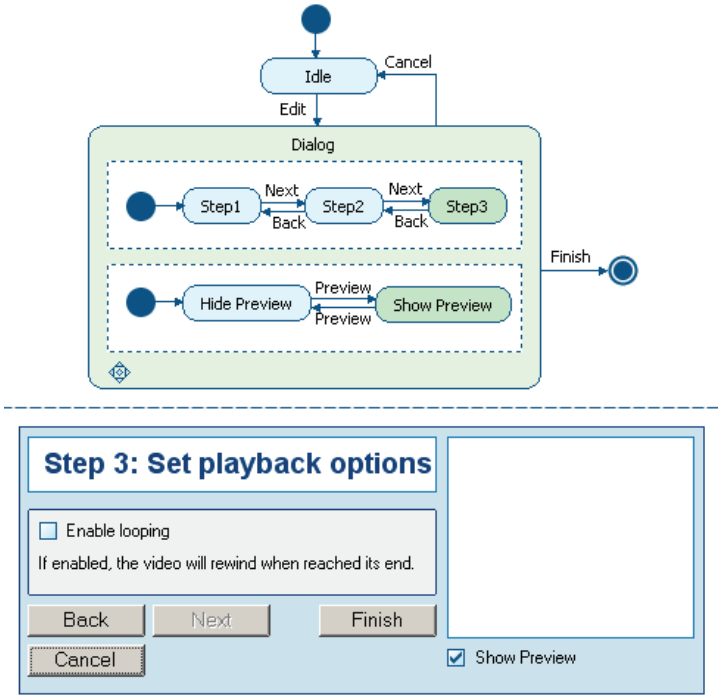


Fig. 5. Simulations can be tracked in the model

The interpretative approach has the advantage that the user interface model can be altered at runtime, and these changes can directly influence the succeeding simulation behaviour. GuiBuilder provides this functionality in a separate hot-code replacement mode.

External tools and other Eclipse plug-ins can connect to the simulator and are thus notified about state changes in the simulated model. They can assign specific actions to the signals and specifically respond to their occurrence. With this mechanism it is possible to actually control a fully fledged application. Besides, the simulation recorder that we developed uses this mechanism for recording a simulation run. The recorder logs the simulation execution. The recorded log can later be used to replay the simulation or to do regression testing after the GUI model has been modified.

External tools can themselves send signals to the simulation and raise events to change the state of the simulated GUI. Thus, the GUI can react to application or external events, too.

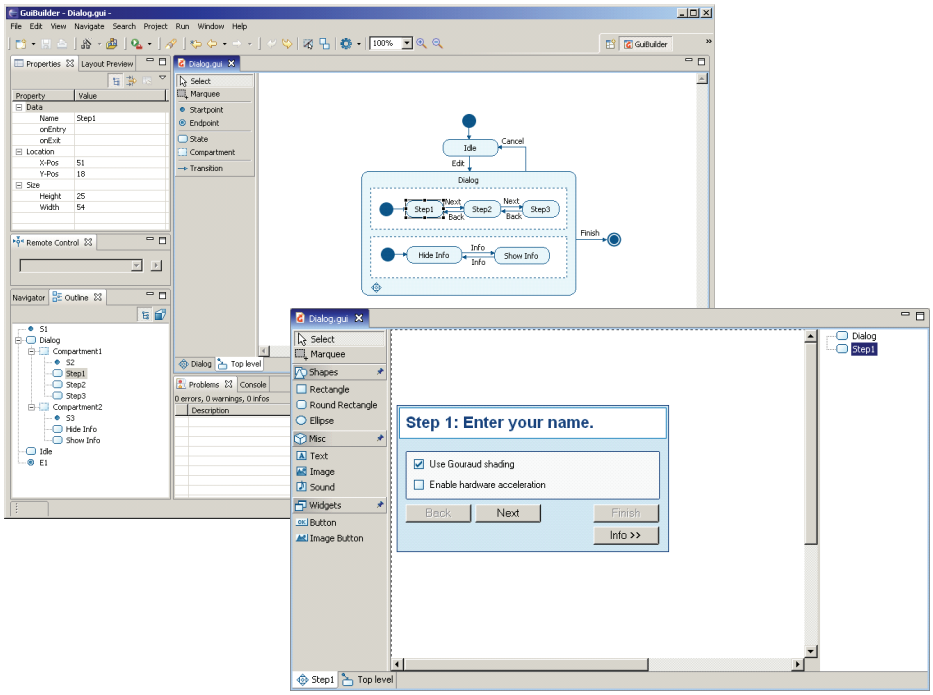


Fig. 6. The GUI of GuiBuilder

While a simulation is running, the editor view of GuiBuilder highlights the current state of the dynamics model in its statechart in green colour (composite state “Dialog” and its concurrent substates “Step 3“ and “Show Preview” in the example of Fig. 5). Thus, dynamic information is fed back into the model representation and can be used e.g. for model debugging.

4 Conclusions

We have integrated the model-driven development paradigm with the GUI-builder tool concept. This provides user interface developers as well as prospective end users with a tool for constructing graphical (multimedia) user interfaces in practice. The GUI model consists of presentation and dynamics models from which a prototype user interface can be generated and simulated.

In a next step, we plan to further improve the capabilities of multimedia processing by extending the dynamic model to deal with timed procedural behaviour. We also want to demonstrate the flexibility of the transformation approach by tailoring the generator function to different target representations.

We have evaluated GuiBuilder in several workshops with high-school students and people who are interested in software development, but not professional software developers or programmers. After a presentation of the tool of about half an hour they were capable of using the tool for constructing, changing and simulating simple applications like a traffic light control with only very limited support by our tutors. Thus, the tool has shown its capability to support end users with little programming skills in building and simulating interactive graphical user interfaces.

Additional information about GuiBuilder can be found at http://www.s-lab.upb.de/Tools/GuiBuilder/

Acknowledgments. The authors are indebted to the three computer science students Dennis Hannwacker, Marcus Dürksen, and Alexander Gebel, who contributed to the concepts of GuiBuilder and implemented the tool.

References

[1] Costabile, M.F., Fogli, D., Mussio, P., Piccinno, A.: A Meta-Design Approach to End-User Development. In: Proc. IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC'05), pp. 308–310. IEEE Comp. Soc, Washington (2005)

[2] Fischer, G., Giaccardi, E., Ye, Y., Sutcliffe, A.G., Mehandjiev, N.: Meta-Design: A Manifesto for End-User Development. CACM 47(9), 33–37 (2004)

[3] Horrocks, I.: Constructing the User Interface with Statecharts. Addison-Wesley, London (1999)

[4] Mellor, S.J., Scott, K., Uhl, A.: MDA Distilled: Principles of model-driven architecture. Addison-Wesley Professional, London (2004)

[5] Pleuß, A.: Modeling the User Interface of Multimedia Applications. In: Briand, L.C., Williams, C. (eds.) MoDELS 2005. LNCS, vol. 3713, pp. 676–690. Springer, Heidelberg (2005)

[6] Pleuß, A., Van den Bergh, J., Hußmann, H., Sauer, S (eds.).: MDDAUI '05. In: Proc. of the MoDELS'05 Workshop on Model Driven Development of Advanced User Interfaces, CEUR Workshop Proc. 159. CEUR-WS.org, (2005)

[7] Puerta, A.R.: A Model-Based Interface Development Environment. IEEE Software 14(4), 41–47 (1997)

[8] Sauer, S., Engels, G.: UML-based Behavior Specification of Interactive Multimedia Applications. In: Proc. IEEE Symposium on Human-Centric Computing Languages and Environments (HCC'01), pp. 248–255. IEEE Comp. Soc, Washington (2001)

[9] Selic, B.: The Pragmatics of Model-Driven Development. IEEE Software 20(5), 19–25 (2003)

[10] van Harmelen, M. (ed.): Object Modeling and User Interface Design. Addison-Wesley, London (2001)

[11] Vanderdonckt, J., Limbourg, Q., Michotte, B., Bouillon, L., Trevisan, D., Florins, M.: UsiXML: a User Interface Description Language for Specifying Multimodal User Interfaces. In: Proc. W3C Workshop on Multimodal Interaction WMI'2004 (2004), http://www.usixml.org

Applying Meta-Modeling for the Definition of Model-Driven Development Methods of Advanced User Interfaces

Stefan Sauer

Abstract. The user interfaces of interactive systems become increasingly complex due to new interaction paradigms, required adaptability, use of innovative technologies, multi-media and interaction modalities. Their development thus demands for sophisticated processes and methods, as they are deployed in software engineering. Model-driven development is a promising candidate for mastering the complex development task in a systematic, precise and appropriately formal way. Although diverse models of advanced user interfaces are deployed in a development process to specify, design and implement the user interface, it is not standardized which models to use, how to combine them, and how to proceed in the course of development. Rather, this has to be defined by methods in the context of organizations, domains, projects. To cope with the definition of model-driven development methods for advanced user interfaces, we propose a meta-method for method engineering. It can be used for modeling and tailoring such development methods. We show how to apply this meta-method for designing development methods in the domain of advanced user interfaces.

1 Introduction

The development of advanced interactive software systems demands for sophisticated engineering processes and methods, not only for the application functionality, but also for their increasingly sophisticated user interfaces. Model-driven development is a qualified approach for dealing with the complex development task of advanced user interface development in a systematic, precise and appropriately formal way. However, it needs to get along with the creative and less formal development techniques that are also used in user interface development.

Diverse models of advanced user interfaces are deployed in a development process to specify, design and implement the user interface. Among these models

Stefan Sauer
University of Paderborn, s-lab – Software Quality Lab
Warburger Straße 100, D-33098 Paderborn, Germany
e-mail: sauer@s-lab.upb.de

are task models, dialog structure or navigation structure models, dialog flow or navigation models, dialog state and presentation state models, abstract and concrete user interface models, models of adaptation, device capability models, and so on. The concrete set of models that is used for a development depends on the domain, purpose, and nature of the interactive system and its user interface. In an integrated development approach, the set of models also has to be compatible with other models of the interactive system such as those regarding application functionality.

In order to cope with this complexity, it is necessary to define precise methods for model-driven development of advanced user interfaces (MDDAUI). It must be specified which models and artifacts are to be produced, how they are related and how to proceed from one to the other by the use of transformations. Such transformations can be executed as manual development tasks or by automated procedures (e.g. model transformations) as part of the development process.

We propose a meta-method for method engineering [1] as a solution for this challenge. It can be utilized for modeling and tailoring engineering methods. We show how to apply this meta-method for designing development methods in the domain of advanced user interfaces.

The meta-method consists of a product and a process part. The product part prescribes which elements must be defined for a development method (*product model*). The *process model* specifies what needs to be done (work model) and how to proceed to obtain the definition of the development method (workflow model).

Engineering a development method then means instantiating the meta-method's product model according to its process model, i.e., the method engineer performs the defined method engineering tasks of the work model and follows the meta-method's workflow model. The resulting development method – the product of method engineering – is an instance of the meta-method's product model.

In our approach, the development method itself contains a *model of domain concepts* from the MDDAUI domain as its first product. Such domain concepts are general concepts from the domain of human-computer interaction such as user, task (not to be confused with the concept "task" from the method engineering domain), dialog, presentation state, widget and so on, but also concepts that are specific to either advanced user interfaces or model-based and model-driven methods. Examples are multimodal interaction and adaptation, or domain-relevant – both general and domain-specific – types of models with their model elements and defined model transformations, respectively. The model of domain concepts defines these concepts, their relevant properties and the interrelationships between the concepts. The domain concepts are paired with notations for their representation to form the *artifact types* (artifact model) of the development method. (Their semantic relations are taken from the model of domain concepts.) The pairing provides us with an adequate, integrated set of (modeling) languages for advanced user interfaces. We thus combine method engineering and language engineering in our meta-method.

The tasks in the process dimension are described as *transformations* that act upon the artifacts of the method's artifact model. The model transformations of model-driven development are a specialization of this transformation concept. The

activities of the workflow generally correspond to the tasks the UI developers have to accomplish, but may adapt them according to the situational context. It can be specified in a rule-based manner which effects a particular development task or activity has on the graph of artifacts. We use a notation based on graph transformation rules to describe the precondition, post-condition and effect of such development activities.

Tools can then be built that (1) use the model of domain concepts as the foundation of their artifact repository structure; (2) that use a representation that conforms to the defined notation of the method's artifact types in their interaction part, i.e., user interface, content and representation media, produced output documents and files; and (3) that use the work and workflow models as the basis for the supported functionality.

In the next section, we will analyze the method domain of MDDAUI in order to derive requirements for appropriate development methods from this analysis. These requirements transitively impose requirements on the meta-method, possibly requiring the specialization of the general meta-method for this class of methods. We structure our analysis according to the characteristics of user interfaces, advanced user interfaces, (advanced) user interface development, models of (advanced) user interfaces, and integrated model-driven development of advanced user interfaces. In Section 3, we give a general introduction into method engineering and the use of meta-modeling for method engineering. Our meta-method for method engineering is presented in Section 4. In Section 5, we show how to apply it for the MDDAUI method domain. We summarize our work in Section 6.

2 Model-Driven Development of Advanced User Interfaces

In this section, we give an overview of MDDAUI. We derive from this the requirements for model-driven development methods for advanced user interfaces. In particular, we look at both the characteristics of advanced user interfaces that impact their development from its product perspective and the inherent characteristics of the development approach from the process perspective.

2.1 User Interfaces

The user interface of interactive software systems is one of the key factors determining its success. Not surprisingly, the development of sophisticated user interfaces is gaining more and more attention not only in the human-computer interaction community, but eventually also in the software engineering community.

The *user interface* is the part of an interactive system where interaction between humans and computers occurs. Interaction is a bidirectional process of action and reaction, with the exchange of information between the human and the computer. User interfaces therefore provide means of input and/or output, thus allowing the users to manipulate a system and, vice versa, the system to indicate the effects of the users' manipulation. The user interface of a software-based system includes both hardware (physical) and software (logical) components. The term

"computer" thereby stands for an increasing multitude of computing platforms, ranging from smart cards and wearable computing devices, across interactive embedded systems, appliances, mobile phones and mobile computers, to desktops and collaboration environments (cf.[2]).

2.2 Advanced User Interfaces

Advanced user interfaces represent the current state-of-the-art in human-computer interaction. It is an intricate task to precisely define the term advanced user interface, since there exists a wide range of user interfaces that are considered advanced. Their common qualification is that they go beyond traditional user interfaces of data-intensive or simple control systems. But this can be with respect to different aspects, e.g. supporting complex interactions, visualizations, multimedia representations, multimodality, context-dependent adaptability, or customization (see [3], [4]). Summarizing and extending the classification of [5], typical facets of advanced user interfaces are:

- they have to provide a high degree of usability,
- increasingly complex functionality is expected,
- more intuitive interaction techniques are built in,
- multimodal interaction is supported,
- tailored and customizable representations of information are offered,
- techniques like animation or 3D visualization are incorporated,
- speech or haptic output are used as additional perception channels,
- temporal media types, like video and audio, and the combination of different modalities require dealing with synchronization and dependency issues,
- different interaction devices are used for different purposes, even within a single modality,
- they use a broad spectrum of presentation, perception, and representation media,
- context-aware user interfaces and adaptation to the context of use by means of context-sensitive and multi-target user interfaces and user interface plasticity appear in ubiquitous computing [2].

2.3 (Advanced) User Interface Development

User interface development generally employs both creative and informal techniques of development such as storyboards and prototyping, and formal techniques such as dialog structure and dialog state models. The development of user interfaces of a software-based system is a multidisciplinary task. It typically involves knowledge (and experts) from areas such as usability engineering, interaction design, graphics and media design, user interface technologies and interaction devices, computer engineering, software engineering, human factors, ergonomics and even psychology. User interface development comprises tasks of specification, design, and implementation. The implementation of user interfaces often

employs dedicated frameworks (e.g. Java AWT, SWT or SWING), toolkits, and tools (e.g. GUI builders).

Advanced user interface development covers a broader spectrum of aspects than traditional user interfaces development. This is due to two reasons: advanced user interfaces have additional aspects that need to be taken into account (product perspective); the development of advanced user interfaces comprises additional tasks, activities, methods and techniques that are not contained in traditional user interface development methods (process perspective).

We can thus distinguish between two different dimensions and interpretations of the term advanced user interface development, which can even be combined to build a third interpretation:

A) development of advanced user interfaces
B) advanced development of user interfaces
C) advanced development of advanced user interfaces

Model-driven development of user interfaces can be subsumed to category B, model-driven development of *advanced* user interfaces belongs to category C.

Advanced-user interface development naturally requires the combination of expertise from human-computer interaction and software engineering. One possible approach is to combine object modeling with user interface design [6]. A series of workshops on bridging the gaps between the software engineering and human-computer interaction communities was hold as an activity of IFIP WG 2.7/13.4 on User Interface Engineering during the last decade (http://www.se-hci.org/bridging/) and resulted in some interesting lines of research (see e.g. [7], [8], [9]) –MDDAUI being one of them!

It is our objective to integrate the knowledge from both domains and to apply the model-driven development paradigm to user interface development. We will look at this methodical integration from the perspective of models and modeling in the next section.

2.4 Models of (Advanced) User Interfaces

A *model* is, according to scientific theory, a representation of a natural or artificial original that focuses on those characteristics and properties of the original that are relevant for the given purpose of modeling, and abstracts from irrelevant properties. In an engineering process, models are used for specification, documentation, and communication. They are themselves objects of processing and transformation, and are a foundation for decision making, analysis, validation, verification, and testing. Models can be built upfront or retrospective in terms of forward engineering or reverse engineering, respectively.

The use of models has gained popularity in both software engineering and human-computer interaction over the years. Models have a long tradition in systems and software engineering. Eventually with the Unified Modeling Language (UML), model-based software development has become popular and common practice. Recently, model-driven development is attracting a lot of attention in the software engineering domain.

Likewise model-based user interface development has found its way into human-computer interaction design and user interface development. Models play an important role in today's user interface development. The purpose of models in the development of user interfaces has been stated in [4]:

> "Models shall act as a kind of bridge between input from various people involved in UI development (end users, domain experts, UI developers, management people, etc.) to integrate all this knowledge and to transfer it into the software engineering process."

However, although both communities make extensive use of models in their development methods, the modeling is still vastly independent.

As in software engineering, the modeling of user interfaces deals with different aspects and happens on different levels of abstraction. In addition, it may also be done with a different degree of detail. Therefore, a holistic model has to combine a set of partial models that are dedicated to modeling specific aspects on a defined level of abstraction. The required degree of detail should be part of an accompanying quality model.

For example, the CAMELEON reference framework for user interface models in [2] structures the development lifecycle in four *levels of abstraction*: tasks and concepts, abstract user interface, concrete user interface, final user interface.

Human-computer interaction and the development of advanced user interfaces naturally address a broad spectrum of *aspects* to be considered. They can be represented by a diversity of dedicated models. Different kinds of models have been widely used in the development of user interfaces. For example, task and dialog models are used in many developments, and traditional approaches for user interface development provide abstract and platform-independent models for basic widget-based user interfaces.

For example, the CAMELEON reference framework [2] proposes a set of models for the modeling of context-sensitive user interfaces. On the conceptual level, three groups of models are differentiated: domain, context, and adaptation. Domain concepts and tasks belong to the domain models. User, platform, and environment models are subsumed in the context models. Adaptation models comprise evolution and transition models. From these models, design models are derived. Among them are concepts and task models, abstract user interface and concrete user interface models and the final user interface model for a given configuration. A third group of models guide the adaptation process of the context-sensitive user interface at runtime. Obviously, there exist relationships between these models that call for systematic transformation.

In our method for object-oriented modeling of multimedia applications OMMMA [10], we use four different types of models in combination: presentation model (structure and layout), state model (interactive control), class model (media and application structure), and sequence model (temporal behavior). Our GuiBuilder method [11] uses a concrete user interface model consisting of a presentation model (structure and layout of user interface elements) and a dynamics model (interaction behavior). The GuiBuilder tool provides an editor and components for model validation, UI prototype generation and simulation.

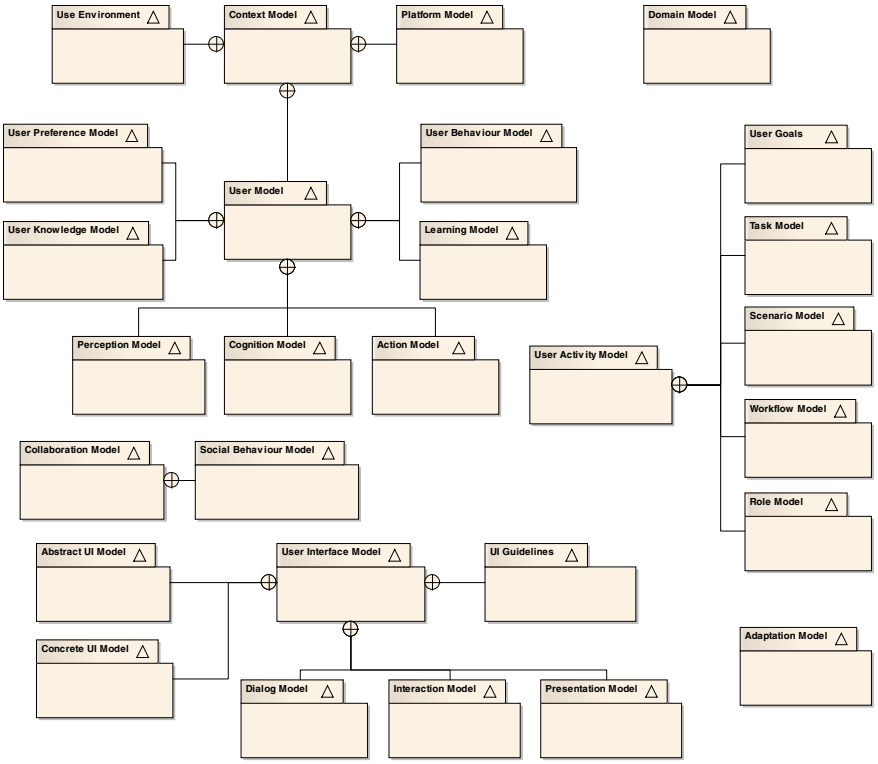


Fig. 1 A large variety of models is used in the development of (advanced) user interfaces

In [12], we have given a list of models and sub-models that are commonly used in user interface development. This list of models does not claim to be complete, but already shows the diversity of models being used. A partly extended set of models is shown in Fig. 1. Some of them may be even further decomposed, e.g. the dialog model into dialog structure, dialog flow, and dialog state models; or the presentation model into presentation structure, presentation layout, and presentation state models.

Which models are actually needed and best suited depends much on the given development task and context. In [5], we concluded that "it is very probable that there is no single set of models optimal for every kind of user interface".

However, not all of the aspects listed in Sections 2.1 and 2.2 can be easily represented by *formal* models in a user interface development method. Therefore, user interface development traditionally employs a number of informal techniques (see Section 2.3) to cover certain aspects, especially if development is performed on a higher level of abstraction. Their results can be considered to be *informal* models. Furthermore, development methods have not only to consider the system perspective, but also look from the perspective of the users.

Advanced user-orientation can be achieved by integrating methods of software engineering with (less formal) methods of user-centered design [8]. Therefore, the analysis and conceptual modeling of users, contexts of use, tasks and usage scenarios have to be covered by an advanced user interface development method as well.

Hence, it is our objective to provide a methodological framework that allows method engineers to define, flexibly select and customize (semi-)formal models and to integrate them with other artifacts to cover all relevant aspects of their advanced user interface development in a coherent set of models and artifacts. The resulting methods combine modeling with informal techniques of other design disciplines such as interaction design, creative design, graphics design, media design, to name but a few. We call the result of such a method development an *integrated method*.

2.5 Integrated Model-Driven Development of Advanced User Interfaces

The guiding principle of MDDAUI is "the demand for a flexible composition of various different models to support the model-driven development of user interfaces with a high degree of usability and customization" [13].

Model Driven Development (MDD) is an important paradigm in software engineering. The basic idea is to systematically specify software using (platform-independent) models, which are then gradually (i.e., using platform-specific models) and (semi-)automatically transformed into executable applications for different platforms and target devices.

MDD employs another core concept in addition to models: *model transformations*. Model transformations can be used to transform the content of a model or between models. Models can be (semantically) transformed or (syntactically) translated. Model transformations can also be used to check and restore consistency or other quality properties of models. The intention of MDDAUI is to apply this software engineering paradigm in the domain of user-interface development.

2.6 Requirements for Integrated MDD Methods for Advanced User Interfaces

From the aforementioned analysis, we can summarize important requirements for MDDAUI methods and, transitively, the method engineering meta-method. We classify the requirements by their origin: the product domain of (advanced) user interfaces (UI, AUI), the method domain of (advanced) user interface development (UID, AUID), and the development paradigms model-based development (MBD) and model-driven development (MDD). The requirements are listed in Table 1. They will be answered by the meta-method in Section 4.

Table 1 Requirements for model-driven development methods for advanced user interfaces that need to be covered by the meta-method

Requirement	Type
The method must be able to treat a user interface as part of an interactive system.	UI
The method must support typical user interface concepts such as user, goal, user interface, dialog, presentation, physical and logical user interface component, platform, device.	UI
The method must support typical concepts of advanced user interfaces and address the relevant aspects for the abstract user interface from the list in Section 2.2.	AUI
The method must allow user interface developers to use multiple methods and techniques that differ in formality and scope, such as creative techniques and formal techniques.	UID
The method must support a combination of different domains of knowledge in a multi-disciplinary development.	UID
The method must support multiple stages of development.	UID
The method must be able to account for implementation practices and techniques by specifying the use of technologies such as frameworks, toolkits, and tools.	UID
The method must account for usability and user needs.	UID
The method must support informal development techniques for user interfaces.	UID
The method must support multiple views.	UID
The method must produce a set of artifacts that are related to each other.	UID
The method must be able to distinguish different stages of development.	UID
The method must provide an integrated artifact model that combines formal and informal representations.	UID
The method must support different disciplines of user interface development.	UID
The method shall be integrated with software engineering practice.	UID
The method must provide restricted views for different developer roles.	UID
The method must include the necessary methods for developing the relevant aspects of advanced user interfaces.	AUID
The method must allow for the combination of methods from software engineering and human-computer interaction.	AUID
The method must be capable of using models in the development.	MBD
The method must support multiple models.	MBD
The method must support models for different purposes, such as specification, documentation, and communication (as indicated in Sect. 2.4).	MBD
The method must support the use of models for capturing development knowledge about the advanced user interface.	MBD
The method must support modeling on different levels of abstraction.	MBD
The method must support the selection of a set of different models that model different aspects.	MBD
The method must allow for the differentiation of degrees of detail in models.	MBD
The method must include the definition of elements of models.	MBD

Table 1 (*continued*)

Requirement	Type
The method must include the definition of relationships between models.	MBD
The method must support model-driven development, i.e., the specification of model transformations within and among models, operating on models and model elements.	MDD
It must be supported by the meta-method to specify model transformations.	MDD
The model shall provide notions of platform-independent and platform-specific models.	MDD

3 Method Engineering for Advanced User Interfaces

In this section, we will introduce the discipline of method engineering and will then discuss the use of meta-modeling for method engineering. This will lead us to the definition of our meta-method in the next section.

3.1 Method Engineering

Method engineering has been an active research area in the field of information systems engineering since the early 1990s. In general, method engineering is concerned with formalizing the use of methods for systems development [14]. More precisely, method engineering can be defined as the *engineering discipline to design, construct and adapt methods, techniques and tools for the development of (information) systems* (based on [15], [14]). The objective of method engineering is to develop a methodological approach for systems development in a given context (and situation) such as an organization or project.

Method engineering mainly addresses two perspectives: a) the systematic development of methods and b) the enactment and execution of methods. Both aspects may themselves be supported by dedicated tools, such as a method development environment and a method workflow engine.

Applying method engineering to the domain of advanced user interface development provides a number of advantages:

- method engineering provides a methodological framework and conceptual infrastructure for method knowledge,
- method engineering supports a systematic development of model-driven development methods for advanced user interfaces,
- by providing specific means for method adaptation, methods can be adapted to a particular situation and context of use *(*cf. *situational method engineering*, see ([14]) for a recent survey),
- concepts of method modularization, reuse and configuration can be used to assemble methods from method building blocks, such as *viewpoint templates* [16], *method fragments* [15], *method chunks* or *method services* [17],

- the meta-models that are used for the definition of methods enable analysis and comparison of methods, even quantitatively by the use of metrics,
- method engineering can ease reuse and provide means for compositional method development, and method integration,
- method engineering builds a sound basis for tool support, e.g. computer-aided software engineering (CASE) tools that may be built by using Meta-CASE tools.

The product of a method engineering process is a method. The users of this product are system and software engineers, and user interface developers in the case of MDDAUI.

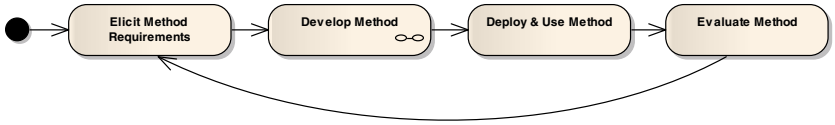


Fig. 2 The general overall method engineering lifecycle is similar to a software lifecycle

The lifecycle of a method is similar to the lifecycle of a software system. We can interpret a method as a conceptual system for system development. Method engineering manages and controls this method lifecycle and may even itself be computer-supported by its own software system, a computer-aided method engineering (CAME) tool [15]. The general overall process model of method engineering is depicted in Fig. 2. Once the domain of discourse has been identified (MDDAUI in our case), the requirements for the method are analyzed. It follows a multi-stage development process. Then the method is deployed, used, and evaluated in order to start another evolution cycle.

3.2 Meta-Modeling for Method Engineering

Meta-modeling has been identified as a promising means for method engineering. Several meta-models have been defined in the literature by different authors, see e.g.[18], [19], [20], [21]. Two standards also exist that use meta-models for the definition of software development methods: ISO 24744:2007 Software Engineering – Metamodel for Development Methodologies [22] and SPEM, the Object Management Group's (OMG) Software & Systems Process Engineering Meta-Model Specification [23]. The latter provides a meta-model as well as a UML profile for the specification of software development methods.

MOF, the OMG's Meta-Object Facility [24], has defined a four-layer meta-model architecture that is commonly used in object-oriented meta-modeling. In this hierarchy, elements of layer n-1 are instances of elements in layer n ($1 \leq n \leq 3$). According to this meta-model hierarchy, we can characterize the levels for the domain of method engineering:

M0 (Runtime layer) – M0 denotes the lowest level of the MOF 4-layer meta-model hierarchy. In this layer, objects of the real world are denoted that exist at execution time of the modeled system. More generally, M0 represents the area of concern, which may be business, software engineering, or method engineering. In the domain of method engineering, the M0 elements are the concrete objects that are produced or modified during a concrete development endeavor.

M1 (Model layer) – M1 is the layer where user models are located. Reality is modeled in a modeling language, such that elements of M0 are instances of elements in M1. In the domain of method engineering, the model of the method is allocated on this level.

M2 (Meta-model layer) – M2 is the layer where meta-modeling takes place. It contains meta-models (models of models) such as the UML meta-model or SPEM which define modeling languages to describe the user models of layer M1. Elements of user models from M1 are then instances of meta-model elements of layer M2. This level holds the meta-method's product model in the domain of method engineering.

M3 (Meta-meta-model layer) – M3 is the highest level of the 4-layer meta-model hierarchy. Meta-meta-models are defined on this layer. They are used to describe the meta-models on layer M2. In the MOF hierarchy, the Meta Object Facility itself is defined on this level. Defining method engineering within an object-oriented meta-model hierarchy, we use MOF for the domain of method engineering on this level as well.

We also build on meta-modeling in our meta-method for method engineering. However, we have discovered that simply employing object-oriented meta-modeling has some shortcomings. In particular, the restriction to solely have MOF's <<instanceOf>> relationship between meta-model layers, and to permit it only between directly neighboring layers, does not allow us to straightforwardly combine the product and the process parts within this framework. Yet for defining a method, we have to combine the method's product model with its process model, as depicted in Fig. 3.

The process model is composed of a work model and a workflow model. We apply this method pattern on both the meta-method level and the method level. However, while the meta-method process model must be an instance of a process meta-model to have execution semantics, all parts of the method are defined as an instance of the meta-method product model, since the complete method is the product of the method engineering process. Yet, the method process model must also be an instance of the process meta-model, since it is a process model itself. We solve this problem by bootstrapping the process meta-model into the meta-method product model with a <<merge>> relationship (see [24]), like this was done for MOF and UML, too. The method is engineered by instantiating the meta-method process model and enacting the thus instantiated process on the method level. This relation is represented by the dependency of type <<producedBy>> between the method and the instance of the meta-method process model. The same pattern applies on the M0 level for the production of the development project's artifacts. Further details on the formal background of our meta-modeling approach for method engineering can be found in [1].

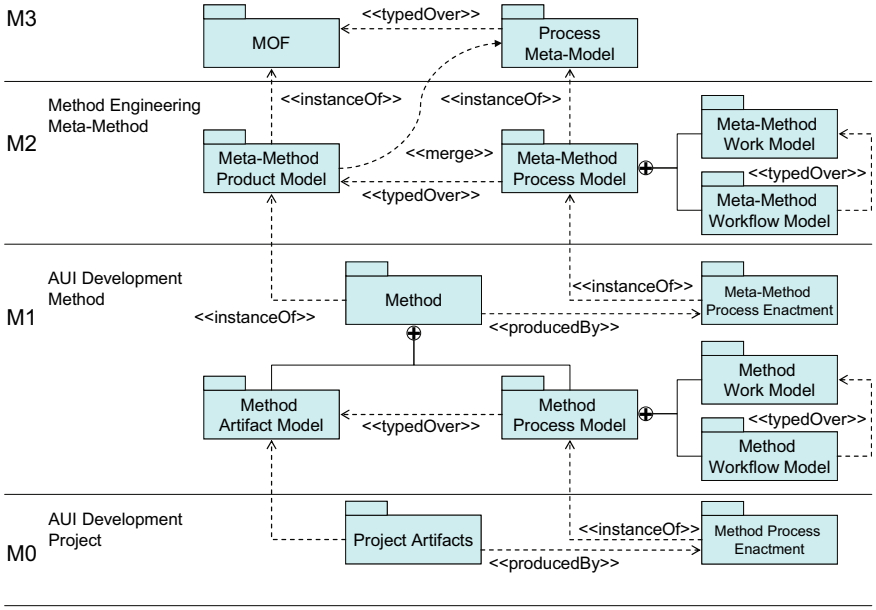


Fig. 3 Applying the meta-modeling approach for the engineering of methods

4 Meta-method for Engineering Development Methods

Conforming to the model presented in the previous section, the meta-method of our method engineering approach consists of a product and a process model. We will give an overview of both in this section. For the process part, we will focus on the workflow model.

4.1 Process Model of the Meta-Method

In Fig. 4, the workflow of the composite activity "develop method" from Fig. 2 is shown. While we describe the process workflow in a rather waterfall-like structure for the ease of presentation here, it may in practice be enacted in a more incremental and iterative fashion.

The meta-method's process combines activities of language engineering and method engineering. A first version of the process was published in [25]. There, we focused on the development of the domain model and artifact model together with language selection (steps 2 to 4 in the process depicted in Fig. 4). In [1], we provide a complete and revised description of the step 1-4, 6 and 7 of the above process in the context of the general method. However, in this work we have specialized and extended the general process for the domain of MDDAUI. We describe the specialized process in the following step by step.

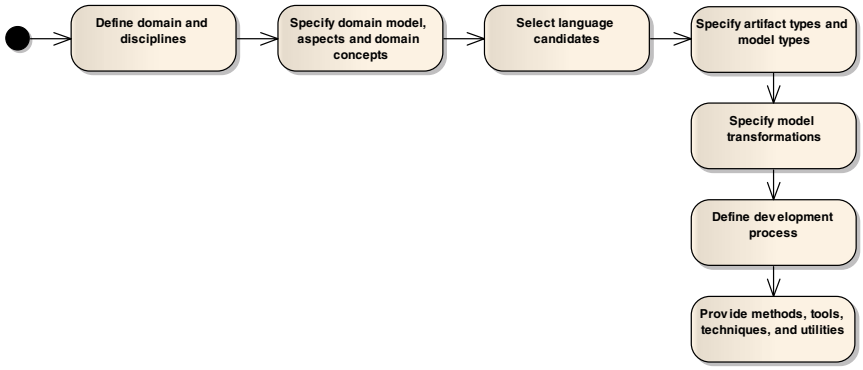


Fig. 4 High-level process model of the meta-method for model-driven development methods

1. **Define domain and disciplines:** The domain is MDDAUI in our case, and disciplines are used to further structure the development method into areas of concern, such as requirements elicitation, conceptual modeling, interaction design, abstract user interface modeling, concrete user interface modeling, user interface implementation, and so on.
2. **Specify domain model, aspects and domain concepts:** The model of domain concepts is set up and organized according to the identified disciplines (in the form of packages that may be hierarchically nested). The disciplines may also correspond to stages of development or levels of abstraction. From the requirements in Sect. 2.6, we have also derived the need for views that represent the perspective of a stakeholder or a particular aspect of the advanced user interface. Core tasks of this activity are the definition of domain concepts and assigning them to disciplines and views.

   Relationships between concepts are added such as composition and aggregation relationships, dependencies, associations.

   The meta-model representation is accompanied by a glossary that contains an entry for each meta-model class. It describes the semantics, purpose and properties of the concept and relationships to other concepts.
3. **Select language candidates:** In order to represent the domain concepts appropriately, languages, together with possible sub-languages (e.g., UML diagram types) and language elements must be identified as candidates.
4. **Specify artifact types and model types:** Candidate languages and language elements from step 3 are assigned to domain concepts from step 2 according to the properties of the domain concepts that need to be expressed. While the model of domain concepts defines the semantics of the method elements in the product model, languages define the syntax and notation for their representation. The artifact model then links language elements with domain concepts. If existing languages or symbols are used, then the method engineer has to take care that the given semantics of the proposed candidate language elements is conformant with the semantics imposed by the composition of step 3, and the

semantics of each language element shall still be unambiguous. Composition hierarchies in the model of domain concepts and the artifacts model must be compatible.

This step of the process is further extended in the domain of model-based development methods. In addition to general artifact types, models can be defined as specializations of the artifact concept. The required and allowed model elements are defined for each type of model, and relationships between models can be defined in the same way as for artifacts.

5. **Specify model transformations:** Since we address model-driven development methods for advanced user interfaces in our approach, this step is an extension to the standard method engineering process of the meta-method. If a model-driven development method is to be defined, the model transformations must be defined as transformation within or between models. This can be done in a rule-based manner.
6. **Define development process:** The definition of the development process reifies the definition of a roadmap through the network of development artifacts. Activities are defined and ordered into workflows that produce the required artifacts in the specified order.

   We have to define tasks, activities for accomplishing tasks, steps of activities and workflows containing an ordered set of activities in this step of the meta-method's process. The process structure contains activities, milestones and control-flow elements. The process model can be extended by object flows of input and output artifact types, and roles that are responsible for executing activities.
7. **Provide methods, tools, techniques, and utilities:** The selection or development and the provision of methods (method modules), tools, techniques, and utilities as well as the provision of tool mentors are required for guiding and simplifying the works of user interface development and producing the required artifacts. Tools are assigned to artifact types, languages or development techniques. Guidance on how to produce the artifacts of a particular type in the selected language shall be explicitly provided, e.g. in the form of guidelines, good and best practices, whitepapers, checklists, templates, examples, or roadmaps. However, even the assignment of languages to software engineering concepts in step 3 can be interpreted as partly associating a technique for the development artifact. Both languages and tools typically have implications on how to produce an artifact. Eventually, tools and utilities are thus related to the activities of the software engineering process model as well. By this, it is shown which activities are supported by tools and utilities and, in turn, which of them are to be used when accomplishing the task of the activity.

4.2 Product Model of the Meta-Method

The product meta-model for method engineering that we propose for model-driven development methods is depicted in Fig. 5. According to the meta-model, the domain is structured into disciplines. Artifacts are related to the disciplines, where they are used. An artifact is always related to a pair of concept and notation. All relevant concepts of the user-interface development domain are elements of

the domain model. Furthermore, aspect views are defined on the domain model to cover particular views on selected aspects of the domain model, e.g. a modeling view such as for task modeling or a view for a given developer role or stakeholder (then possible relations to the respective classes `Model` and `Role` are not modeled as associations of this meta-model).

Models are an important concept in model-centered, i.e., model-based and model-driven, development paradigms. To account for that, we introduced the class `Model` as a specialization of the class `Artifact` in our meta-method's artifact model. Models contain model elements, as indicated by the composition relationship between the classes `Model` and `Model Element`. This allows method engineers to define model types and their element types directly as they commonly define artifact types in their methods.

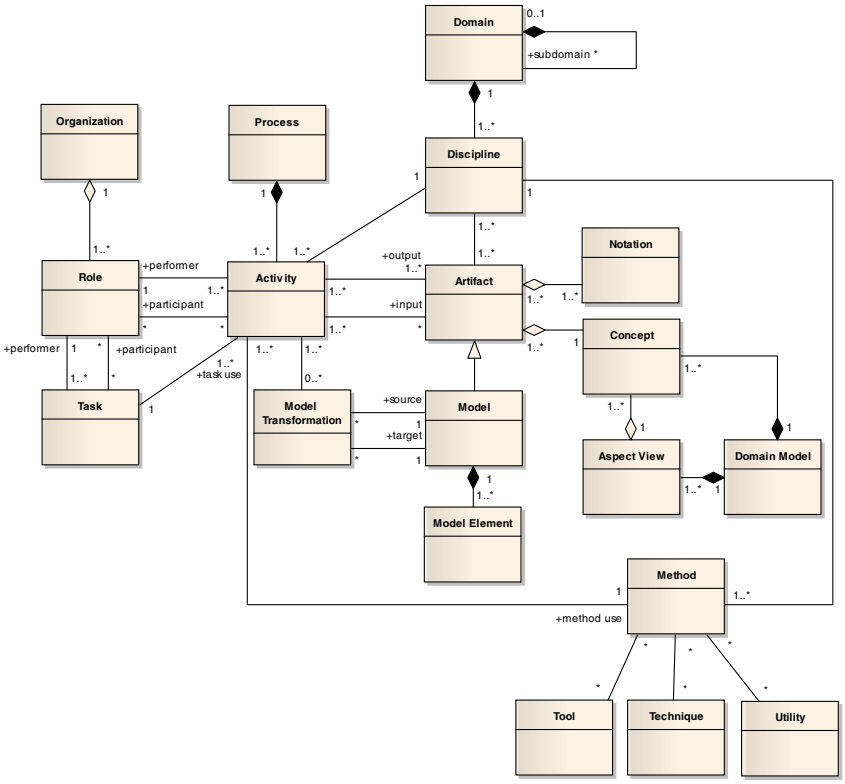


Fig. 5 Meta-model in the context of the model-driven development paradigm.

In the model-driven development paradigm, *model transformations* play a prominent role. They should therefore be considered as first-class citizens of any model-driven development method. Thus, for model-driven development methods, we have included the class `Model Transformation` which has source and

target relations to class `Model`. A transformation rule then operates on the model elements (not modeled in this meta-model) of the related models. Activities are the binding element between the information and the process view of the method description. Activities are owned by disciplines and are the constituents of workflow processes. They operate on artifacts which they use as their input and output parameter objects. Each activity uses a defined method to produce its output. Alike the activity, the method is also associated with a discipline. Such a method can provide tools, techniques and utilities that support the performers in accomplishing the task that is related to the activity. Each model transformation is related to one or more activities, meaning that the transformation is executed as part of the activities in order to transform the elements of the related models as specified by the transformation.

5 Applying the Meta-Method: Development of Model-Driven Development Methods for Advanced User Interfaces

After we have seen the product model and the general workflow model of the meta-method in the previous section, we will now look at some consequences when this approach for method engineering is applied in the MDDAUI domain. We will concentrate on some important aspects of such a method definition.

The first major result of the method engineering process that is released to user interface developers as the users of the method is a structured model of artifact and model types, together with their relationships. This is typically represented as a model of packages, sub-models (see Fig. 1 in Section 2.4) and classes. Such model can become quite large, therefore it is important to employ the described means of structuring. An excerpt from such a model of an advanced user interface is depicted in Fig. 6. It shows five types of models and three classes representing model elements.

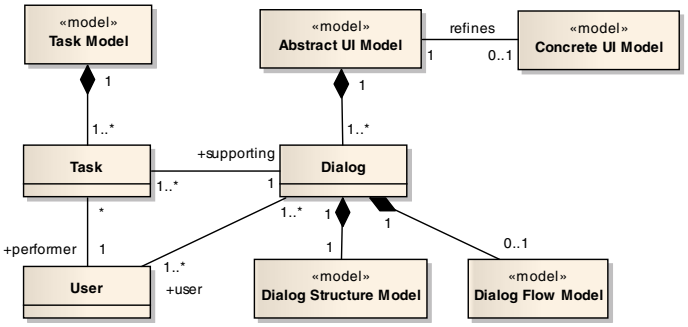


Fig. 6 Excerpt from the artifact model of a development method for user interfaces that is used as the type definition of a transformation rule

The definition of workflows does methodically not differ from the definition of workflows for the meta-method as shown in the previous section. We will therefore omit to present another example here. However, the use of transformations for the specification of development tasks and their effect on the product model was not shown there. The same approach can also be deployed for the specification of model transformations in model-driven development methods.

Effects of development tasks as well as model transformations on the models of the user interface can only be expressed in a limited way by using activity diagrams or composite structures [1]. Even if object flows are represented, they can only make reference to the state of individual objects. They are insufficient for modeling the effect of a task or transformation on the object structure, i.e., the graph of objects that are connected by association links, of the modeled system. We therefore included collaborations in our methodical framework that are interpreted as graph transformation rules [26]. These transformation rules are typed over the product model of the method.

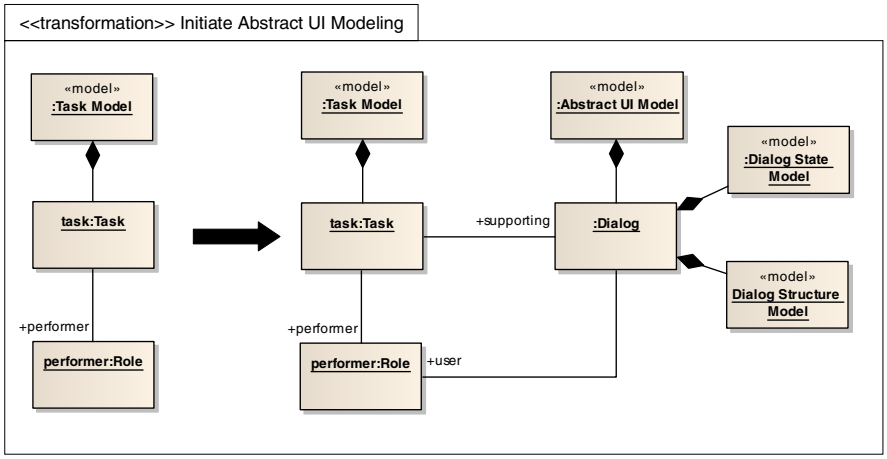


Fig. 7 Example for a model transformation rule defined on the instances of the meta-model

Fig. 7 gives an example of such a transformation rule. It states for the transformation "initiate abstract UI modeling" that for each occurrence of the pattern on the left-hand side in an instance of the product model, the structure on the right-hand side must be produced by the transformation. In particular, it states that if a task model contains a task that is performed by the instance performer of class `Role`, then a dialog must be generated as part of the abstract user interface model which supports the given task and is used by the `performer:Role` to accomplish the task. Furthermore, two more models have to be instantiated: a dialog state model and a dialog structure model, that are both associated to the generated dialog element. The rule can be interpreted as a visual contract stating pre- and post-conditions of the transformation [27].

6 Conclusions

We presented a meta-method for the development of systems, software and user interface development methods in this chapter. It builds on the concept of object-oriented meta-modeling based on the 4-layer MOF architecture, yet extends it to account not only for the product model, but also for the work definitions and workflows that form the process model.

We applied the concepts of method engineering in general and our meta-method in particular to the domain of model-driven development of advanced user interfaces (MDDAUI). Based on the analysis of requirements for such a development method stemming from both the product domain of (advanced) user interfaces and the method domain of integrated model-driven development, we adapted the general method engineering meta-method to cover models and model transformations as first-class citizens of the method description. Finally, we briefly showed some results of applying the meta-method to the target domain, especially graph transformation rules for the specification of tasks, activities and transformations in a user interface development process.

References

[1] Engels, G., Sauer, S.: A Meta-Method for Defining Software Engineering Methods. In: Engels, G., Lewerentz, C., Schäfer, W., Schürr, A., Westfechtel, B. (eds.) Graph Transformations and Model-Driven Engineering. LNCS, vol. 5765, pp. 411–440. Springer, Heidelberg (2010)

[2] Calvary, G., Coutaz, J., Thevenin, D., Limbourg, Q., Bouillon, L., Vanderdonckt, J.: A unifying reference framework for multi-target user interfaces. Interact with Comput. 15(3), 289–308 (2003)

[3] Pleuß, A., Van den Bergh, J., Sauer, S., Hußmann, H., Bödcher, A.: Model driven development of advanced user interfaces (MDDAUI) – MDDAUI'06 workshop report. In: Auletta, V. (ed.) MoDELS 2006. LNCS, vol. 4364, pp. 101–105. Springer, Heidelberg (2007)

[4] Pleuß, A., Van den Bergh, J., Sauer, S., Görlich, D., Hußmann, H.: Third international workshop on model driven development of advanced user Interfaces. In: Giese, H. (ed.) MODELS 2008. LNCS, vol. 5002, pp. 59–64. Springer, Heidelberg (2008)

[5] Pleuß, A., Van den Bergh, J., Sauer, S., Hußmann, H.: Workshop report: model driven development of advanced user interfaces (MDDAUI). In: Bruel, J.-M. (ed.) MoDELS 2005. LNCS, vol. 3844, pp. 182–190. Springer, Heidelberg (2006)

[6] Van Harmelen, M. (ed.): Object modeling and user interface design: designing interactive systems. Addison-Wesley, Longman (2001)

[7] Kazman, R., Bass, L.: Guest editors editorial: special issue on bridging the process and practice gaps between software engineering and human-computer interaction. Softw. Process Improv. Pract. 8, 63–65 (2003)

[8] Engels, G., Sauer, S., Neu, B.: Integrating software engineering and user-centred design for multimedia software developments. In: Proc. 2003 IEEE Symp. Human Centric Computing Languages and Environments (HCC 2003), pp. 254–256. IEEE Computer Society, Los Alamitos (2003)

[9] Seffah, A., Vanderdonckt, J., Desmarais, M.C.: Human-centered software engineering: software engineering models. In: Patterns and Architectures for HCI, Springer, London (2009)
[10] Engels, G., Sauer, S.: Object-oriented modeling of multimedia applications. In: Chang, S.K. (ed.) Handbook of Software Engineering and Knowledge Engineering, vol. 2, pp. 21–53. World Scientific, Singapore (2002)
[11] Sauer, S., Dürksen, M., Gebel, A., Hannwacker, D.: GuiBuilder – A tool for model-driven development of multimedia user interfaces. In: Van den Bergh, J., et al. (eds.) Model Driven Development of Advanced User Interfaces, MDDAUI 2006. CEUR-WS, vol. 214 (2006), http://CEUR-WS.org/Vol-214/
[12] Van den Bergh, J., Meixner, G., Sauer, S.: MDDAUI 2010 workshop report. In: Van den Bergh, J., et al. (eds.) Proc. 5th Intl. Workshop on Model Driven Development of Advanced User Interfaces MDDAUI 2010. CEUR-WS, vol. 617 (2010) urn:nbn:de:0074-617-8
[13] Meixner, G., Görlich, D., Breiner, K., Hußmann, H., Pleuß, A., Sauer, S., Van den Bergh, J.: Fourth international workshop on model driven development of advanced user interfaces. In: Proc. 13th Intl. Conf. Intelligent User Interfaces (IUI 2009), pp. 503–504. ACM, New York (2009)
[14] Henderson-Sellers, B., Ralyté, J.: Situational method engineering: state-of-the-art review. J. Univers. Comput. Sci. 16(3), 424–478 (2010)
[15] Brinkkemper, S.: Method engineering: engineering of information systems development methods and tools. Inf. Softw. Technol. 38, 275–280 (1996)
[16] Nuseibeh, B., Finkelstein, A., Kramer, J.: Method engineering for multi-perspective software development. Inf. Softw. Technol. 38, 267–274 (1994)
[17] Rolland, C.: Method engineering: towards methods as services. Softw. Process. Improv. Pract. 14, 143–164 (2009)
[18] Jeusfeld, A., Jarke, M., Mylopoulos, J. (eds.): Metamodeling for method engineering. MIT Press, Cambridge (2009)
[19] Bollain, M., Garbajosa, J.: A metamodel for defining development methodologies. In: Filipe, J., et al. (eds.) ICSOFT/ENASE 2007. CCIS, vol. 22, pp. 414–425. Springer, Heidelberg (2008)
[20] Gonzalez-Perez, C., McBride, T., Henderson-Sellers, B.: A metamodel for assessable software development methodologies. Soft. Qual. J. 13, 195–214 (2005)
[21] Henderson-Sellers, B., Gonzalez-Perez, C.: A comparison of four process metamodels and the creation of a new generic standard. Inf. Softw. Technol. 47, 49–65 (2005)
[22] ISO, ISO/IEC 24774:2007 Software engineering – metamodel for development methodologies. International Organization for Standardization, Geneva (2007)
[23] OMG, Software & systems process engineering meta-model specification, version 2.0. Object Management Group (2008), http://www.omg.org/specs/
[24] OMG, Meta object facility (MOF) core specification, version 2.0. Object Management Group (2006), http://www.omg.org/spec/MOF/2.0/PDF/
[25] Engels, G., Sauer, S., Soltenborn, C.: Unternehmensweit verstehen – unternehmensweit entwickeln: von der Modellierungssprache zur Softwareentwicklungsmethode. Inform. Spektrum 31(5), 451–459 (2008)
[26] Heckel, R., Sauer, S.: Strengthening UML collaboration diagrams by state transformations. In: Hussmann, H. (ed.) FASE 2001. LNCS, vol. 2029, pp. 109–123. Springer, Heidelberg (2001)
[27] Lohmann, M., Sauer, S., Engels, G.: Executable visual contracts. In: 2005 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC 2005), pp. 63–70. IEEE Computer Society, Los Alamitos (2005)