SITUATIONAL METHOD ENGINEERING FOR THE ENACTMENT OF METHOD-CENTRIC DOMAIN-SPECIFIC LANGUAGES

DISSERTATION
for the degree of
Doctor of Natural Sciences (Dr. rer. nat.)

Benjamin Honke

University of Augsburg

Department of Computer Science
Software Methodologies For Distributed Systems

April 2013

Situational Method Engineering for the Enactment of Method-Centric Domain-Specific Languages

Supervisor: **Prof. Dr. Bernhard Bauer**, Department of Computer Science
University of Augsburg, Germany

Advisor: **Prof. Dr. Wolfgang Reif**, Department of Computer Science
University of Augsburg, Germany

Date of defense: July 15th, 2013

Copyright: ©Benjamin Honke, Augsburg, April 2013

Abstract

Model-driven technologies influence today's software engineering more and more. During the last decade, a multitude of so-called meta models (or modeling languages) were developed to lift the level of abstraction from textual programming languages to the more conceptual level of models, which can be processed (e.g., by using model transformation, model validation, or simulation), and from which code can be generated automatically. Beside many benefits, which arose using this new technology, other drawbacks came up. Especially, the multitude of domain-specific and complex meta models, the relationship between different meta models, and unfamiliar design principles, still hamper effective application of model-driven technologies. This situation, for example, becomes obvious in automotive industry, where meta models, such as AUTOSAR, EAST-ADL, or TADL, were developed to enable the effective design of different concerns in automotive software development. Although, a lot of documentation in textual form is available, there is a gap between informal documentation and formal meta-model specification. Especially, matching complex meta models with the overwhelming number of documentation to apply meta models correctly proves sometimes difficult and causes a gap between research and the application of new technologies in real industrial development projects.

Therefore, the basic idea behind this thesis is to provide an integrated process model, which enables the execution and enactment of model-driven development processes in form of an effective guidance system. On the one hand, this requires the extension of conventional control-flow semantics of today's software development processes, which are modeled for management and simple documentation purposes only. On the other hand, our approach integrates more detailed information about guidelines, roles, or work products, which is normally distributed across different informal documents. We put all these information into a comprehensive and computer-interpretable model, which is afterwards interpreted to guide developers' work automatically. Based on this, the process model not only allows us to determine a sequence of actions, which have to be executed to produce particular output, but it also enables us to generate new artifacts, such as activity-specific editors to support respective development activities. This works as follows:
While generated editors can exactly provide developers with needed capabilities, using an all-purpose standard editor allows for the application of all capabilities at any time of the development process. Because of the increasing number of large meta models or domain-specific languages, this could be a confusing task for most of the time. Therefore, our approach proposes the generation editors, that restrict the capabilities and the available set of design elements, i.e., language elements of a meta model, to the situational needs of a respective development activity only.
In addition, the process model links each development activity with an individual set of computer-interpretable guidelines, which are relevant for the activity only, in contrast to the global set of "constraints" or guidelines as used in standard editors. These guidelines are evaluated in the context of a specific development activity and provide developers

with situational guidance information.

As the process model is interpreted and monitored all the time, we are enabled to log all modeling actions. These logs are either used to improve the process afterwards in doing conventional reviews, or to analyze these logs to derive traceability information at process execution time, and to identify potential inconsistencies between work products. This information is used to decide on the repetition of individual development activities to correct the inconsistencies.

Finally, as it proves difficult to design complex process models for each project situation from scratch, we combined the general approach with process line engineering techniques to enable reuse of already available information and to combine these information with additional computer-interpretable design artifacts. Therefore, a so-called method repository is used to store all interpretable development activities as variants for variation points designed in a reference process, which can be compared with the reference product (aka. platform) as used in product line engineering. The final process is generated by matching project-specific requirements with available variants and combining these variants with relevant variation points of the reference process.

Acknowledgments

First of all, I would like to thank my supervisor Prof. Dr. Bernhard Bauer for his excellent mentoring. His guidance, support, friendship, and motivation were the basis for the successful completion of this thesis. He did not only gave me the opportunity for conducting a Ph.D., but also taught me about organizing things, performing research and communicating with project partners.

I want to thank Prof. Dr. Wolfgang Reif, who accepted to be advisor of my thesis.

Special thanks go to all my colleagues of the Software Methodologies for Distributed Systems lab, in which I found a friendly and cheerful atmosphere to work.

In addition, I would like to thank all project colleagues, partners and other people, that contributed to this thesis in working together and by discussing joint research ideas. Especially, I want to thank Stefan Voget and Stefan Kuntz, which provided me with valuable insights in industrial practice and their long-term experiences.

Especially, I gratefully thank my brother Florian and my parents Barbara and Wolfgang for their education and support during my studies. They have always encouraged me to do my best in all matters of life.

Contents

1 Introduction **13**

1.1 Problems and Challenges . . . 15
1.2 Objectives, Approach and Contributions . . . 17
1.3 Outline . . . 20

2 Foundations **23**

2.1 Automotive Modeling Initiatives . . . 24
2.1.1 AUTOSAR . . . 24
2.1.2 MAENAD . . . 30
2.1.3 TIMMO-2-USE . . . 33
2.1.4 Conclusions from above Initiatives . . . 34
2.2 Model-Driven Engineering and Semantic Technologies . . . 35
2.2.1 Meta-modeling Technical Space . . . 35
2.2.2 Ontological Technical Space . . . 39
2.2.3 Guidelines, Best Practices, and Validation . . . 39
2.3 Business Processes Management . . . 41
2.3.1 Business Process Management Architectures . . . 44
2.3.2 Business Process Modeling . . . 46
2.4 Methodology Engineering . . . 50
2.4.1 Terminology of Methods and Processes . . . 50
2.4.2 (Situational) Method Engineering . . . 54
2.4.3 Tool Support for Methodology Engineering . . . 62
2.5 Software Product Line Engineering . . . 64
2.5.1 Variability Engineering . . . 66
2.5.2 SPLE Development Process . . . 69
2.5.3 Tool Support for Software Product Line Engineering . . . 72

3 Software Process Line Engineering **73**

3.1 Motivation . . . 74
3.1.1 Structure-driven Dimension . . . 74
3.1.2 Behavior-driven Dimension . . . 74
3.1.3 Resource-driven Dimension . . . 75
3.2 Overview: Software Process Line Engineering . . . 76
3.3 Process Family Engineering . . . 78
3.3.1 Process Family Definition . . . 79
3.3.2 Variant Design . . . 85

3.3.3 Configuration Criteria Definition 87
3.4 Feature Model Generation 89
3.5 Situational Process Engineering 93
3.5.1 Feature Model Configuration 94
3.5.2 Generating planning domain 95
3.5.3 Final Process Derivation 96
3.5.4 Resource-oriented Process Analysis 100
3.5.5 Deployment 104
3.6 Case Study 104

4 Computational Method Engineering 107
4.1 Motivation 108
4.1.1 Process-centric Requirements 108
4.1.2 Method-centric Requirements 109
4.2 Overview: Computational Method Engineering 109
4.3 Technical Process Design 111
4.3.1 Requirements for Development Processes on Technical Level . . . 111
4.3.2 Technical Process Modeling: Core Concepts 112
4.3.3 Process Modeling Requirements 114
4.4 Artifact Design 115
4.4.1 Meta Models & Views 117
4.4.2 Process Model - Meta Model Relationship 132
4.4.3 Artifact Content Propagation 134
4.5 Method-specific Editor Design 138
4.5.1 Editor-specific Meta Model View Re-arrangement 139
4.5.2 Element Usage Scenario Definition 141
4.6 Guideline Design 144
4.6.1 Guideline Characterization 145
4.6.2 Requirements for Situational Method-centric Guideline Design . . 149
4.6.3 The Generic Guideline Meta Model 151
4.7 Role-centric Workflow Management 166
4.7.1 User Skills 167
4.7.2 Data Access 167
4.8 Case Study 167
4.8.1 Artifact Design 170
4.8.2 Editor Design 173
4.8.3 Guideline Design 173

5 Method-driven Guidance of Development Processes 179
5.1 Motivation 179
5.1.1 Task-centric Challenges 180
5.1.2 Workflow-centric Challenges 180
5.2 Overview: Method-driven Guidance of Development Processes 181
5.3 Method-driven Editor Generation 181
5.3.1 Generation of Structural Editor Code 183

5.3.2 Generation of Behavioral Editor Code . . . 184
5.3.3 Challenges and their Solutions . . . 186
5.4 Artifact-specific Information Management . . . 187
5.4.1 Artifact Observer Mechanism . . . 188
5.4.2 Artifact Element Assignment Strategies . . . 191
5.5 Situational Model Validation - Guideline Application . . . 195
5.5.1 Guideline Realization . . . 196
5.5.2 Guideline Application . . . 206
5.5.3 Guideline Effects . . . 208
5.6 Engineering Process Coordination . . . 209
5.6.1 Overview: Flexible Workflow Management . . . 211
5.6.2 Consistency Check: Setup Phase . . . 214
5.6.3 Consistency Check: Monitoring Phase . . . 216
5.6.4 Consistency Check: Evaluation Phase . . . 216
5.6.5 Consistency Check: Control Phase . . . 219
5.7 Case Study . . . 219
5.7.1 Guidance Preparation . . . 220
5.7.2 Guidenace Application . . . 226

6 Evaluation 231

6.1 Motivation . . . 231
6.2 Architecture Analysis . . . 232
6.2.1 Conceptual Architecture . . . 232
6.2.2 Prototypical Implementation . . . 233
6.2.3 Evaluation with HASARD . . . 235
6.3 Scenario-based Evaluation . . . 239
6.3.1 Scenario-based Modifiability Evaluation . . . 239
6.3.2 Scenario-based Reusability Evaluation . . . 241
6.3.3 Scenario-based Maintainability Evaluation . . . 242
6.4 Dynamic Analysis . . . 244
6.4.1 Performance Evaluation of Software Process Line Engineering . . . 244
6.4.2 Runtime Complexity of the Application of Computational Method Engineering . . . 246
6.5 Descriptive Evaluation . . . 247
6.5.1 Process Improvement Standards and Maturity Levels . . . 247
6.5.2 Process Improvement with Situational Method Engineering for Process-Centric Languages . . . 249
6.5.3 Checklist for Organizations . . . 251
6.6 Case Study . . . 253
6.6.1 Setting up the process line for M3 and Situational Process Engineering 253
6.6.2 Variant Design for M3 . . . 256
6.6.3 M3 Enactment . . . 259

7 Related Work 261

7.1 Related Work on Method Engineering . . . 262

7.2 Related Work on Computer-aided Method Engineering Environments . . 265
7.2.1 Overview about Computer-aided Method Engineering constituent Parts . . . 266
7.2.2 Approaches for Process-Centered Software Engineering Environments . . . 266
7.2.3 Approaches for Computer-aided Method Engineering Environments 269
7.3 Discussion and Comparison of the Approach with Related Work 272
7.3.1 Comparison of Method Engineering Approaches 272
7.3.2 Comparison of Process-Centered Software Engineering Environments Approaches . . . 273
7.3.3 Comparison of comprehensive Computer-aided Method Engineering Approaches . . . 277

8 Conclusions and Outlook **279**
8.1 Summary of Thesis . . . 279
8.2 Future Research . . . 281

Bibliography **283**

List of Figures **322**

List of Listings **325**

Appendices **326**

A Acronyms **326**

B Translational Statement Semantics **331**

C HAZARD Analysis **353**

D CMMI enabled Process Areas **355**

E Case Study **357**

F Curriculum Vitae **368**

1 Introduction

In the past, organizations of the embedded sector, in particular, in the automotive domain, discovered the benefits of realizing more and more functionality in software instead of hardware or mechanics. Due to lightweight, more favorable, and easy repeatable solutions, software becomes more and more important, and provides new opportunities for innovative products. For example, within only 30 years, the number of software in modern premium cars increased from 0 to more than 10.000.000 lines of code, which are distributed across up to 70 Electronic Control Units (ECUs) to provide more than 2000 individual functions exchanging up to 2500 signals between different bus systems [Bro06, Kes09]. That way, 90% of all innovations are driven by electronics and software and 50-70% of the development costs of the software/hardware systems are software costs in today's vehicles [Bro06, HMG11].

This rapid change from hardware and mechanic development to software related development activities strongly influences software engineering and causes new challenges. The increasing number of control units hosting a multitude of distributed software functionality increases the complexity and efforts of system development [NSSLW05]. In particular, cost, flexibility, extensibility, and integration efforts are changing the fundamental paradigms for the definition of large embedded architectures [FK06,LT09]. Therefore, for efficient development of embedded systems, well defined processes, powerful tools, and techniques must be provided. In order to meet these and other challenges, conventional software development is more and more replaced by model based (or model driven) approaches [HMG11].

The model based paradigm is not new for the software engineering area to conceptualize information in the form of models, which are used as primary artifacts to raise the level of abstraction at which developers create and evolve software [GSC+04]. An evolution of this paradigm, is Model Driven Engineering (MDE) [B05], whereof approaches, such as Model Driven Software Development (MDSD) [SVC06] or Object Management Group (OMG)'s Model Driven Architecture (MDA) [OMG03], are representatives from. That model driven paradigm provides techniques to automate the propagation of domain information in the form of models between different abstraction levels, i.e., from a high user-specific level down to the platform-specific, operational level. As a result, the complexity of the software artifacts can be drastically reduced by separating concerns and aspects of a system under development [HT06].

Therefore, the advantages of creating and processing models to face the new challenges, which arise from growing complexity, were discovered by the embedded sector and led to a range of initiatives to establish models. In the automotive sector, for example, various projects, such as AUTomotive Open System ARchitecture (AUTOSAR), TIMing MOdel - TOols (TIMMO), Model-based Analysis & Engineering of Novel Architectures for Dependable Electric Vehicles (MAENAD), or Safe Automotive soFtware architEcture (SAFE), aim at providing meta models, i.e., modeling languages, tools, and other techniques, by which different concerns of system and software development can be handled more efficiently. Additionally, to apply the new techniques in a correct manner, a multitude of reference processes and documentation is another essential outcome of these projects.
However, the powerfulness of developed modeling languages and the new ways of thinking, hamper developers to keep track of the application of these new technologies. Indeed, many reference processes and associated documentation are defined to provide guidance, but the mostly informal explanations are often too complex and must be enabled by additional means.

Therefore, following Osterweil's statement, that "software processes are software too" [Ost87], this thesis introduces a new approach, which aims at an effective application of computing power and model-driven technology to support the execution of development processes. Although, the approach is applicable to different domains, this thesis focuses on the model-driven development of embedded systems, and, in particular, of automotive systems. This is due to a strong cooperation with our industrial partner from automotive, that supports us in understanding and identifying challenges, which are relevant in practice.

Therefore, the focus of this thesis is on how to provide developers with guidance support in their daily work, which, in particular, means, that we face challenges, such as

- Provision of effective means to manage and to evolve process descriptions for the situation at hand.
- Support of developers in designing sound process descriptions, which enable effective guidance.
- Support of the enactment and execution of sound process descriptions using computer power.
- Provision of a continuous tool chain and environment.

The remainder of this chapter is organized as follows: In Section 1.1, we introduce identified problems and challenges, which we will face in this thesis. The main contributions of this thesis are summarized in Section 1.2. Finally, we detail the structure of the overall thesis in Section 1.3.

1.1 Problems and Challenges

This section describes concrete problems of using new paradigms in today's embedded systems development, which were initially identified in [Hon08] and discussed intensively with our industrial partner. Based on this, we derive the different challenges, which we face, when aiming at the guidance of developers of embedded systems using innovative technologies.

Situational and Project-dependent Development Processes

Although, many reference processes are available for different application scenarios in various domains, they can not be applied one to one to actual projects. Projects depend on situational characteristics, such as customers, employees, or the product to be developed. Therefore, processes must be customized, i.e., tailored to meet actual circumstances. Furthermore, situational processes consist of various building blocks, which represent information sources containing more or less information about how to conduct particular tasks of the development. As many processes for even more situations are conducted, as more building blocks become available and must be managed for conducting and optimizing future projects.

Problems

Support for managed variability and situational configuration of process knowledge is a critical precondition to enable flexible guidance based on a multitude of information.

- Out of the box processes are not applicable to match project-specific requirements and process tailoring is a time-consuming and error-prone task, which requires high expertise.
- Although, processes are defined to provide a general workflow of what has to be realized, they do not consider support of how to conduct a particular task.
- Processes are generally perceived from an abstract business-oriented point of view and do not consider the technical or operational level, which supports the development, likewise.

Challenge 1

Provide a lightweight, easy to use framework, which enables the situational design of process descriptions, which considers automation, reuse, adaptation, and evolution of existing and upcoming process information.

Design of Sound Process Descriptions Enabling Automated Guidance

Normally, processes are only used as "paperware" documents, which are required to be available for certification purposes or organizational reasons. Although, modern process management systems are available to organize process knowledge, developers are used to realize their tasks based on their skills and experiences. However, skills and

experiences of individuals, is only one part of effective development. A lot of information, which would enhance the quality and efficiency of software development, results from many parties, such as colleagues, inter- and intra-organizational research projects, or academia. This knowledge, though, is mostly distributed across various information sources and can not be used by developers in an efficient manner. This distribution of information and the overwhelming number of mostly textual information sources, hampers processes to be conducted and to exploit available knowledge, as defined by some organization.

Problems

Software development guidance, which directly influences developers' work without large efforts, requires a comprehensive, integrated, contextual view on available information.

- Available information, such as specifications, documentation, guidelines, or best practices is only informally available in form of, for example, natural language text, and can not be processed automatically.
- Due to high-level process definitions, there is no support for the assignment of activities and guidance on the concrete level, on which developers are working at project runtime.
- Available information is integrated insufficiently with the process or, in particular, with individual development activities, and can not be provided with developers automatically.

Challenge 2

Enable the comprehensive design of processes and constituent parts, which incorporate various information sources in form of computer-interpretable contextual knowledge, that suits developers' guidance needs and enables them to focus creative parts of their work.

Development Process Execution and Automated Guidance

Once developed, defined process descriptions and all associated information should directly influence the performance of actual work. However, software process models are processed seldom, but used for documentation only. Additionally, error-prone and time-consuming tasks, such as the validation of design guidelines or the appropriate usage of modeling formalisms, change impact analysis across development artifacts, or the assignment of individual development steps, are done manually or supported insufficiently. Therefore, mechanisms are missing to bridge the gap between process documentation and process execution on operational level, whereby the proper performance of processes and the correct application of guidance information to real projects can be ensured. In parallel, as many advantages as possible, which are associated with computer-supported guidance of development processes, should be exploited.

Problems

Today's software development processes are for documentation purposes, while the information encoded in these processes is used insufficiently. As a result, developers are confronted with simple standard activities more than focusing essential and creative development activities.

- There is a gap between process descriptions and current project performance.
- The coordination of tasks, which have to be conducted during a development process, as well as, an appropriate application of information associated with a task proves to be difficult.
- During the development, the change of intermediate products may cause unknown or negative impacts on dependent products. Actually, such influences must be considered by developers themselves, which are not supported in an appropriate way.
- In order to prove the compliance with standards and regulations, the traceability and processing of relevant intermediate products, such as documents or specifications, must be ensured. Actually, this is part of time-consuming manual tasks, which are conducted without efficient computerized support.

Challenge 3

Enable the automated processing of development process design information, in order to guide developers through process activities, while providing contextual information and conducting general administrative tasks, such as traceability and consistency management between artifacts, automatically.

1.2 Objectives, Approach and Contributions

This section identifies objectives to deal with the problems and challenges in modern software development processes, as outlined in Section 1.1. Therefore, we provide an overview of identified objectives, as illustrated in Figure 1.1, and list the main contributions of this thesis. To achieve these contributions, our approach follows the design-science paradigm of information systems research [HMPR04], by creating new and innovative artifacts to extend the boundaries of human and organizational capabilities.

Situational and Project-dependent Development Processes

Figure 1.1 depicts the design and situational configuration of development processes on stakeholder-specific abstraction levels. On business level, abstract process descriptions are (re-)used to be refined on technical level with automation support. On each level and in between, appropriate information management is enabled by adequate techniques and tools in order to overcome *Challenge 1*.

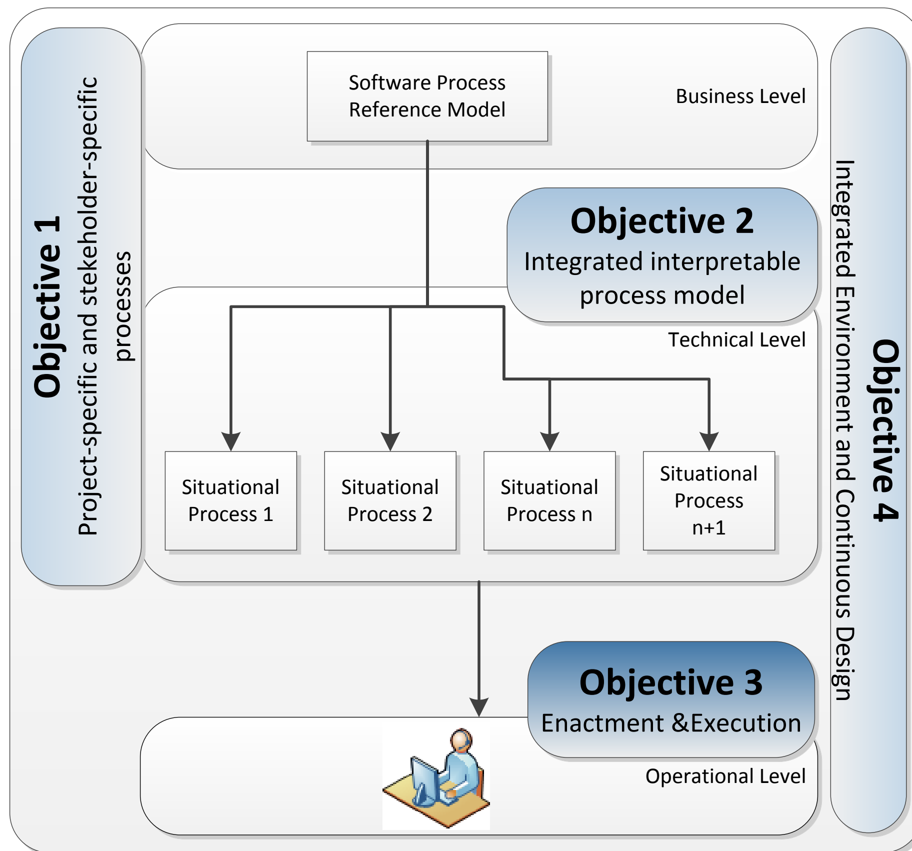


Figure 1.1: Objectives overview

Objective 1

Enable a reuse-based and scalable management of project-specific development processes, that faces the individual needs of business and technical experts in equal shares.

Contributions

To enable Situational and Project-dependent Development Processes, we developed the following artifacts:

- We adopt the paradigm of Software Product Lines and Model Driven Software Development to software development process design in order to provide managed variability and reuse capabilities with software processes on different abstraction levels. The resulting software process line approach considers business-oriented and technical process design needs, and is combined with techniques known from the method engineering research area, in order to deal with the assembly of different process-relevant information sources in a flexible way.
- For software development process tailoring, we provide a planning-based transformation between the stakeholder-specific abstractions, which uses situational char-

acteristics of process constituent parts and feature modeling techniques, to bind variabilities in a technically-enhanced, i.e., computer-interpretable, process.

- In order to enable and simplify the realization of a software development process line, adequate tools and transformations are developed, and integrated into a prototypical environment.

Design of Sound Process Descriptions Enabling Automated Guidance

As today's process descriptions are mainly business-centered and for documentation purposes, we provide means by which process information is complemented by additional computer-interpretable information on technical level. On that level, we provide a sound control-flow semantics, which enables an automated assignment of activities, that are associated with processable information models, which enable automated guidance capabilities and the automation of time-consuming or error-prone standard activities to overcome *Challenge 2*.

Objective 2

Close the gap between process models and other information sources in a way, that enables provision of appropriate information with specific activities and developers at project execution time.

Contributions

To enable the design of sound process descriptions, which allow automated guidance, we developed the following artifacts:

- We introduce a new control-flow semantics for software development process models, which satisfies the flexible needs, which are required for the execution of creative software development activities.
- We combine method engineering techniques with an aspect-oriented mechanism, by which process-related information can be complemented with additional relevant information sources.
- We apply model-driven development techniques in order to integrate information about the content of work products, tooling capabilities, and guidelines with process information in computer-interpretable way.

Development Process Execution and Automated Guidance

Information, which is incorporated with software process design models, should influence the work of developers more efficiently on operational level to overcome *Challenge 3*. Therefore, means are provided to interpret that information automatically. This concerns the generation of process-enabling tools from process models, the interpretation of workflow models for the assignment of development activities and contextual information, as well as, the analysis of monitored runtime information for efficient development process management.

Objective 3

Close the gap between process descriptions on design level and current process execution on operational level, and exploit additional benefits, which come from computer-interpretable development processes.

Contributions

To enable development process execution and automated guidance, we developed the following artifacts:

- Software development process information is made explicit for developers by the means of a prototypically implemented workflow interpreter, that enables a flexible, model-driven workflow management in form of the assignment of activities, associated Computer-aided Software Engineering (CASE) tools, relevant guidelines and other contextual information.
- We develop a model-driven CASE tool generator, which uses software process model information in order to provide activity-specific editor capabilities.
- We develop a model-to-model transformation in order to provide modeled guidelines with a translational semantics, which enables them to be ensured in the context of respective development activities, as defined by the process model.
- We develop a mechanism, by which work product-specific process model information is used to assign performed modeling activities and affected design information with work products at process runtime. By using monitored information, enhanced workflow management based on change impact analysis is introduced to avoid inconsistencies between artifacts.

Overall Objective

In order to efficiently support developers in their daily work, our overall objective is to provide a comprehensive approach, which faces all challenges within one integrated environment. Therefore, we have to align all abstraction levels from business to technical down to the operational level with appropriate tool support. Secondly, since we aim at facing the challenges, which arise from model-driven technologies in the embedded sector, we particularly have to provide developers with support for the appropriate application of that paradigm at project runtime. Therefore, we aim at the provision of relevant information for product development in form of domain-specific languages, which are restricted to process-relevant needs, i.e., we aim at a process-centric domain-specific language.

1.3 Outline

The overall structure of this thesis is illustrated in Figure 1.2, whereby the arrows indicate a suggested reading sequence. Readers, which are familiar with the technical background may skip chapter 2 or parts of it. While chapter 4 and chapter 5 are strongly related to

each other and should be read in sequence, chapter 3 can be read separately. After the main parts of our approach are described, they are evaluated in chapter 6 and compared in detail with related work in chapter 7. Finally, chapter 8 concludes our thesis by summarizing the contributions and providing an outlook for further research.

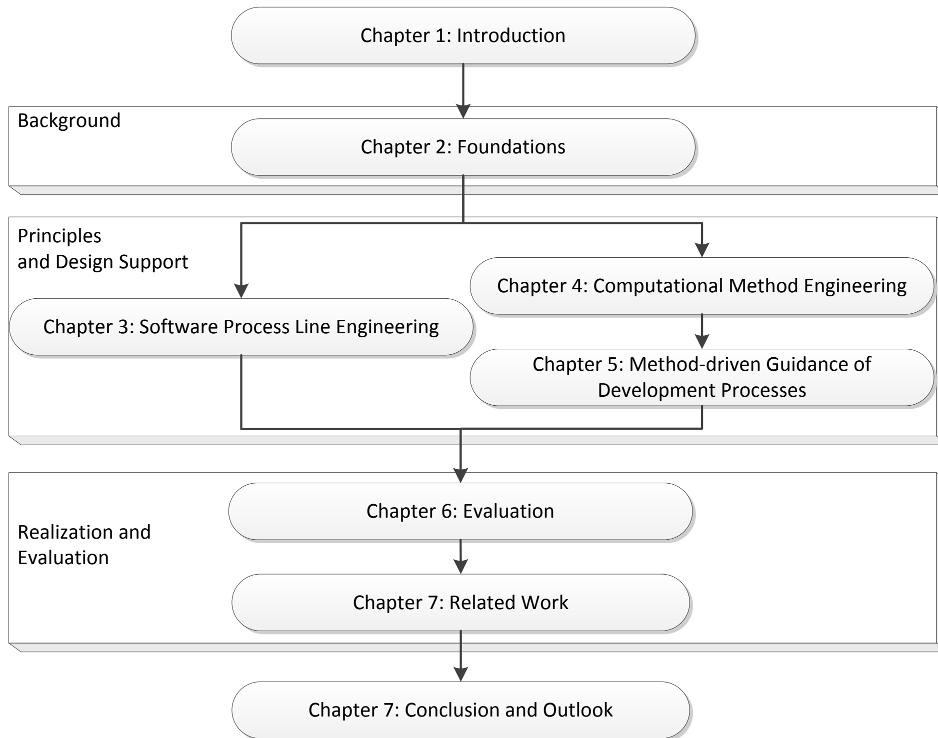


Figure 1.2: Overview about the chapters of this thesis

Chapter 1 provides an introduction and motivates our thesis. It details our aimed application environment, for which we identified problems and challenges. Based on that, we derive objectives to be faced in this thesis and list our contributions.

Chapter 2 first describes the actual research activities in the automotive industry, as an reasonable representative of the embedded software development, to sketch a picture of upcoming challenges. Subsequently, it provides necessary background information about technologies and research areas, which are relevant to this thesis. Thereby, the model-driven engineering paradigm, business process management technologies, method engineering, and software product lines are described.

Chapter 3 describes our software process line approach to overcome the complexity of process-related information management on stakeholder-specific abstraction layers and situational tailoring. We introduce a methodology for setting up and managing a process

line. We further discuss, how situational processes can be derived from the process line using a planning based approach.

Chapter 4 details the technical design of software development processes. Therefore, we first identify requirements, which must be met by such a model, before we describe developed meta models and other information, to complement process models with relevant information, which is sufficient to our requirements. Additionally, we introduce an innovative control-flow semantics to meet an identified need of flexible workflow management.

Chapter 5 details the realization of technical software process models on operational level. Therefore, we describe how individual design information on technical level is interpreted or transformed to achieve relevant means, which supports the guidance of developers at project runtime.

Chapter 6 provides an overview about realization and implementation of our approach and evaluates the approach by different means.

Chapter 7 presents in detail different work, which is related to our approach. Thereby, we discuss characteristics of individual approaches and compare their advantages and disadvantages with respect to our approach.

Chapter 8 summarizes our contributions and concludes with an outlook on future research possibilities, that can be continued based on the results of this thesis.

2 Foundations

To overcome the complexity of development projects a lot of standards, frameworks, and guidelines were developed in the past. For example, the V-Model XT [FHKS09] or maturity models, such as Capability Maturity Model Integration (CMMI) [ACT08] or Software Process Improvement and Capability Determination (SPICE) [Dor93], offer well documented process references. Additionally, there are cross-domain de facto standards, such as Unified Modeling Language (UML) [SB07], Systems Modeling Language (SysML) [OMG08b], or Advancing Traffic Efficiency and Safety through Software Technology (EAST-ADL) [ATE08] serving as design standards, which integrate several domain specific information models on the product side. Moreover, domain specific standards, such as AUTOSAR [AUT12] aim on simplification of information exchange, collaboration, and integration. Other standards, such as XML Metadata Interchange (XMI) [OMG07] or STEP [ISO02], support tool interoperability by common data exchange formats.

On the other side, Situational Method Engineering (SME) [SW94, Har97a, BSH98, MR05, SHK09] is a discipline, which provides strategies and techniques for building situational methods and processes following the above standards, while considering special requirements on individual products, domain specific processes, disciplines, and other required resources. Contrasting SME, which enables the customization of methods and processes, Product Line Engineering [CN01, PBV05, CAF04, Fam05] enables customization of, e.g., software products, based on commonalities and variabilities. By combining SME with Product Line Engineering to Process Line Engineering, where a process line of similar processes uses a common factory, that assembles and configures parts designed to be reused across the varying development processes in the process line, highly tailored processes can be generated to enable realization of above standards, frameworks, and guidelines best. The development processes can be modeled using standard process definition tools and further refined by computer-interpretable information to enable automated guidance, traceability, activity based validation, and best practice capabilities.

This section presents the necessary background on technologies used in the context of our approach, namely Model-driven Engineering, Process Management, Method Engineering, as well as Product Lines. However, before detailing this research areas, we first provide insights to current research initiatives of the automotive sector, to sketch a picture of current challenges motivating our approach.

2.1 Automotive Modeling Initiatives

This thesis focuses on the improvement of model-driven engineering processes and was particularly motivated from the challenges in the automotive software development sector. This domain is influenced by various initiatives, standards, and specifications, and the following introduces the most relevant ones to better understand faced challenges and the examples used in the subsequent chapters.

2.1.1 AUTOSAR

Automotive software development is characterized by a widespread landscape of different Original Equipment Manufacturers (OEMs) and suppliers. A global distributed development of components, as well as, a missing standard for development leads to individual solutions of different suppliers. However, individually created solutions are not able to interact with other components or they are hard to be integrated into one vehicle in a first attempt. Furthermore, the exchange or the upgrading of components turned out to be difficult.

Having these problems in mind, an initiative called AUTOSAR [AUT12] was founded in 2003 to establish a standard modular software infrastructure for application and basic software. This enables exchanging parts of the system's software and describing an embedded automotive system on a technical level close to implementation [RHER07]. The standard allows to describe facets of the software and the hardware of an embedded automotive system by a common format, which is defined in form of a meta model. AUTOSAR enables modularity, scalability, transferability and re-usability of software among projects, variants, suppliers, customers, etc. In general, the main objectives of AUTOSAR are:

- to manage increasing Electric/Electronic (E/E) complexity associated with growth in functional scope
- to improve flexibility for product modification, upgrade and update
- to improve scalability of solutions within and across product lines
- to improve quality and reliability of E/E systems
- to enable detection of errors in early design phases

To reach these objectives and in order to control complexity at the same time, AUTOSAR defines several self-contained description documents. These documents precisely describe individual parts of the architecture and affect relevant properties of a general component based system. Dependencies between these descriptions, as well as, a form of guide line for building these documents are described by the AUTOSAR methodology. The descriptions themselves, i.e., the language and rules for building specifications, are defined by AUTOSAR by the means of a meta model. The AUTOSAR architecture, the methodology, and the meta model are discussed more closely below.

2.1.1.1 AUTOSAR Architecture

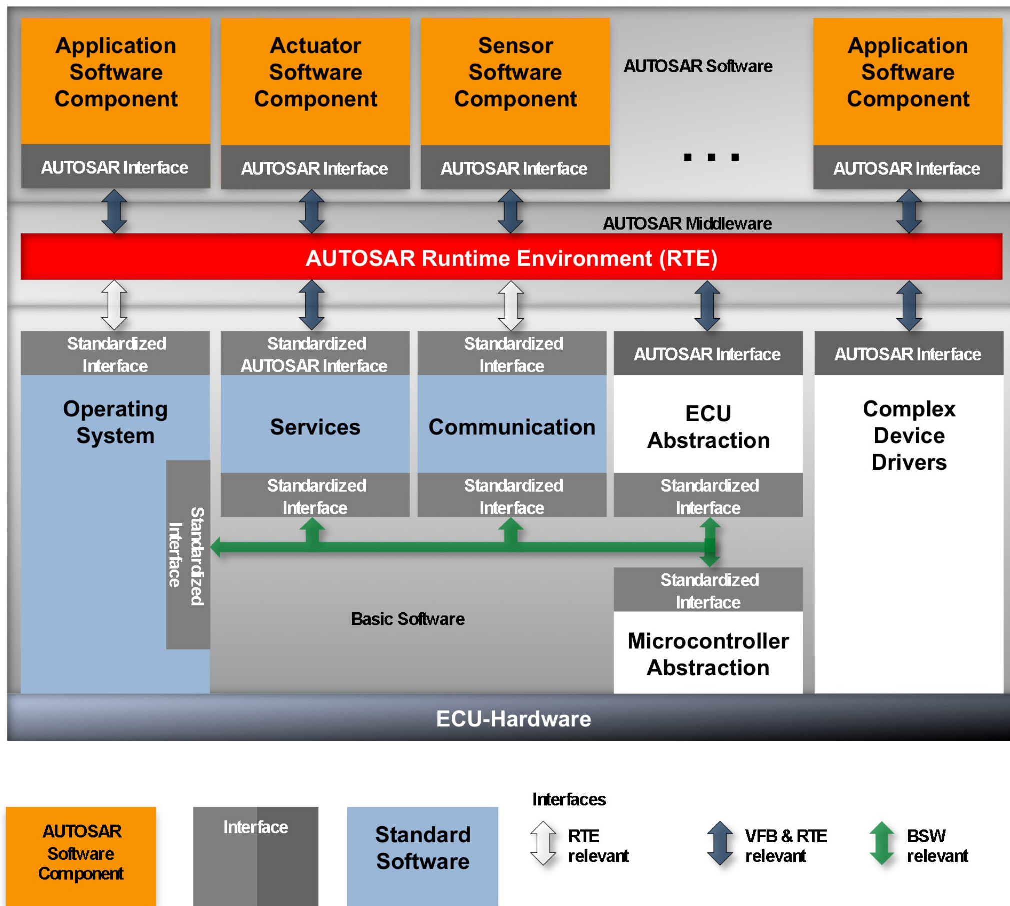


Figure 2.1: AUTOSAR refined layered software architecture from [AUT12]

Figure 2.1 depicts the software architecture specified by AUTOSAR. It is a layered and component based architecture. Above the concrete hardware of micro-controllers or ECUs, AUTOSAR specifies three Abstraction Layers: Basic Software Layer, AUTOSAR Runtime Environment (RTE) and Application Layer.

The Basic Software Layer, which is situated below the AUTOSAR Runtime Environment, abstracts from hardware and provides services to AUTOSAR Software Components by standardized and ECU specific software components. To enable exchangeability of services and for customization of different supplier specific ECUs, the Basic Software Layer is further detailed into three abstraction layers (Service layer, ECU Abstraction Layer, Microcontroller Abstraction Layer), plus the possibility to implement Complex Device Driver, which cannot be mapped into a single layer. Figure 2.1 shows that there are different abstraction layer stacks, whereas each stack stands for an individual func-

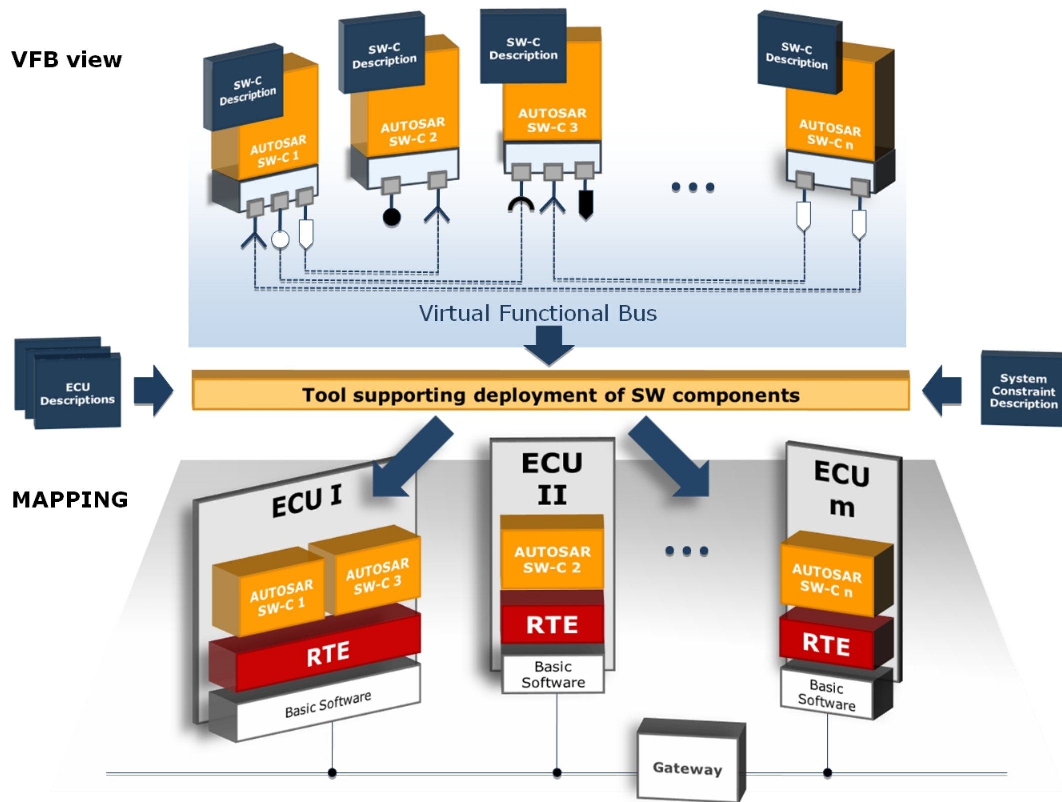


Figure 2.2: Basic AUTOSAR Approach from [AUT12]

tion, such as memory, communication, I/O or device driver functionality.

The next level of the AUTOSAR architecture is the RTE layer. This layer is realized twofold: As RTE and as Virtual Functional Bus (VFB). The RTE realizes the VFB on a concrete ECU and is explicitly generated for each ECU, as depicted in Figure 2.2. The figure shows the relation between VFB and RTE aligned with the basic AUTOSAR approach. In order to fulfill the goal of relocatability, AUTOSAR Software Components are implemented independently from the underlying hardware. This independence is achieved by providing the VFB. The VFB provides a virtual hardware, mapping independent system integration, and abstraction of the AUTOSAR Software Components interconnections. Hence, communication between different software components and between software components and their environment (e.g., hardware driver, operating system, services of the basic software layer, etc.) can be specified independently of any underlying hardware (e.g., BUS systems or microcontrollers). Thereby, communication is realized via standardized AUTOSAR interfaces, which encapsulate each component. The interfaces enable a much earlier integration of AUTOSAR Software Components.

Upside the RTE the Application Layer contains application software components. Application software components use the services of the Basic Software Layer and communicate with other components over the RTE or VFB exclusively.

2.1.1.2 AUTOSAR Meta Model

In order to represent the information describing the aforementioned architecture and its components via a standardized and machine readable format, AUTOSAR defines a meta model according to the MDE paradigm, which enables additional advantages, such as model transformation and automated code generation in the automotive software development. AUTOSAR is defined as a Domain-specific Language (DSL), which satisfies special requirements of an automotive system. By the means of its meta model, AUTOSAR defines an abstract syntax, which provides all relevant language elements, which are necessary for the specification of an automotive system. The meta model is described using the UML, whereas a special UML Profile was created, which can be applied to almost any UML tool in order to use the AUTOSAR syntax.

Furthermore, the meta model and its language elements are subdivided into packages, as depicted in Figure 2.3. The packages, called AUTOSAR Templates, describe information about software components, ECU resources, or the general system. These Templates are detailed in the following.

Figure 2.3 depicts an overview of the templates and relationships among them. The template, which holds all parts together is the **system template**. As the name implies, it describes properties of the entire system. Generally, a filled template defines the relationship between the pure software view on the system and a physical system architecture with networked ECU instances. By means of the system template, five major elements can be defined: Topology, Software, Communication, Mapping and Mapping Constraints. The topology part of the system template describes the physical System Topology of a vehicle modeled in AUTOSAR. This is formed by a number of so called ECU instances which are interconnected to each other in order to form ensembles of ECUs.

Furthermore, the system template composes software elements, which are specified to run on a particular ECU. Each software component is defined more closely by the means of the **software component template**. In order to distribute the software over distributed ECUs, the system template defines a so called system mapping which maps application software components to certain ECUs. Beyond the software to ECU mappings the system mapping also maps the data exchange between software component signals, as well as, the way a signal should take between software components. However, the system mapping also contains further relevant mappings and elements to describe the communication using signals, frames and PDUs, which are explained more closely in the system template specification of AUTOSAR.

Another important template to describe an AUTOSAR system is the **ECU resource template**. It provides syntax for describing and checking the consistency of characteristics and features of automotive ECUs. This template is used to specify the hardware of

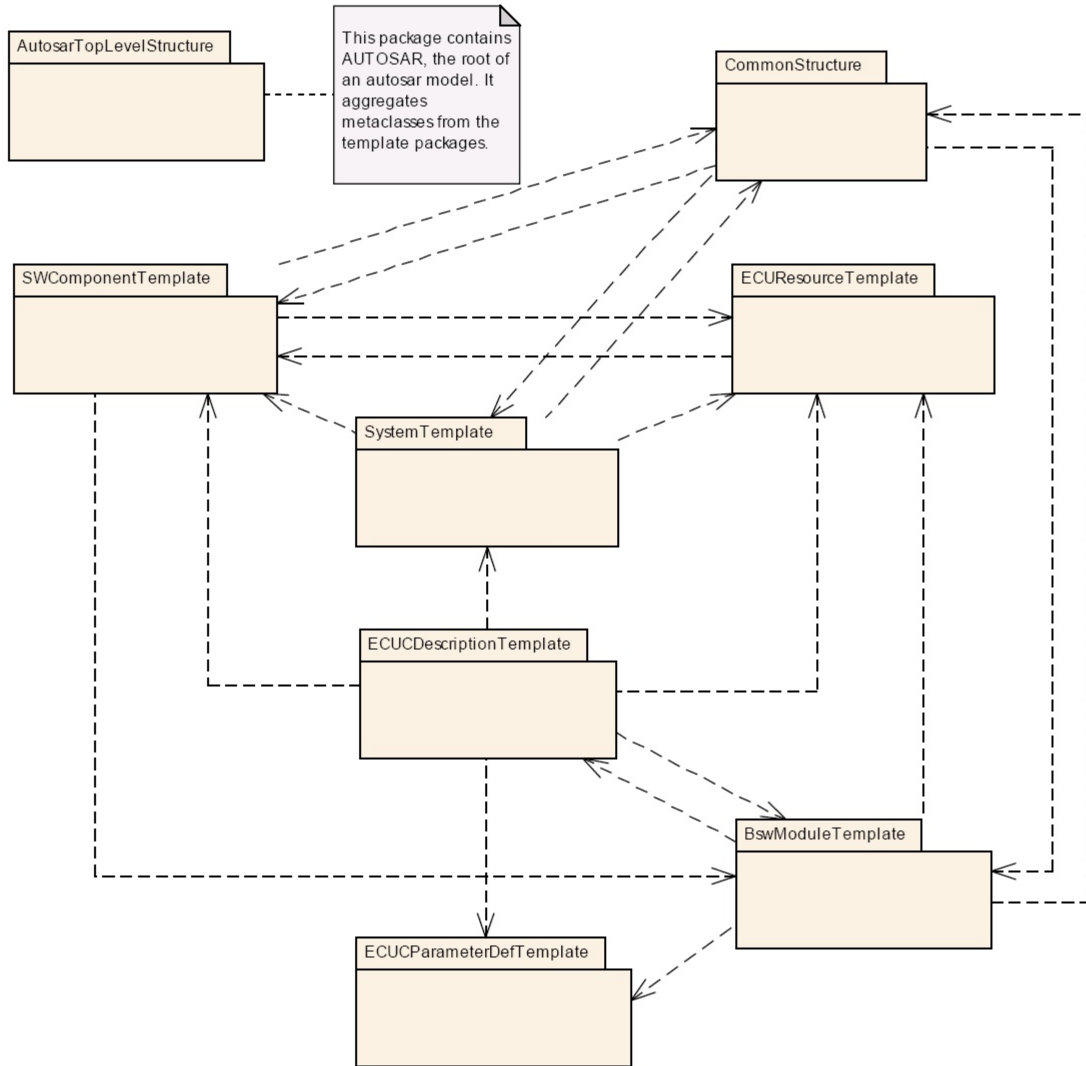


Figure 2.3: AUTOSAR: Package Overview from [AUT12]

ECUs in detail. Hardware ports, memory, processing unit, peripherals and other electronics of an ECU are described by the means of the ECU resource template, which also depends on the system template.
The above mentioned templates build the core of the meta model. The core is complemented by other templates to describe other facets of an AUTOSAR system. They are called **generic structure**, **common structure**, **ECU description template**, **BSW module template**, and **ECUC parameter def template**. The dependencies among these documents or templates are prescribed by the AUTOSAR methodology, as described in the following.

2.1.1.3 Methodology

The AUTOSAR methodology [AUT12] does not define a concrete proceeding or development process. It defines a recommended process, which specifies when certain informa-

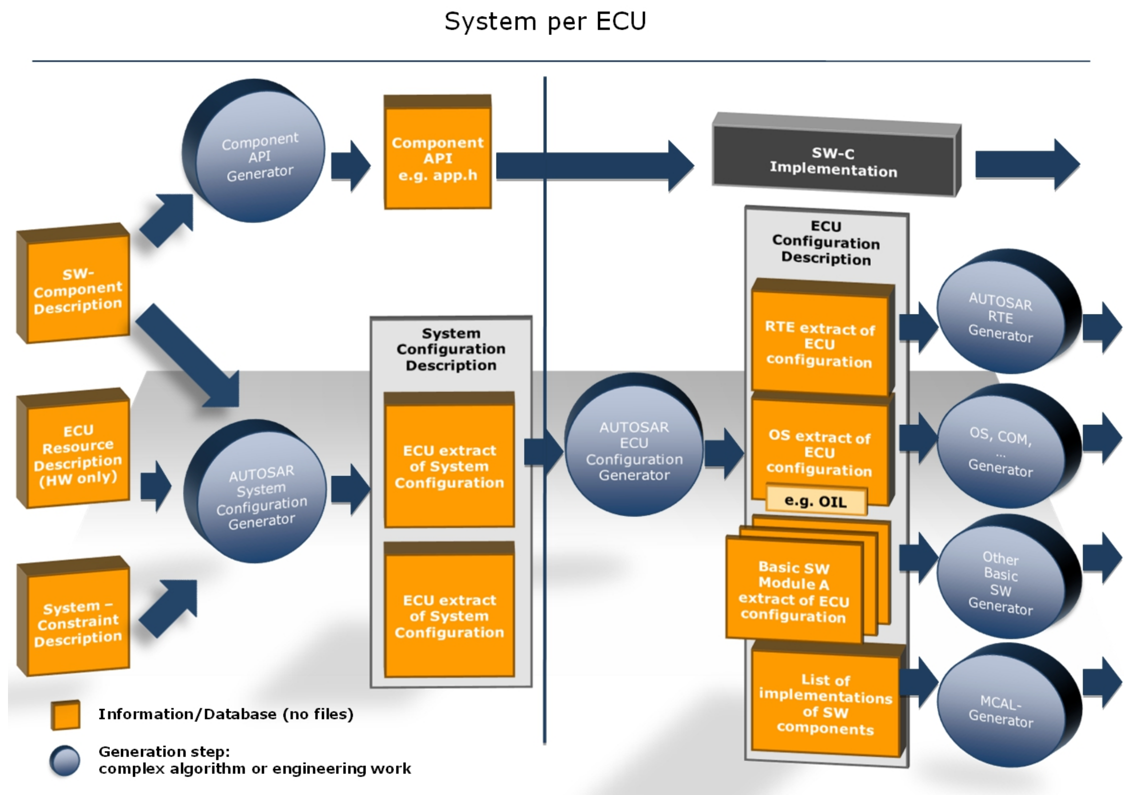


Figure 2.4: Autosar Methodology: Overview from [AUT12]

tion has to be available for further processing. The process describes independently from concrete responsibilities or a detailed schedule the dependencies between documents or artifacts, which have to be produced. In this context, AUTOSAR uses the term work-product for such an information object. Additionally, the methodology describes, which work-products have to be brought together in order to generate new work-products out of several input work-products by specified activities. These steps can partly be automated. The processing of several work-products passes different activities until, at last, all necessary information, which describe the general system, are arranged by the means of multiple descriptions. This means, that the methodology describes dependencies between different information for building a system and steps for generating new work-products inside a workflow. The workflow spans all development phases, from system design to executable ECU-specific code.

Figure 2.4 depicts a simplified view of the AUTOSAR methodology. While the blue circles represent complex engineering tasks or generation steps, the yellow rectangles represent the methodology's main work-products. These work-products are specified by the means of the AUTOSAR template components. Using a schema generator, XMI format can automatically be generated from the meta model or rather from the model itself. For this reason, AUTOSAR specifies so-called **model persistence rules for XML** to enable the exchange of all models by the established standard Extensible Markup Lan-

guage (XML). Further work products may be object code, header files or others. On the other side, the arrows between work-products define process steps for transforming or gathering of necessary information. Details on the AUTOSAR methodology, i.e., further process steps, work-products, and dependencies, may be found in the AUTOSAR Methodology specification [AUT12].

2.1.2 MAENAD

This section describes the MAENAD project, which aimed at the complementation of AUTOSAR on system level and the refinement of a specific Architecture Description Language (ADL) developed for the automotive domain. This ADL, called EAST-ADL, has been developed by multiple agents in the automotive domain for several years, including both OEMs and suppliers. In a previous version, EAST-ADL was developed within the scope of the EAST-EEA ITEA project. Beyond the objective of EAST-ADL to define a standardized Architecture Description Language for modeling all aspects of an (automotive) system, it also targets documentation and a methodology. A resulting architecture ensures interoperability between software and hardware to enable re-usage and exchangeability of distributed components. The ITEA project finished 2004 and was further developed within the ATESST, Advancing Traffic Efficiency and Safety through Software Technology, IST-project. The current version of EAST-ADL is version 2.1 and results from MAENAD, which was an FP7 project funded by the European Commission.

2.1.2.1 EAST-ADL Meta Model

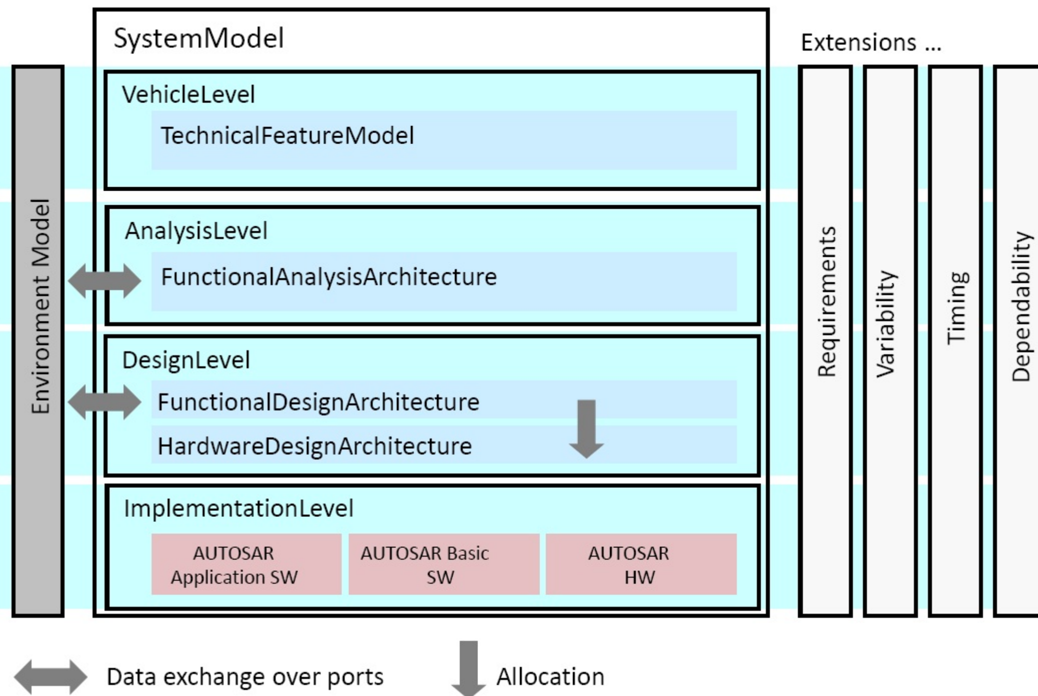


Figure 2.5: EAST-ADL Overview from [MAE12]

EAST-ADL is an UML based solution and an UML profile captures the ADL in a XML file. This enables using well-known tools for tool interaction and model exchange, as well as, open use and standardization. While AUTOSAR focuses implementation-specific details on software level, EAST-ADL complements AUTOSAR on system level with aspects, such as vehicle features, functions, requirements, variability, software components, hardware components and communication. Therefore, EAST-ADL defines language elements to specify characteristics of a system architecture, which can be classified into the following main concerns:

- **Requirements:** The elements of EAST-ADL's requirements language base on SysML constructs and are used to specify two types of requirements: functional requirements, which focus on the "normal" functionality, that a system has to provide, and quality requirements, which focus on non-functional properties of the system.
- **Functional Abstraction:** This part of the specification defines language constructs to capture the functional decomposition and behavior of the embedded system and the environment. The structural view of a model focuses on the static structure of components of the system and its environment and their static relationships. This includes the internal structure and external interfaces. Furthermore, EAST-ADL provides an own simple algorithmic behavior model and enables to reference external defined behaviors, likewise. This makes it possible to reference behavior definitions, which may be specified by the means of other tools and/or mechanisms (e.g., Statemate or Simulink).
- **Behavior Constraints:** the EAST-ADL enables the annotation of requirements, application modes and functions, implementation and resource deployment, and anomalies with different categories of behavior constraints to analyze the dependability and performance of various system aspects.
- **Timing Modeling:** To specify the timing of automotive functions, EAST-ADL enables the design of timing requirements and timing properties by the means of the Timing Augmented Description Language (TADL), which was developed in the TIMMO project, as detailed in the subsequent section.
- **Functional Safety Modeling:** Functional safety modeling is considered throughout the architecture design process for developing a safety critical system, in compliance with the Standard ISO 26262 [ISO10]. Therefore, EAST-ADL provides language constructs for, e.g., vehicle-level hazard analysis and risk assessment, the definition of safety goals and safety requirements, the Automotive Safety Integrity Level (ASIL) decomposition and the error propagation.
- **Verification and Validation:** This area concerns the possibility to describe Testing and Verification strategies or methods.
- **Variability:** Language elements of the variability package are used to enable the description of various variants of architecture's artifacts in, e.g., a product line.

2.1.2.2 Methodology

Similar to AUTOSAR, EAST-ADL provides a methodology to guide the usage of the language for the construction, validation, and reuse of models for automotive embedded software, without considering a specific development process. As depicted in Figure 2.6, the methodology defines a core part, which is complemented by extensions. The core part describes the central activities in using a top-down approach: During the vehicle phase external requirements are analyzed, before a technical feature model is constructed. The technical feature model is used to derive the FunctionalAnalysisArchitecture during the analysis phase. This is a logical representation of the system, which neglects the distinction between hardware or software. During the design phase the FunctionalAnalysisArchitecture is refined into the FunctionalDesignArchitecture, which details sets of (structured) hardware and software components and their interfaces, a hardware architecture, and a mapping from functional components to hardware or software. The implementation phase of EAST-ADL is mainly a reference to the concepts of AUTOSAR, to provide an implementation and configuration of the final solution.

Each of these phases, is extended by validation and verification activities and a set

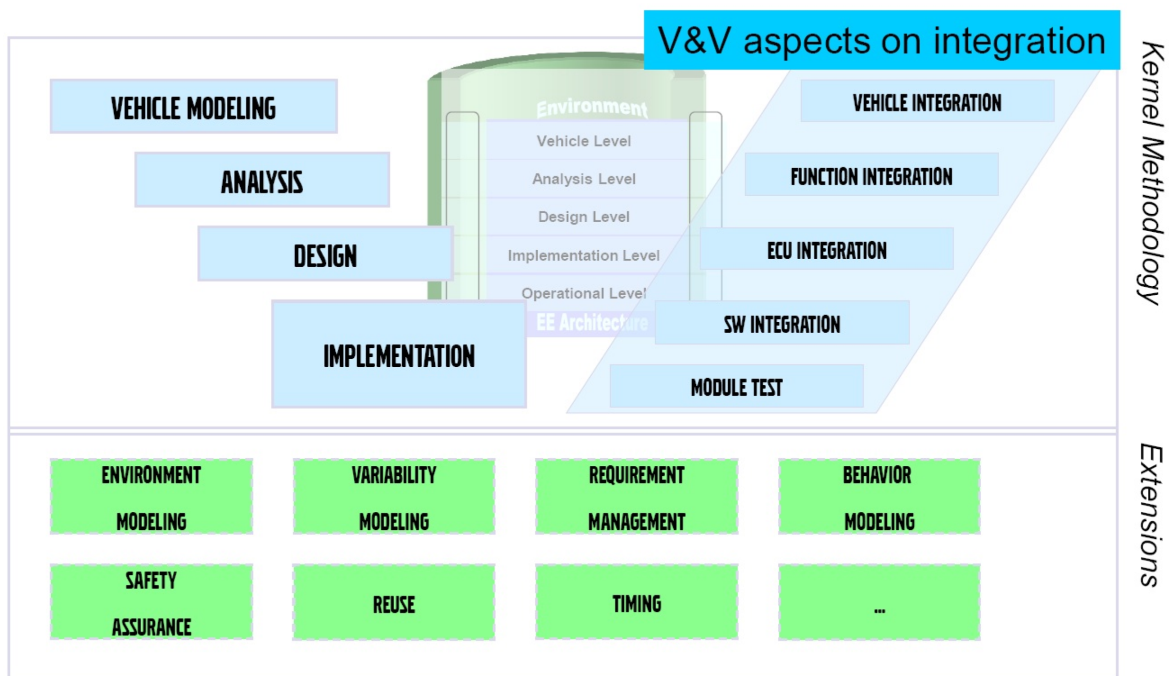


Figure 2.6: EAST-ADL Methodology Overview from [MAE12]

of methodological extensions, such as Environment Modeling, Safety Assurance, Timing, Variability Modeling, or Behavior modeling. After the first version of EAST-ADL's methodology was defined using Software Process Engineering Metamodel (SPEM) to define a set of elementary work tasks, which produce a set of output artifacts from a set of input artifacts, the current version was restructured to follow a Generic Methodology Pattern, which splits each phase of the EAST-ADL core part and each relevant extension into one extensible set of generic default activities.

2.1.3 TIMMO-2-USE

TIMMO-2-USE stands for TIMing MOdel - TOols, algorithms, languages, methodology, and USE cases and was an ITEA2 project, which started in October 2010 for a duration of two years. The main objective of TIMOO-2-USE was the development of different types of timing constraints for the design of distributed real-time automotive systems. Beside that, the project focused on the development and validation of tools, algorithms, languages, and a methodology, based on the results of its predecessor project TIMMO.

2.1.3.1 TADL Meta Model

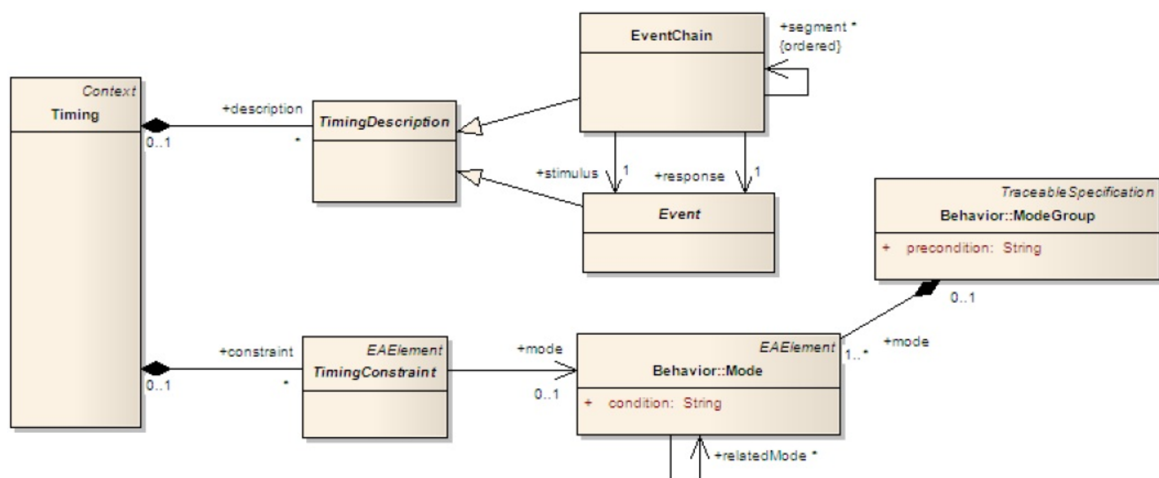


Figure 2.7: Basic TADL2 elements, from [tim09]

In TIMMO-2-USE, version two of a meta model, called TADL, was developed to formalize timing constraints of a system specified by the means of, e.g., EAST-ADL and AUTOSAR. An overview of the meta model is given in Figure 2.7. Generally, a Timing element represents a collection of timing descriptions, which are composed of events and event chains, and the timing constraint to indicate that the corresponding TimingConstraint is only valid when the specified mode is active. These can roughly be grouped into restrictions on the recurring delays between a pair of events, restrictions on the repetitions of a single event, and restrictions on the synchronicity of a set of events. Additionally, the meta model specifies various concrete realizations of events and constraints to concertize timing information according to the various levels of abstractions, as defined by EAST-ADL and AUTOSAR.

2.1.3.2 Methodology

The methodology is strongly related to previous projects, such as TIMMO and ATESST2 (the predecessor project of MAENAD), which considered timing aspects, as well. In contrast, while the methodology of TIMMO-2-USE considers a top-down and a bottom-up

development, it focuses on a certain set of use cases related to timing, that are mapped to the Generic Methodology Pattern (GMP) mentioned above. The pattern is applied on all levels of abstraction, as defined by the EAST-ADL: Vehicle, Analysis, Design, and Implementation. While TIMMO-2-USE also sketches the idea of transforming timing information from one level to another, the GMP is applied on each level according to its corresponding level of detail. The GMP consists of the following 6 generic tasks:

1. Create Solution: During this task, an architecture has to be defined without any timing information.
2. Attach Timing Requirements to Solution: During this task, timing requirements have to be defined with regard to the current architecture.
3. Create Timing Model: During this task, a formalized model for the calculation of specific timing characteristics based on properties of the current architecture has to be developed.
4. Analyze Timing Model: During this task, the timing model has to be evaluated and results have to be described.
5. Verify Solution against Timing Requirements: During this task, the obtained analysis results have to be compared with the specified timing requirements.
6. Specify and Validate Timing Requirements: During this task, mandatory timing characteristics have to be identified as timing requirements, which are propagated to the next development phase.

2.1.4 Conclusions from above Initiatives

Previous section have exemplified, that various efforts were made during the last years to manage the various concerns of the development of the complex embedded system of a vehicle. Similar to e.g., AUTOSAR, MAENAD, or TIMMO-2USE, a multitude of different initiatives, such as Automotive SPICE, the ISO26262, ASAM, MSR, etc. , were established to develop new means in form of meta models or DSLs, guidelines, best practices, and/or reference methodologies. However, not only the automotive sector faced such challenges, but other domains make similar efforts to overcome similar challenges.
All these efforts result in a overwhelming mass of information, which is organized in a plenty of "paperware" documents to aim at the support of even more development projects. However, it is nearly impossible for any party, which is involved in development, to be compliant with all specifications, at the same time. It is even more difficult for responsible roles to keep all the fast moving information in mind. This, particularly, influences the correct application of language elements, which were defined by some meta model and its evolutions, as well as, an appropriate order of required development activities to produce corresponding output.
Therefore, to support, in particular, humans in doing their daily work, i.e., providing them with situational information following a predefined methodology, the following approach was developed. By combining process-related with product-related information from various sources into one framework, an integrated information management is

enabled. Furthermore, we not only combine information, but also refine the for the most time textual information on a technical level, to automate guidance capabilities on operational level. As a result, training periods will be reduced and up-to-date information are applied accordingly.

2.2 Model-Driven Engineering and Semantic Technologies

In this section, we describe Technical Spaces (TSs), which are relevant for this thesis. According to Kurtev et al. [KBA02b], an TS *is a working context with a set of associated concepts, body of knowledge, tools, required skills, and possibilities*. Initially, five technological spaces were identified by Kurtev et al. : abstract syntax TS, XML TS, Data Base Management Systems (DBMS) TS, Ontology engineering Ontological Technical Space (OTS), and MDA as defined by the OMG. As some operations may be performed easier in one space than in another, bridges are needed between them to combine the different facilities, as demonstrated e.g., in [GDDD04]. Our approach was realized based on two of these TSs, namely the OTS and Meta-modeling Technical Space (MMTS), whereas the main parts were realized in the latter TS. These two technical spaces are detailed in following:

2.2.1 Meta-modeling Technical Space

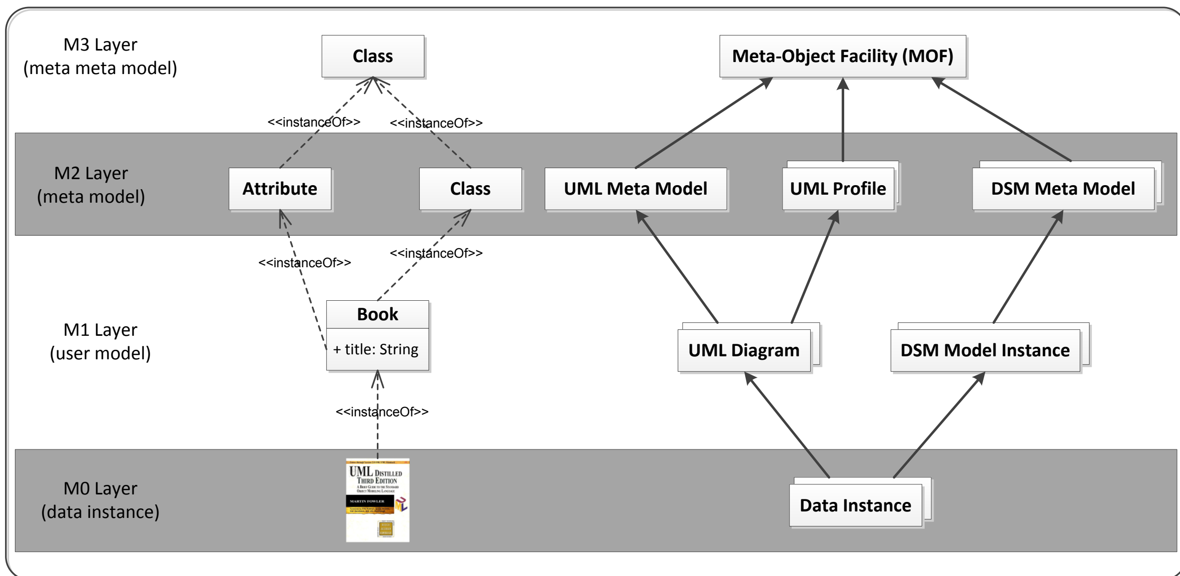


Figure 2.8: Meta modeling layers, from [Lid11]

MDE [Ken02], MDSD [BBG05b], or OMG's MDA [OMG03] approach are prominent exponents of the software engineering paradigm in the MMTS, which promotes the creation and processing of models as a simplified abstraction of real world phenomena. Models are described by the means of modeling language, which is defined by the means

of a meta model. The meta model defines the abstract syntax, i.e., the relevant vocabulary and a set of construction rules, to create a valid set of models. Additionally, because of each model is represented using a particular modeling style, an abstract syntax must be complemented by a concrete syntax, which specifies the concrete representation (e.g., textual or graphical) of defined language elements. Beside that, models must provide a semantics to make the language (computer-)interpretable, understandable, and analyzable. Following the classification of programming language semantics from [SK05], we distinguish four types of semantics:

1. the translational semantics definition translates concepts of the vocabulary to concepts with a well-defined semantics in another language
2. the extensional semantics definition extends the semantics of existing concepts, which already provide semantic information
3. the operational semantics definition uses the language itself to specify the operational behavior of vocabulary concepts explicitly (see also [Mos92])
4. the denotational semantics definition specifies a declarative mapping to semantic domain concepts

Figure 2.8 illustrates MMTS's basic philosophy of modeling on four different meta modeling layers: while real world phenomena, such as a book, a vehicle, or a concrete person, belong to the real world M0 layer, a user model abstract relevant real world entities, on the next layer M1. This user model corresponds to its meta layer specification, which is situated on the next higher layer M2 to define all language elements. This principle is demonstrated in Figure 2.8, where a book of the real world, is modeled on meta layer M1 using the concepts and relationships between them, as defined on meta layer M2, i.e., a book is a class, which has an attribute to indicate a book's name. The final meta layer M3, again abstracts the concepts used to define a meta model on meta layer M2. On meta layer M3, the meta-meta model defines the abstract elements, such as classes, properties, and relations, for the definition of meta models in general. The meta-meta model on the M3 layer is self-describing, i.e., all elements of the meta-meta model are described using already defined elements of the same meta layer. A prominent example of a meta-meta model is Meta Object Facility (MOF) [OMG06b], which is a standard published by the OMG.

During the software development process, a number of different meta models can be applied to create various models and to refine them on different abstraction levels. Thereby, each model is written in the language of a corresponding meta-model's abstract syntax using its concrete syntax. Various meta models are available for different areas, such as the general-purpose language UML, the automotive-specific AUTOSAR, or the Common Warehouse Meta model (CWM). The models are grouped into abstraction levels or packages to provide particular views on the system being built and to satisfy various stakeholder needs. Finally, models provide information to enable the generation of platform-specific application code.

2.2.1.1 Model Transformations

Beside models and meta models, model transformations play a major role in the MMTS. Basically, two types of model transformations are distinguished: Model-to-Model (M2M) transformations and Model-to-Text (M2T) transformations. Both types use information of one or more input models to produce new information in form of a model or text. While an M2M transformation is called horizontal, if it maps between two models of the same abstraction level, an M2M transformation is called vertical, if it defines a mapping between models of different abstraction levels, e.g., in form of a refinement. In particular, a transformation is called M2T, if it uses the input models to produce text-based output, such as code of a programming language or other text. M2T transformation is also called code generation.

To support model transformation, various frameworks are available for M2M and M2T. A prominent example for M2M transformation is the Query View Transformation (QVT) [OMG05] specification, which is part of MOF 2.0 and defined by the OMG. QVT defines three model transformations languages: QVT-Operational, which is a imperative language for unidirectional transformations, QVT-Relations, which is a declarative language enabling bidirectional transformation, and the declarative QVT core, which is a more simple but less expressive language than QVT-Relations. Another popular transformation framework and toolkit is ATLAS Transformation Language (ATL), which initially was developed by the INRIA ATLAS group. Meanwhile, it is released and published as an M2M component of the Eclipse Modeling project. On the other side, various frameworks, such as Java Emitter Template (JET) or XPAND, are available to transform models into code. There is an exhaustive number of model transformation frameworks, which would go beyond the scope of this thesis. However, a very detailed list of available frameworks, as well as, their strengths and weaknesses, can be found in [Ros08].

2.2.1.2 Meta Model Extension Mechanisms

Although, there are many predefined meta models available out of the box, such as the general-purpose modeling language UML, which can be used immediately, sometimes they are not sufficient to support a planned project best. Therefore, basically, three approaches exist to customize or extend meta models for a domain-specific needs:

- Meta model-based Extension: To extend a meta model, this type of extension mechanism, uses language elements of the next higher meta layer, i.e., language elements of the meta meta model, which originally defined the meta model to be extended. For example, to extend UML, which is situated on meta layer M2, this mechanism would apply language elements from meta layer M3 (MOF). This kind of extension is stable, frozen, and more formal.
- Stereotype-based Extensions: Extensions based on stereotyping is a specific UML mechanism, which provides a flexible and lightweight means for extending the UML meta model. By the means of UML-Profiles, stereotypes, constraints, and tagged values, existing language elements can be restricted to establish a new modeling language.

- Aspect-oriented Extensions: the aspect-oriented mechanism is similar to stereotyping, since it does not change the original meta model or modeling tool. In contrast, some external aspects are defined and automatically woven into the existing meta model in a defined way.

In this thesis, we mainly applied the meta model-based extension mechanism to develop new meta models, which we used for the creation of additional design models to extend existing process models. Beside that, we also applied the aspect-oriented extension mechanism to weave the additionally created models with existing process models as aspects. The mechanisms, which was developed in [Lau10], is based on aspect-oriented modeling (e.g., [SRF+]), which distinguishes between positive and negative model variability to add or remove elements from a given core model. In the approach, which is shown in Figure 2.9 from its conceptual viewpoint, positive variability weaves externally defined aspects into a given core meta model. This works as follows: a configuration model (ConfModel) links the meta model to be extended with a relevant set of defined profiles (Profile). Similar to UML-Profile, a profile defines various aspects representing additional properties, which have to be woven with target meta model elements (Class) of the enriched meta model. Concrete implementations of a defined aspect (AspectInstance) are associated with the configuration model.

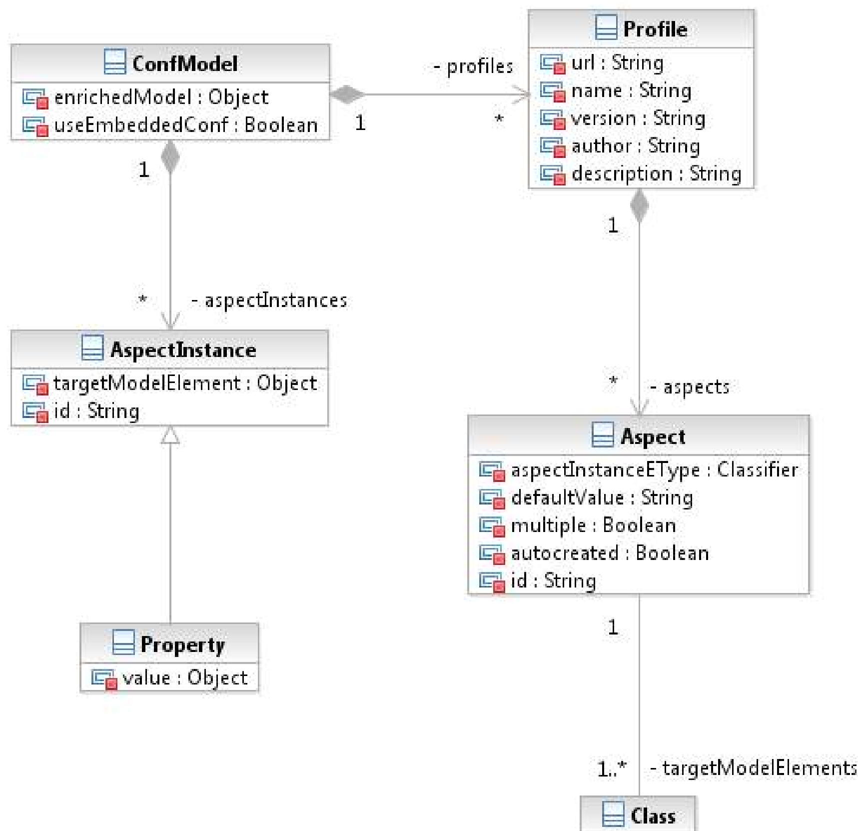


Figure 2.9: Aspect-oriented base model

2.2.2 Ontological Technical Space

The second TS, which is relevant to thesis, is the Ontological Technical Space. In this TS, which can be considered as a subfield of knowledge engineering, standards and techniques are provided to represent data and to reason about expressed data. In general, entities and relations in a specific domain [UG96] are described using a formalized language (e.g., logic), by which an ontology is created. Following the most cited definition from [Gru93], an "ontology is a explicit specification of a conceptualization", whereas conceptualization refers to an abstract, simplified representation of a real world phenomenon, similar to a model in the MMTS. However, while all ontologies are models, not all models are ontologies, as outlined in [AGK06]. The main difference is, that ontologies are built upon logical structures, such as Description Logic (DL) [BHS04], on which specialized reasoning software can be used to automatically process modeled knowledge to check for consistency and to infer new facts (cf. [BL04]).

The Semantic Web [BL01] is one initiative, which provides accepted standards and techniques in the OTS. Most of the Semantic Web languages are standardized through the World Wide Web Consortium (W3C) and are based on XML, which is a standardized format to structure data. However, XML is not interpretable per se. Therefore, Semantic Web languages, such as Resource Description Framework (RDF) [BM04], RDF Schema [BG04], and Web Ontology Language (OWL) [BHH+04] underpin XML with the required formal model, which enables machine interpretation and reasoning. By the means of RDF, data are represented in form of triples or statements, which consist of a subject (a resource), a predicate (a property of the subject), and an object (the value of the predicate). While RDF only provides means to specify ontological information on instance level [GDD09], RDF-Schema extends RDF by the capability to create taxonomies and ontologies. Therefore, it additionally provides language constructs, such as *Class*, *subClassOf*, and *subPropertyOf*. OWL [BHH+04] and the subsequent version OWL 2 [MPSH08] are more powerful languages to describe knowledge in the OTS. The ontology language, which we used in Section 3.5.4, is OWL 2, which is based on the logic $\mathcal{SROIQ}^{D+}$ [HKS06]. In addition to OWL 1 and RDF Schema, OWL 2 provides the concepts of disjoint roles, own data types, and the definition of types, when restricting cardinalities.

A number of popular modeling tools, such as Protege, the NeOn Toolkit, or KAON are available to enable the definition of information and relations by the means of an ontology. Furthermore, related technologies, standards, and tools for this knowledge representation quickly emerged, and established this branch of research area as a well-recognized area of computer science over the past years.

2.2.3 Guidelines, Best Practices, and Validation

Regardless from a concrete technical space, effective information management and design requires experiences and expert knowledge, which evolves over time. Therefore, knowledge about identified challenges and approved solution strategies often is recorded

to avoid repetitive inefficient working. In particular, for software engineering or programming, the term software pattern was established to provide means for efficient solutions of recurring and complex problems. According to [RW05], software patterns can be further categorized into three groups:

- Architectural Style (cf. [GEM10]) expresses a fundamental structural organization schema for software systems. It provides a set of predefined element types, specifies their responsibilities, and includes rules and guidelines for organizing the relationships between them. Examples of an architectural style are client-server, pipes and filters, or peer-to-peer.
- Design Pattern (cf. [GHJ94]) provides schema for refining the elements of a software system or the relationships between them. It describes a commonly recurring structure of interconnected design elements, that solves a general design problem within a particular context. Examples of a design pattern are the observer pattern, the singleton pattern, or the visitor pattern.
- Idiom (cf. [Laf96]) is a low-level pattern specific to a programming language. An idiom describes how to implement particular aspects of elements or the relationships between them by the using the features of a given language. Examples of an idiom are the Java naming conventions or parameter classes.

Furthermore, general guidelines, which guide an engineer in achieving an intention in a given situation, programming style guidelines, such as Misra C [MIS98], and naming conventions provide organizational rules to reduce efforts and to improve the appearance and readability, when writing program code. All over, patterns, guidelines, and conventions, can be subsumed as best practices defined as *a management or technical practice, that has consistently demonstrated to improve one or more of: Productivity, Cost, Schedule, Quality, User Satisfaction, Predictability of Cost and Schedule* [Wit00]. Practices either impact single tasks of an engineering activity, or they provide general advices for managing a concrete problem spread across various tasks. As the name implies, a **best** practice also must evolve with innovations and new insights.

Likewise, in the MMTS best practices are applied to enhance the quality of models and to avoid recurring problems. Similar to the programming technical space, patterns, styles, conventions, and guidelines are available for efficient modeling. Information about best practices, often is encoded in standards, literature, or organizational documents using natural language text. Other guidelines concerning, for example, the validity of models (i.e., the abstract syntax), is encoded in meta models, whose compliance is ensured as basic functionality of modern modeling environments. On the other side, to validate the static and dynamic semantics of design models, guidelines often are encoded using more formal notations. While dynamic semantics refers to the runtime behavior of models, which can be validated, e.g., by simulation, static semantics concerns the reasonableness of valid models, which can be validated before runtime. Therefore, different formalisms were proposed in literature using, for example, model transformation [BJ06] or graph transformations [ALSS08] to encode guidelines, which ensure the favored static semantics in design models. The most prominent formalism to ensure the static seman-

tics, though, is Object Constraint Language (OCL) [OMG06a] or an ECLIPSE-specific extension Epsilon Validation Language (EVL) [KRPP10]. OCL is a textual language, which is based on the MOF, which enables the association of MOF based models with constraints, invariants, and conditions. These invariants and conditions may realize guidelines, which are evaluated automatically, using an OCL interpreter.

2.3 Business Processes Management

Software development processes, as focused on in this thesis, are a special form of a business process. Therefore, the following section details the main concepts in this field, before Section 2.4 will go into details of software development processes and methodologies.

According to [Dav93] and [HC93], *a **business process** is characterized by five elements: (1) a business process has customers; (2) a business process consists of activities; (3) these activities create value for the customer; (4) activities within a business process are carried out by humans or machines; (5) business processes often involve several organizational units.* That means, that in order to fulfill individual business goals, a business process consists of various activities and sub-activities, which are conducted either by humans or automatically. Hereby, by processing input, an activity creates new value as output, which is delivered to other stakeholders or customers. Activities are carried out by distinct roles with different responsibilities and skills. The performed activities roughly can be categorized into core processes, which directly conduce to the business success, and supporting processes. Depending on an enterprise's strategic direction, software development is considered as a core process or a support process. In either case, different processes must be aligned to enable work and collaboration of IT and human resources efficiently. Therefore, Business Process Management (BPM) extends traditional workflow management [JB96], and provides *methods, techniques, and tools to support the design, enactment, management and analysis of operational business processes* [AHW03]. *A Business Process Management System (BPMS) is a generic software system that is driven by explicit process representations to coordinate the enactment of business processes* [Wes07]. Various organizations, such as the Workflow Management Coalition (WfMC), the OMG, or the Organization for the Advancement of Structured Information Standards (OASIS) are affected in the standardization of process technologies. In 2006, the European Association of Business Process Management (EABPM) was founded as an association of national and European organizations to promote BPM. It is in close collaboration with its American pendant, the Association of Business Process Management Professionals (ABPMP), and published the "Guide to the Business Process Management Common Body of Knowledge" [ABB+09], in 2009.

To capture business process knowledge, normally, a process model is used. A **process model** is a *formalized view of a business process, represented as a co-ordinated (parallel and/or serial) set of process activities that are connected in order to achieve a common goal* [WfM99a]. In general, a process model consists of nodes and directed edges to define an order between

nodes. Hereby, nodes either are activity models to represent work conducted by an enterprise, events to control the behavior of an activity model, or other nodes, which enable individual workflow patterns, as discussed in detail below. Accordingly, *a* ***business process model*** *consists of a set of activity models and execution constraints between them* [Wes07]. To describe business processes, various Process Definition Languages (PDLs) exist, which can be categorized into activity-oriented, object-oriented, role-oriented or speech-act approaches on the one side [KKB96], and descriptive or prescriptive approaches on the other side [Lon93].

Based on a (business) process model, *a concrete case in the operational business of a company, consisting of activity instances* [Wes07] is called a **business process instance**. A ***workflow*** *is the automation of a business process* (instance), *in whole or in part, during which documents, information, or tasks are passed from one participant to another for action, according to a set of procedural rules. Likewise, a* ***workflow model*** *is defined through a directed graph consisting of nodes and edges, which show the control flow of the workflow [SOSF04]. A workflow management system is a software system that defines, creates, and manages the execution of workflows through the use of software, running on one or more workflow engines, which is able to interpret the process definition, interact with workflow participants, and, where required, invoke the use of IT tools and applications* [Wes07].

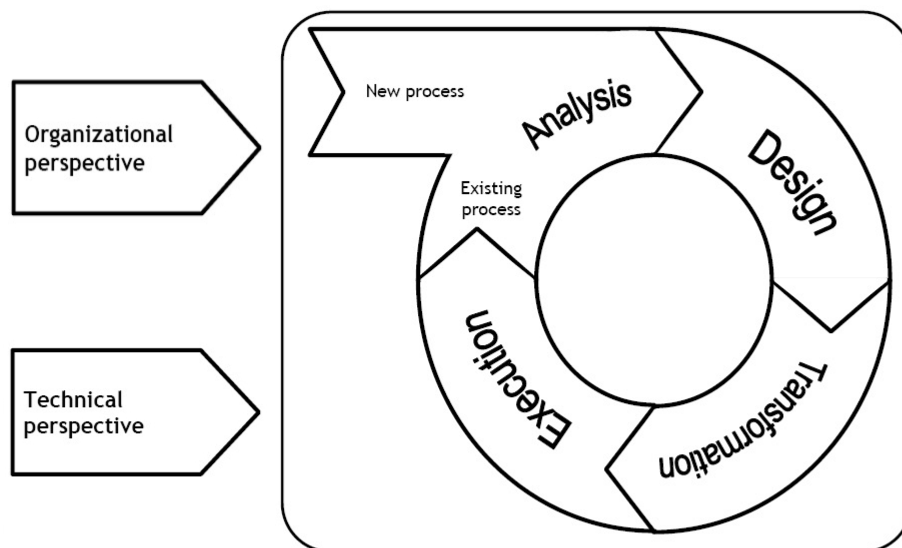


Figure 2.10: Lifecycle of Business Process Management

To achieve BPM, different related activities must be performed. These activities are strongly related to each other and can be organized in a cyclic structure, which is often called the business process life-cycle. Although, many authors, such as Georgakopoulos et al. [GT98], van der Aalst [AHW03], or Weske [Wes07], provide different life-cycle definitions, they basically mean the same on different levels of granularity. In this thesis, we mostly follow the definition of Lautenbacher [Lau10], which is illustrated in Figure 2.10.

- During the analysis phase, stakeholders and business goals are identified to derive relevant business processes. Identified processes are core processes and supporting processes, which are conducted within or across organizations.
- The design phase deals with the specification of identified business processes. Using a given PDL, required activities and responsibilities are defined for each business process, before the order is defined, in which the activities should be performed. Optionally, for each activity, it must be decided, which data or applications are required and whether or not an activity is conducted manually or automatically.
- During the transformation phase, a business process model, which abstracts from technical details, is implemented to fulfill individual technical requirements. These technical requirements can depend on an organization's IT landscape, or technical resources, which are required to automate the process model.
- The execution phase is responsible to process the result of the transformation phase, i.e., the technical process definition. This phase can be subdivided into process enactment, which focuses *on human execution by preparation of task specific artifacts or other contextual task variables* [LQA+], and process execution, which focuses on fully automation of tasks.

After a business process is completed, it has to be analyzed with regard to its potential for optimization and reuse in subsequent projects. Therefore, existing processes pass the business process life-cycle several times.

The life-cycle phases indicated, that the business process life-cycle satisfies various needs of involved stakeholders by the means of different perspectives or viewpoints. In literature, a multitude of perspectives was defined [CKO92, JB96, ASJW05], while there is no consensus about a mandatory set of perspectives on business processes. This depends on the organizational context and its needs. An overview about established perspectives is given in [JB96], from which the most relevant are summarized in the following:

- Functional perspective: The functional perspective decomposes an inter- or intra organizational process landscape (or parts from it) into processes and constituent parts, i.e., subprocesses. Normally, that kind of perspective follows a particular structure, such as an organization's structure, which is given by the organizational perspective.
- Organizational perspective: The organizational perspective organizes an enterprise's responsibilities and roles, and assigns them to the specific tasks of a process or subprocesses from it.
- Behavioral perspective: The behavioral perspective describes the control-flow, i.e., the dependencies and the execution order, of processes, subprocesses, and/or tasks. It defines the start and the end of a control-flow, and it defines, for example, whether parts can be executed in parallel or alternatively.
- Informational perspective: The informational perspectives defines relevant information objects and the data flow, by which information objects are passed. The

information objects, such as documents, parameters, models, are consumed and produced by different processes and activities. One way to define the data flow of these objects, is to follow these dependencies.

- Operational perspective: The operational perspective defines the application programs, which are used during workflow execution. In addition, it defines relevant application interfaces, used technologies, such as Web-Services, JAVA RMI, etc. , and relevant dependencies between applications.

From above perspectives, the behavioral perspective is particular to our thesis (cf. Section 3.5.3). To adequately represent the control-flow in the behavioral perspective, so-called control-flow patterns ([VTKB03, RH06]) are used. Originally, 20 control flow patterns were developed and continuously extended. The basic control-flow patterns sequence, parallel split, synchronization, exclusive choice, and simple merge, practically are realized in almost any modern PDL. While the sequence pattern is used to indicate a consecutive order of tasks in linear order, the parallel split pattern is used to indicate, that two or more branches are meant to be executed concurrently. In contrast, the exclusive choice pattern is used to indicate a decision-based branch of various alternative control-flows. The synchronization pattern and the simple merge patterns are the counterpart patterns to bring split branches together.
In addition to the control flow patterns, data patterns [Nic], resource patterns [RHE04], exception handling patterns [RAH06], and presentation patterns [RWM+11, RHW+11] have been defined.

Furthermore, our approach, particularly, distinguishes the organization or business level from the technical level, as depicted in Figure 2.10: While the organizational level serves as a business-oriented perspective, which enables project management independent from technical details, the technical level complements the business perspective with technical details. These technical details, not only address an organization's IT infrastructure needs, but also information, which are required to guide the activities of a software development process on operational level. Subsequent chapters will show, that we further subdivide the technical perspective into four perspectives in order to represent guidelines, applications, data element, and roles on technical level.

2.3.1 Business Process Management Architectures

As already mentioned, a workflow management system is a software system, that defines, creates, and manages the execution of workflows. Therefore, a workflow management system consists of build-time functions to define workflows, and runtime control functions, which enable the administration of activities and dependencies in between (aka. coordination [MC94, CHW96]), as well as, interactions between humans and the computer system. To provide a common view on workflow management systems, the Workflow Management Coalition defined a reference model, which is depicted in Figure 2.11. The figure shows the main components and interfaces of a workflow architecture.

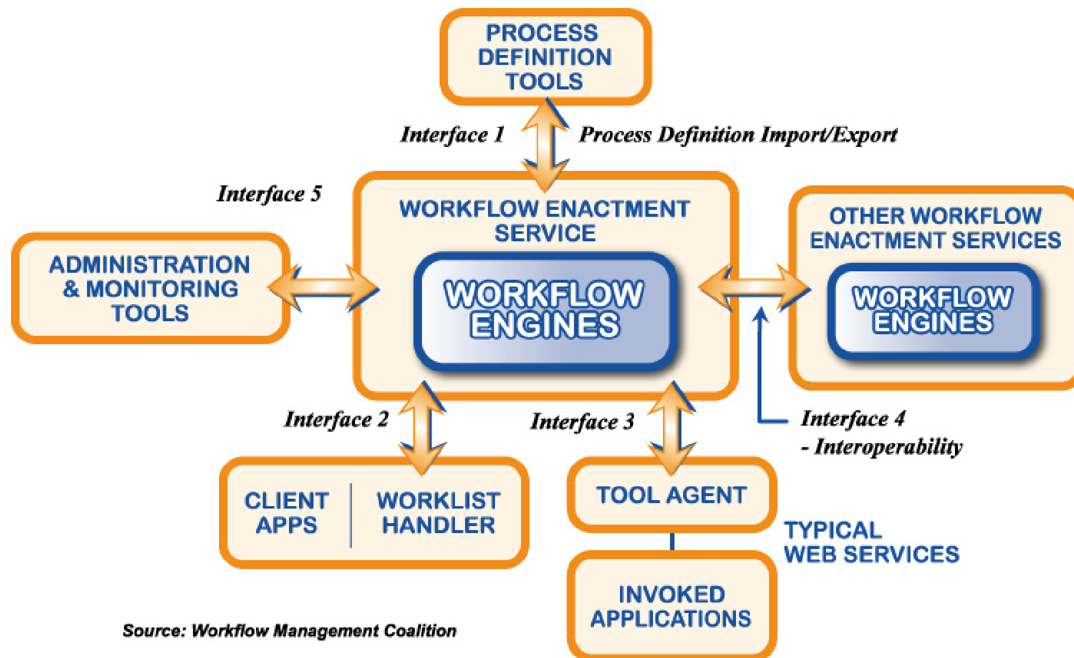


Figure 2.11: Workflow Reference Model - Components & Interfaces from [Hol95]

The Workflow Enactment Service, is the central part of the architecture, which consists of one ore more workflow engines, that provide the runtime execution environment for a workflow instance [Hol95]. A workflow engine provides interpretation of process definitions, which normally are defined in external environment using a graphical notation (cf. Section 2.3.2) and stored in an interpretable format, such as XML Process Definition Language (XPDL) [WfM05], jBPM Process Definition Language (jPDL) [Bae04], or Web Service Business process Execution Language (WSBPEL) [JEA+07]. Beside an interface, which enables the import and export of process definitions from external components, the enactment service has to provide other enactment services from different involved parties with interfaces to monitor and administrate running processes. Additionally, the enactment service should enable workflow client applications (e.g., a worklist handler, which interacts with the end-user) and external applications (e.g., a database) to interact with the workflow engine.

Traditional workflow management, as described above, deals with the strict execution of predefined process models, which is well suited to support business process with static control structures. However, there are different types of business processes, such as a software development process, which are characterized through a highly dynamic environment. Therefore, in [Wes07], Weske discusses the idea of flexible workflow man-

agement. Thereby, to face situations, which can not bee foreseen at build time, a dynamic adaptation of workflow models at runtime is proposed. Following this idea, in Section 5.6, we develop a mechanism to flexibly manage a software development process.

2.3.2 Business Process Modeling

In this section, we give an overview about most popular PDLs for business process modeling, namely UML, Java Workflow Tooling (JWT), SPEM, and Business Process Modeling Notation (BPMN). While the meta models of BPMN, SPEM and UML are standardized by the OMG with a differing focus on business processes and system/software development processes, the proprietary meta model JWT, was developed to provide a lightweight meta model for process modeling in general. A more comprehensive overview about existing PDLs and a comparison of their supported language features can be found in [SAJ02] and [Lau10].

2.3.2.1 Unified Modeling Language

The Unified Modeling Language is a MOF-based meta model, that allows for graphical modeling of static and dynamic structures of software and other systems. It is developed by the OMG and actually available in version 2.4.1. UML is subdivided into an infrastructure part [OMG11a] and a superstructure part [OMG11b] to define the abstract syntax and the static semantics of UML language elements. While the infrastructure part refers to basic language constructs, the superstructure part defines the concrete language elements used in the context of specific diagram type.
Actually, UML supports 7 diagram types to model static structures and 7 diagram types to model the dynamic behavior of a system. Thereby, activity modeling, which is organized in UML's activity packages, emphasizes the sequence and conditions for coordinating lower-level behaviors and provides means to model the flow of a process. Basically, an activity diagram organizes different actions and control nodes, which are connected via activity edges to define the process flow according to a distinct set of control flow patterns. Actions can have inputs and outputs, by which the data flow of the process model can be defined. The completion of the execution of an action, as well as, the occurrence of an external event enable the execution of a set of successor nodes or actions, that take their inputs from the outputs of the predecessor action.

2.3.2.2 Java Workflow Tooling Metamodel

The Java Workflow Tooling meta model, is developed in the context of Eclipse SOA's Java Workflow Tooling project [JWT10]. The meta model is consists of several packages: Core, Processes, Events, References, Organizations, Application, Data and Functions. It

enables the design of nested activities, atomic actions and a distinct set of control nodes, whereof a control flow is defined. Each action is complemented by information about inputs and outputs, as well as, roles and Web-Service Application in addition to general applications, such as a JAVA executable. Furthermore, JWT provides an extension mechanism, that allows to extend the basic meta model with additional features. In [HR07], we demonstrated the transformation of graph-based JWT process models into the block-structured language Business Process Execution Language (BPEL) to be executed using a workflow management system.

2.3.2.3 Business Process Model and Notation

The Business Process Modeling Notation [BPM09] is a standard for business process modeling defined by the OMG. It is similar to UML activity diagrams, and beside a meta model and an exchange format, it provides additional graphical means to describe business processes for technical users and business users. Furthermore, it enables the definition of organizational structures, functional breakdowns and data models, as well as, inter- and intra organizational collaboration and conversation. In particular, BPMN 2.0 defines its own execution semantics, by which a clear and precise understanding of the operation of the elements is described. Additionally, BPMN defines a mapping between process models and WS-BPEL [JEA+07], which enables BPMN models to be executed by the means of a workflow engine.

2.3.2.4 Software Process Engineering Metamodel

The Unified Method Architecture (UMA) is an evolution of SPEM 1.1, which was developed by the OMG in order to provide a meta model, which particularly supports the design of system and software development processes. Based on UMA, in 2005, the OMG started the further development of SPEM, which is actually available in version 2.0 [OMG08a]. The SPEM 2.0 meta-model is structured into seven main meta-model packages, as depicted in Figure 2.12:

1. Its **Core** package contains those meta-model classes and abstractions, that build the base for classes in all other meta-model packages.
2. The **Managed Content** package provides language elements to manage the various textual descriptions comprising a development activity. These concepts can either be used standalone or in combination with process structure concepts.
3. The **Method Content** package is used to define fundamental methods and techniques for software development. It provides the concepts for defining life-cycle and process-independent reusable method content elements. Therefore, it provides a base of documented knowledge of software development methods, techniques, and concrete realizations of best practices.

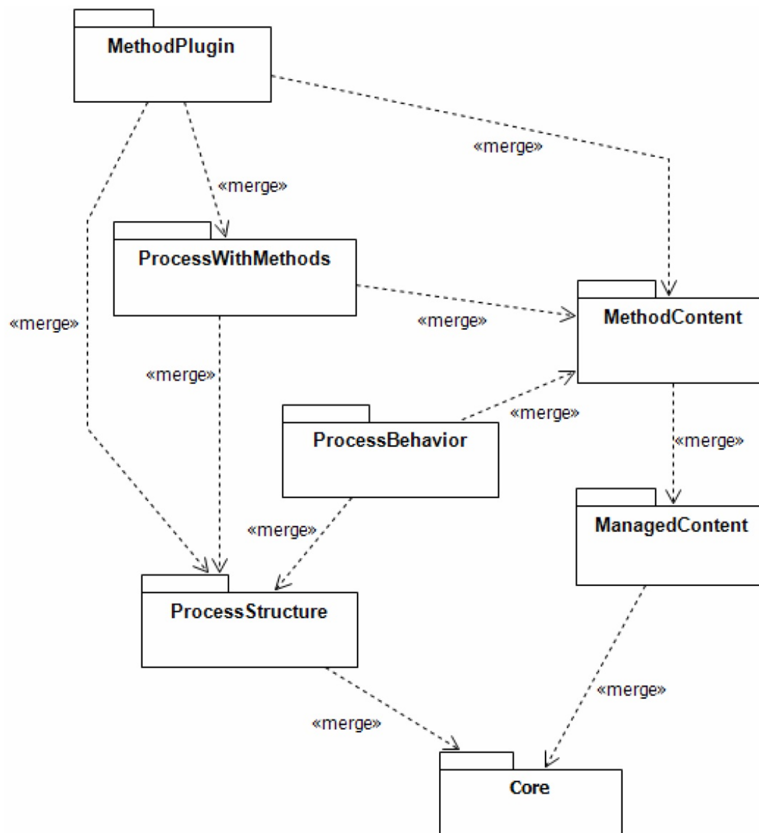


Figure 2.12: Structure of the SPEM 2.0 Meta-Model from [OMG08a]

4. The **Process Structure** package defines the base for all process models, from which the core data structure is a breakdown or decomposition of nested activities.
5. The **Process Behavior** package provides concepts to extend the static breakdown structure with externally-defined behavior models. Therefore, the meta model defines the ability for implementers to choose a generic behavior modeling approach, that best fits their needs. For example, BPMN or UML activity diagrams.
6. The **Process with Methods** package integrates processes defined with the Process Structure package with instances of the Method Content package to define new and to redefine existing processes. Processes place these methods, which were defined by the means of the Method Content package, into the context of a life-cycle model comprising, for example, phases and milestones.
7. The **Method Plug-in** package introduces concepts for designing and managing maintainable, large scale, reusable, and configurable libraries or repositories of method content and processes.

One of the most important features of SPEM is the separation between method content elements, i.e., work products, tasks, roles or guidance elements, and processes, i.e., Delivery Processes or reusable Capability Patterns. This separation, which is depicted in Figure 2.13, enables adaptivity and variability of processes, since method content, as well as, processes are reusable for various project-specific processes in different ways.

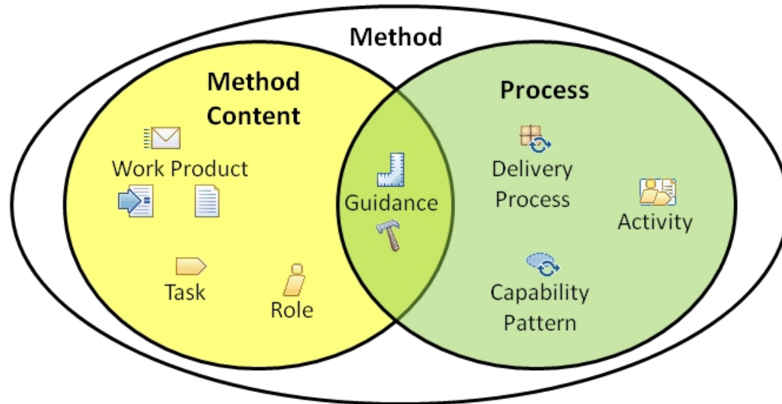


Figure 2.13: Separation of Method Content and Processes from [Hau05]

Therefore, an SPEM 2.0 usage scenario is, that organizations provide reusable process and method libraries, which can be selected and tailored to the specific needs of a required process or method. As illustrated in Figure 2.14, this is realized using an SPEM Method Configuration, which is deployed to teams for enactment.

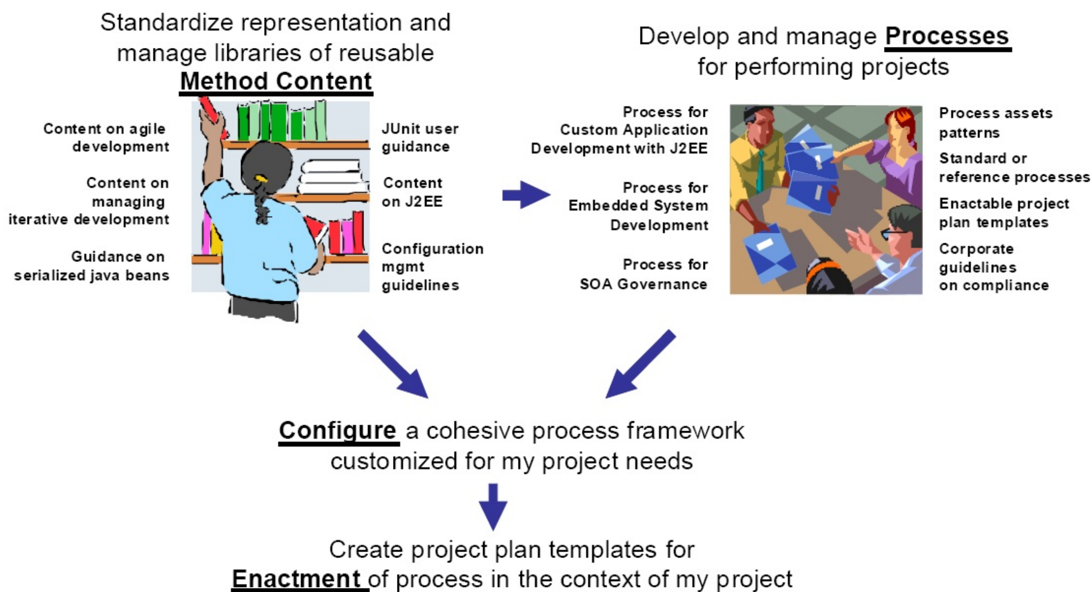


Figure 2.14: SPEM 2.0's conceptual usage framework from [OMG08a]

Although, a graphical representation, as provided by all of the above modeling approaches, is understood by familiar stakeholders well, Business Process Models benefit from a formal foundation to increase potential for analysis and to reduce the risk of ambiguities [AHW03]. For example, Petri nets [JR91] are one way to express state-based process models in a formalism, which is associated with analysis techniques to check particular characteristics of the model, such as liveness, reachability, or boundedness. Subclasses of Petri nets, such as Colored Petri Nets [Jen03] and Workflow nets [Aal98],

process algebras [Bae05], or the π-calculus [Mil99] are other formalisms, which are used as underlying basis for high level modeling languages, such as BPMN or UML activity diagrams.

2.4 Methodology Engineering

As we focus on software (development) processes, it is important to determine the characteristics of such processes and to differentiate them from other types.
The probably most important characteristic of software processes is, that while general process models are important to provide a general guidance, they are seldom followed exactly in software engineering. In [MD06], the authors explain, that developers often have different, personal views on what roles they are supposed to play. Furthermore, as contingencies and breakdowns occur in actual systems development, software engineering is critically dependent on the unique abilities of the creative people involved in those processes. Thus, although a process focus is important for obtaining coherence in the organization, the creativity of an organization's people bring processes to life. Beside the essentiality of creativity, different techniques are conceivable to achieve a particular goal, to foster the individual qualities of an engineer, or to consider the individual needs of a product under development. Furthermore, software processes are not executed straightforward, as, e.g., a simple purchase order business application. Instead, they are distinguished by iterative and/or incremental cycles, which are influenced by external and internal factors, such as change requests, identified design drawbacks, or functional adaptations.

Therefore, before introducing a discipline, which faces relevant needs, called Method Engineering, we first clarify the terminology in the field of software development processes and methods. This terminology often is confusing ([MÖ2]) due to parallel developments of different modeling notations, the strong influences from other disciplines, as well as, a multitude of standards in this area, such as SPEM, BPMN, or CMMI. In particular, the terms process, method, and methodology are used ambiguously and must be distinguished.

2.4.1 Terminology of Methods and Processes

In [BB01], the authors explain, that there are different types of processes with a distinct orientation. For example, business processes monitor business applications, such as financial, banking or administrative processes, manufacturing processes manage the production of materials, workflows define processes to be enacted by a workflow engine, or Enterprise Application Integration defines batch processes controlling the routing of data between applications.
However, independent from a concrete process type or its orientation, a process can be

defined as a set of partially ordered steps intended to reach a goal [HF92], and, in general, it can be compared to the (usually predefined or often repeated) way you relocate yourself from home to the work environment [HSR10]. Most process definitions like this are generic, but the most common consent definition is, that a **process** at least is a means, that may be used in different situations at different granularities for facilitating human understanding and communication, supporting process improvement, supporting process management, providing automated process guidance, and providing automated execution support [AMB+04]. In particular, a **software (development) process** focuses software engineering, which requires the consideration of a coherent set of policies, organizational structures, technologies, procedures, and outcomes that are needed to conceive, develop, deploy, and maintain a software product. According to [Fug00], this exploits a number of contributions and concepts:

1. Software development technology: technological support, such as tools, infrastructures, and environments, used in the process.
2. Software development methods and techniques: guidelines on how to use technology and accomplish software development activities.
3. Different stages of software development (e.g., requirements specification and development/deployment) must be shaped in such a way to properly consider the context where software is supposed to be sold and used.

Additionally, a process must be distinguished from a process description [Ost87]. A *software (development) process model* is such a description of a software process at the type level, i.e., it can can be instantiated several times. Similar to any other process type, a software process model can be descriptive, to describe the history of how a particular software system was developed, prescriptive, to define desired processes and how they should/could/might be performed, or explanatory, to provide explanations about the rationale of processes [Rol98, Sca01]. Independent from its aim, process models can be classified into four groups [Dow87, Rol98]:

- Activity-oriented models focus on the activities, which are performed in producing a product, and their ordering.
- Product-oriented process models, also focus the notion of activity but, additionally, link activities to associated products.
- Decision-oriented models focus the successive transformations of the product based on consequences of decisions. That way, decision-oriented process models, not only describe how a process proceeds, but also why it proceeds.
- Contextual models are based on a specific intention, which has to be achieved in a subjectively perceived situation. Work, that has to be done next, only depends on the current situation and the intention, which makes contextual models flexible and reactive, while it blows up your knowledge base, since any conceivable context must be covered. This, especially, makes it difficult for large processes.

In either cases, to represent a software process model, a **software process meta model** is required to provide a set of generic concepts, which ensure the genericity of the process

representations (e.g., graphical, textual) [SSRG96] on different abstraction levels, such as, e.g., the **life cycle level**, the **development process level**, or the **atomic step level** [Die12]. Most prominent members of such meta models are, e.g., OPF [OPE09], the ISO/IEC International Standard 24744 [ISO07], aka. SEMDM (Software Engineering Metamodel for Development Methodologies), or SPEM, as discussed in detail above.
Using one of these meta models, a software process model is constructed on different abstraction levels from bottom-up or from top-down ([HBO94]). While in the bottom-up approach atomic work units are aggregated into Grouping Elements (GEs), a top-down approach consecutively refines GEs into more detailed parts and work units. As our approach follows the latter strategy, the following exemplifies the top-down decomposition of a software development process.

- **Life-cycle Level**
  On the life-cycle level, a general software project is subdivided into significant time periods in a project, ending with a major management checkpoint or a set of outcomes. Typical kinds of these so-called stages are repetitive cycles (iterations), phases, and milestones [ZHSF05,OMG08a]. General collections of widely-accepted phases can be found in literature, such as [Som07]. There, Sommerville suggests the four main phases *Software Specification, Software Design & Implementation, Software Validation and Software Evolution*. As high level references, which organize the phases, that occur during a development process, so-called Life-cycle Framework Models [AMB+04], such as the *spiral model, the waterfall model, incremental/iterative development, or the V-Model*, are available on the life cycle level. In general, iterations and phases, constitute broader collections of work, that an organization must master to successfully carry out the essential work of software development. Therefore, in accordance with Clements and Northorp [CN01], we will refer to these elements as **Practice Area**s.

- **Development Process Level**
  Individual practice areas or stages, which e.g., are predefined in a life-cycle model, are refined into sub-stages and activities. While practice areas and activities, basically, are GEs, which organize a development project according to different checkpoints, durations and disciplines, often an activity is also seen as an unit of work, that relies on a specific worker, as well as, on distinct input information to produce new output [OMG08a]. However, from our point of view, such vague distinction between GEs and units of work, may lead to confusion, when talking about processes, activities, or methods, as discussed below. Therefore, in this thesis, we refer to an **activity** only as GE, which in contrast to a practice area must have a more clear purpose of creating or updating one or only a small number of artifacts. The granularity of an activity should be no more than a few hours to a few days, while phases may take longer periods in time.
  That way, practice areas and activities constitute a software process on the development process level. On this level, various standards and frameworks, aka. **Software Process Models**, such as, CMMI [cmm08], (Automotive) SPICE [Aut10], or ISO/IEC 12207 [ISO08b], are available as reference frameworks or bodies of knowledge to complement the life-cycle model with a description of relevant activities

and an associated set of outcomes.

- **Atomic Step Level**
  Activities are refined by atomic work units, aka. tasks or practices, to concrete the work that must be performed on the more detailed atomic step level. For example, an activity *software requirements engineering* may be refined into the more concrete atomic work units called *requirements elicitation, requirements documentation, and requirements validation* in order to provide developers with guidance information about how to perform the *requirements engineering* activity.
  Similar to GEs (phases, iterations, and activities), by which a particular order between practice areas and activities is defined, atomic work units are also organized in a network, which represents a non-linear sequence of actions, that structure and transform available computational objects (resources) into intermediate or finished products [Sca01]. This network, which is referred to as process likewise, can be composed using the same control flow patterns, as discussed for business processes in the last section. Contrasting the GEs, which provide an abstract temporal ordered description of *what* has to be performed, atomic work units must be further detailed in order to provide an idea about *how* to perform individual work units.

Although, the term software development process refers to a coordinated set of process activities, it often is used as synonym for **methodology**, which, originally, means "the systematic study of methods that are, can be, or have been applied within a discipline" [def13]. Additionally, in literature the terms methodology and method are mostly used synonymously as an approach to perform a software/systems development project [HSR10]. Therefore, to avoid confusions, this thesis uses the terms methodology and method synonymously.
From our point of view, a method refers to the in-depth coverage of an atomic work unit only. Thereby, we agree on that each method bases a specific way of thinking, which consists of different parts, such as guidelines, rules and heuristics with corresponding development work products and developer roles (played by humans or automated tools) [HSR10]. However, to avoid confusions, when talking about methods and processes [HBO94], in this thesis, we clearly distinguish the two concepts. Therefore, contrasting the above definition, we refer to a **method** as an atomic entity, which complements atomic work units of a process with information, that a software developer must be cognizant of: in particular, the work products, involved people, tools, and individual guidance to realize a particular way of thinking. Various methods can be combined into a contingent way of thinking in form of a complex method or a systematically structured set of methods, but we call such a combination of different methods a software development process.

The software process meta model, which we use in this thesis, looks as illustrated in Figure 2.15. The meta model organizes components of a process into **Method Components** and **Grouping Elements**. While **Grouping Elements** are used to structure the various stages and activities of a development process, single work units are realized by methods or so-called Method Chunks. A method or Method Chunk (MC), basically, con-

sists of a product part and process part [HBO94]. As depicted in the meta model, our notion of an MC refers to multiple product parts, representing relevant input and output information of an MC, and one process part, which is considered as a guideline providing rules, heuristics, and practices. A Process is defined by aggregating multiple MCs into GEs.

Therefore, we also distinguish the disciplines of Method Engineering and Process Engineering: Process Engineering (PE) provides means and techniques to describe common properties of a class of processes, that have the same nature [Rol98], and to manage abstract activities and tasks to produce project-specific life-cycle models [RPR98, MKP98], i.e., it is located on the life-cycle level and the development process level. In this context, PE relies on activities and techniques, which are similar to business process management activities, as discussed in the last section. Therefore, business process management perfectly matches the needs of PE in the software engineering domain.
Contrasting usually descriptive approaches in PE, Method Engineering (ME) attempts to provide prescriptive models [RSM95]. Thereby, it provides techniques to develop a contingent collection of means, which help to accomplish a specific task in software engineering and complements PE with relevant information about how to perform individual work units. ME focuses the construction, modification, and combination of methods. The following section details the discipline of method engineering.

2.4.2 (Situational) Method Engineering

A large number of methods was developed for information systems development, during the last decades. Amongst others, there are structured approaches, prototyping approaches, systemic approaches, object-oriented, agile approaches, or formal approaches. As methods must be well-suited to the needs of their users, i.e., developers, and, in particular, due to the necessity to change methods from one business situation to another [HSR10], Method Engineering and the more specific discipline of Situational Method Engineering have established during the last years.

According to [KW92], ME provides means and techniques to improve the usefulness of system development methods by creating an adaptation framework, whereby methods are created to match specific organizational situations. The definition of Kumar was extended in [Bri96], where Brinkkemper refers to ME as an engineering discipline to design, construct and adapt methods, techniques and tools for the development of information systems. That means, in a more comprehensive view, ME provides contingent ways of thinking in order to efficiently realize concrete products, artifacts, or other deliverables, which are required during the overall software development lifecycle [KW92, Har97b, MKP98].
Contrasting the general organizational needs focused by ME, SME focuses on the construction of methods, which are tuned to specific situations of actual development projects

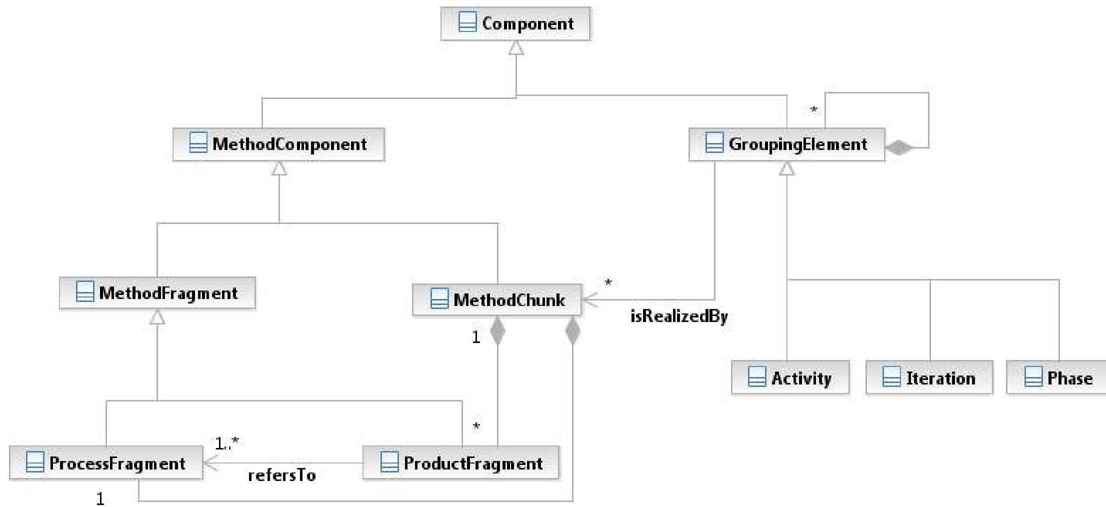


Figure 2.15: Overview: Meta model for software development processes

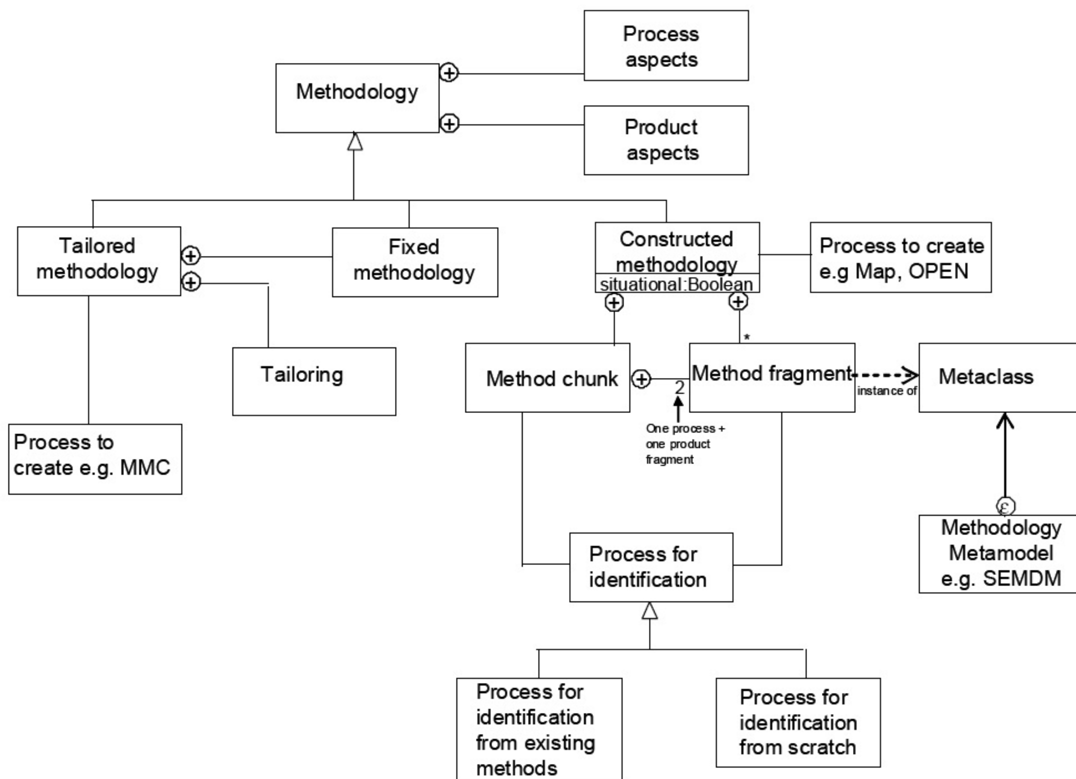


Figure 2.16: An overall high-level model of the SME approach [HSR10]

[KW92]. Some of the challenges for SME include the creation of method bases, tools to support method construction, theory and tools to support quality evaluation of the constructed method, the availability of SME tools and their interface to other tool-sets, and knowledge of what works in what situation and why [HSR10].

Figure 2.16 gives an overview about the different facets of SME. In the depicted meta model, the term methodology is the main concept, i.e., the overall software development process, which is characterized by different process and product aspects. Methodologies (or processes) either can be tailored, constructed, or fixed, whereby so-called meta-processes describe relevant tasks to achieve tailoring and construction efficiently. The figure also depicts, that in ME/ SME, each method is made up of smaller constituent parts (for example, method chunk and method fragment). To detail the various facets of SME, the following describes different research activities and relevant questions discussed in literature.

2.4.2.1 Components of a Method

SME/ME is based on the fact, that each method is composed of different smaller parts. The most common terms, which are used by many authors to describe parts of a method are: Method component, Method Chunk, or Method Fragment.

The term Method Fragment was coined by Harmsen [HBO94] as building block of a situational method. These Method Fragments (MFs) can further be classified according to the dimensions perspective, abstraction level, and layer of granularity, as discussed in [Bri96]:

Perspective: The first dimension already is mentioned by Harmsen, when he distinguishes the process and the data perspective of methods, by introducing product fragments, which are the products and sub-products to be delivered by a method, such as deliverables, milestone documents, models, diagrams, etc. , and process fragments, which represent the stages, activities and tasks to be carried out in order to produce product fragments. However, various authors ([Har97b, MDK99, BB01, BPKJ07]) have recognized the need for additional fragment types or perspectives to detail a method considering its required resources, such as roles (i.e., human-related factors) and tools (i.e., infrastructure-related factors). In this thesis, four specific perspectives become relevant to comprise a method with the following information:

- Artifact: The artifact perspective defines the product-specific part of a method. By the means of artifacts, aka. work-products, it defines, what has to be produced, modified, or used by a method. Artifacts are the tangible products of the project, which are consumed by distinct roles as **input** to produce individual **output**. Artifacts have different forms: a model, a model element, a document, source code, executable code, or the final product are examples of an artifact.

- Role: The role perspective of a method defines a human-related factor, which is responsible and capable to perform a specific unit of work. Therefore, a role defines the behavior, responsibilities, and capabilities of an individual, or a group of individuals working together as a team. One individual may have different roles in the course of the development process. Roles, for example, are: requirements engineer, software architect, timing specialist, or test manager.

- Tool: The tool (or application) perspective refers to the infrastructure-related factors of a method. It describes the capabilities of an automation unit, that supports the associated roles in performing the work defined by a method. A tool can be a fully-automated, or it can be a general purpose tool, which enables the the processing of artifacts.

- Guideline: The guideline perspective represents the process-related facet of a method. It is a set of indications on how to proceed to achieve an objective or to perform a method [RPB99]. It is a specific type of guidance, that provides additional details, rules, or recommendations on artifacts and their properties. Additionally, it can include details about best practices, to provide proven ways or strategies of doing work to achieve a goal, that has a positive impact on work product or process quality [OMG08a]. For us, a guideline embodies method knowledge to guide the developers in achieving an intention in a given situation.

Abstraction: The second dimension of MFs, which was proposed in [Bri96], is the abstraction dimension, which distinguishes the conceptual level and the technical level. While MFs on the conceptual level are descriptions of information systems development methods or part thereof, the technical dimension refers to the operational parts of a method. This distinction strongly influences our approach, that distinguishes the business level to represent the conceptual perspective and the technical level to represent operational information.

Granularity: The third dimension of an MF is determined by the granularity layer, which means the decomposition level of a method. For example, a method from a process perspective consists of stages, which are further decomposed into activities and individual steps [Bri96]. A similar decomposition, for example, can be made in the product-related perspective, where a final product is subsequently decomposed into milestones, deliverables, models, and model components. In our approach, which, particularly, focuses on the operational part of a method on technical level, decomposition is situated on the conceptual level, i.e., on business level. On that level, our approach allows for the application of different ME or PE techniques to decompose stages, activities, and products.

Different types of relationships exist to combine MFs. A general overview is given by Brinkkemper in [Bri96], where he distinguishes two general types of relationships between fragments of the same type, such as product refinements or precedence relationships between process fragments, and relationships between fragments of different

types. The latter kind of relationships is subdivided into support relationships, such as a relationship between a tool and a depending product fragment, and input/output relationships, expressing the fact that product fragments are consumed/produced by process fragments.

Contrasting Method Fragments, other authors prefer the term Method Chunk [RP96b, RPR98, MR05], aka. Method Component [WK04], which can be seen as a more meaningful combination of different MFs. Although, originally defined as a pre-determined linkage of only one process-oriented component (aka. guideline) with one product-oriented component, i.e., a one-to-one relationship [HSGP08], individual meta models, such as OPF [OPE09], SPEM, or the new ISO/IEC 24744 [ISO07], allow for more complex relationships between product and process parts, as well as, the incorporation of different MF types, such as human- or infrastructure- related ones.
This is also depicted in the meta model shown in Figure 2.17. The meta model, which was defined in [RR01a], defines a method (or process in our terminology) as complex structure, which consists of different chunks on various abstraction levels. A Chunk, i.e., an MC, appears under its synonym of guideline, which is based on one or more product parts in combination. Each guideline has an interface to describe the usage scenario of a guideline (chunk), and an intention to describe the goal of the chunk, i.e., the product to achieve. Additionally, the guideline is associated with a descriptor, which describes the reuse situation, e.g., the application domain or the supported design activity, and reuse intention, e.g., the objective which can be achieved by using the chunk in a given situation. This idea of a chunk interface inspires our idea of a Method Chunk interface, which we are introducing in chapter 3 in order to link variation points with variants in our process line. Similar to a software program, a variation point is an interface, for which a corresponding body (the variant) provides detailed information about how to achieve a chunk's intention in a process line. Moreover, the approach distinguishes three kinds of a guideline: a simple guideline, which provides advices to master an individual engineering activity in narrative form, a tactical guideline, which is used to compose different guidelines, and a strategic guideline, which is used to structure different kinds of sub-guidelines.

While the linkage of MFs in form of MCs speeds up the usage of methods by reducing the number of components, which have to be identified during configuration, the definition of predefined complex fragment combinations may lead to redundant method fragment information [HSR10]. However, regardless whether Method Fragments or Method Chunks are preferred, it is well established, that they should be stored in a method-base or repository [SIWyS93, Bri96, Har97b, RPR98, RR01b], to provide a database from which parts can be copied out for situational method construction based on specific project characteristics.

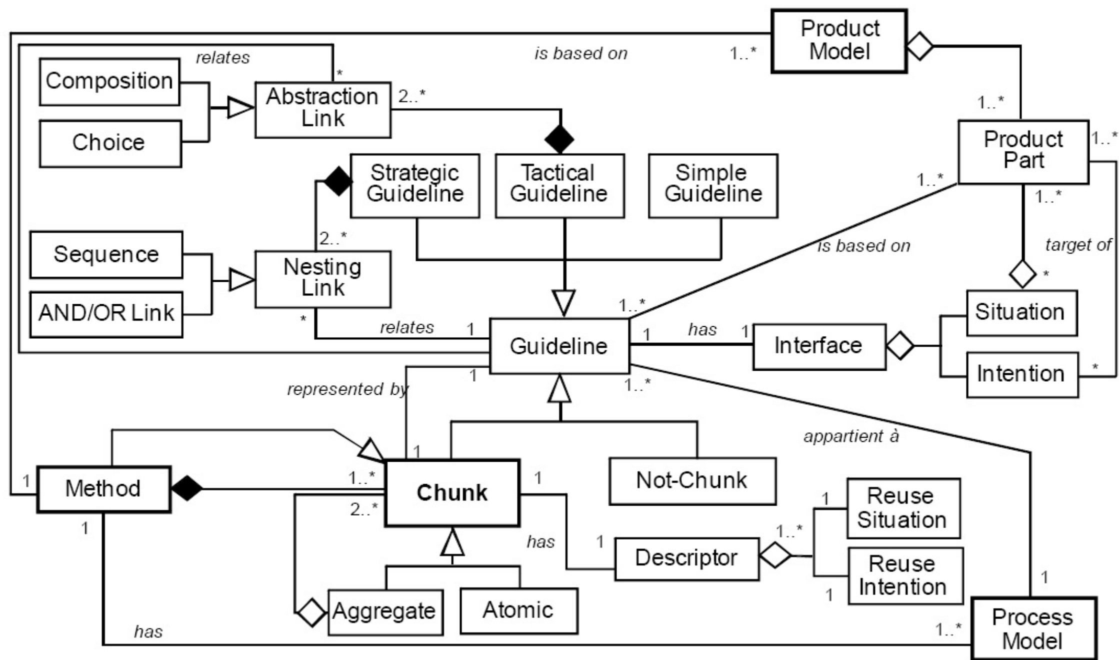


Figure 2.17: A Method meta model according to [RR01a]

2.4.2.2 Identification and Construction of Methods

As many meta models are proposed by various authors to address the structure of methods, as many authors address a meta process, which guides users in the application of these meta models [GP01, RDR03, PG07]. In [NR08], for example, they extract the following generic activities, which cover most relevant method engineering activities:

1. Method Requirements Analysis: During this activity, important features of the method under construction should be identified, before they are defined in a formal way to describe the method's needs.
2. Method Design: Based on the identified requirements, a blueprint for the method is defined, during this activity.
3. Method Implementation: During the method implementation activity, suitable methods are constructed. The result of this activity can be a set of CASE tools providing means to support the processing of a method's products, and/or process support environments to guide the developers.
4. Method Test: The final activity focuses on the verification and validation for determining whether a newly developed method realizes the predefined method requirements or not.

In contrast to that generic method engineering process, in [RR01a], they suggest different more concrete strategies to achieve efficient method engineering. The strategies are described using a so-called map [RPB99], which is a popular notation in the ME community not only to describe a meta process, but also the resulting method.

A map, in general, is a directed graph, which consists of intentions (nodes) and strategies (edges). An intention represents a task, which has to be accomplished, while a strategy suggests a distinct way to achieve a particular intention. A map consists of different so-called sections, which are defined as a triplet consisting of a source intention, a target intention and a strategy, i.e., an intention can be achieved applying different strategies. Within a map an intention is a starting point, from which one has to decide, which intention must be achieved next. Each strategy to reach a selected intention can be further decomposed into more detailed sub-maps, whereby on each level a strategy is enacted to provide advice on how to perform. For the enactment of a strategy, different guidelines are associated with the parts of a section: an Intention Selection Guideline (ISG), associated with the source intention to support the selection of an appropriate target intention; a Strategy Selection Guideline (SSG), associated with a pair of intentions to provide the set of alternative strategies to get from the source intention to the selected target intention; and an Intention Achievement Guideline (IAG), associated with a section (node pair plus strategy) to provide real advice on how to achieve the selected intention.

The chunk identification approach proposed in [RR01a] uses the map notation to describe the meta process for the definition of MCs. This meta process is illustrated in Figure 2.18 and consists of four intentions linked by different strategies to achieve these intentions. The first two intentions (Define a section and Define a guideline) allow the method engineer to restructure an existing method or to define new ones by the identification of sections and associated guidelines. The subsequent intentions (Identify a method chunk and Define a method chunk) correspond to the identification and definition of MCs from sections and guidelines. To achieve the individual intentions, i.e., to create a MC from scratch or from existing methods, various strategies were identified. While in [HSR10] a good overview about the strategies is provided, details can be found in [RR01a] and [Ral04].

Beside the identification and construction of method constituent parts, other approaches focus on the construction and tailoring of methods from identified parts stored in some method base. General kinds of construction patterns have been defined in [RP96a], where Rolland identifies general method construction patterns, such as *Identify, Describe, Construct, Define, Check and Refine*, which guide the construction of different methods. However, the most prominent method development strategies are classified, as follows:

- Ad-hoc: Ad-hoc Method development creates a new methodology from scratch. This, for example, is described in [Ral04].
- Paradigm-based: The paradigm-based method development instantiates, abstracts [Rol02] or adapts [Tol98] an existing meta-model in order to produce a methodology.
- Extension-based: Method development through reuse of an existing methodology to enhance it by new concepts and properties, whereby patterns can be used, as originally described in [DS98].

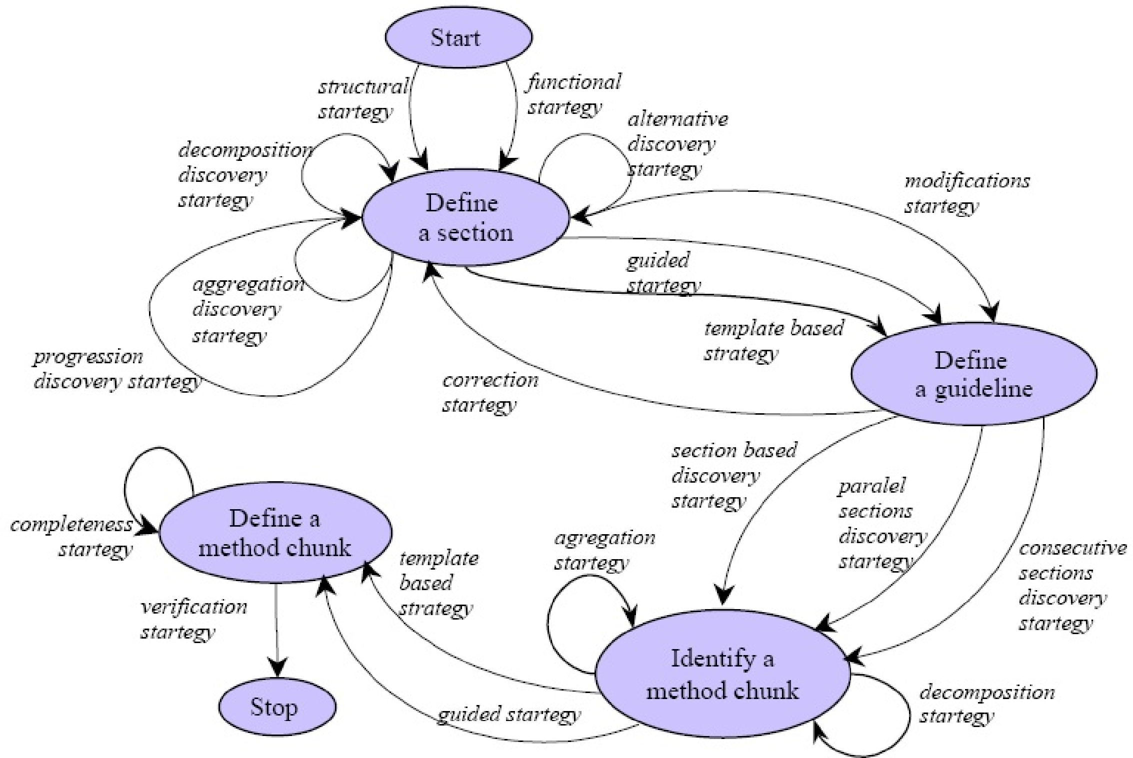


Figure 2.18: Method reengineering process model from [RR01a]

- Assembly-based: The assembly based method development uses repository stored method components to construct or enhance existing method. This strategy was originally described in [RR01b] and extended, e.g., in [KDS07].
- Architecture-based: The architecture-based method development is realized during a life-cycle, where common properties are abstracted in a method architecture, which is configured with concrete methods by situational needs. This is discussed, e.g., in [PG07].
- Method-oriented Architecture (MOA): The most recent strategy is MOA. The MOA-based method development adapts the Service-oriented Architecture (SOA) [Erl05] paradigm for the needs of method engineering by describing methods as services in parallel with service discovery principles. Details can be found in [DIKS09].

A different approach, which is provided by Niknafs et al. [Ali07], uses ontologies for an assembly-based approach to support Method Engineering. They propose a semantic data-model, which provides concepts to define semantics of method fragments, the associations between them, and the method base. By describing Method Fragment as complete and unambiguous as possible, semantic search of method fragments, checking semantic consistency of fragments, and other checks, such as semantic completeness, semantic conformity with the meta-models of product and process models is enabled.

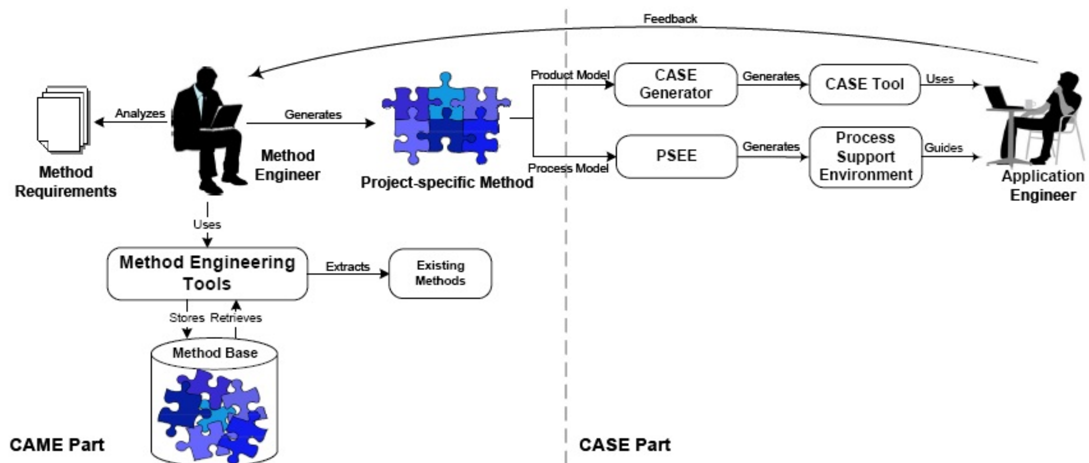


Figure 2.19: General architecture of CAME environments from [NR08]

2.4.3 Tool Support for Methodology Engineering

To provide software support to ME activities, the term Computer-aided Method Engineering (CAME) tool/environment was coined. In general, CAME aims at supporting the development of methods, similar to the so-called CASE tools, which support the development of information systems. In [Rol97], Rolland suggests the following functionality of a CAME tool:

- Definition and evaluation of contingency rules and factors or descriptors to enable an appropriate choice of the method components.
- Storage of method components, method construction knowledge, past experience, heuristics, etc. in a repository called the method base.
- Retrieval of the contents of the method base by the means of a query language for accessing the contents of the method base.
- Composition of method components by providing the knowledge permitting the development of a new method.
- Validation and verification of the constructed method. The CAME tool should not only support the selection and assembly tasks, but also check the resulting method. The tool, therefore, should incorporate guidelines to ensure the correctness of the method.
- Adaptation facilities for modification of the contents of the method base as a result of the experience gained.
- Support and guidance of the method engineering task.

The listed capabilities show, that the software process domain not only encompasses the domain of method or process modeling, but also an enactment and a performance domain [Dow93]. While the modeling domain encompasses the definition and maintenance

of software process models or methods, the performance domain concerns current tasks, that are performed by human or non-human agents during a performed software process. These two domains are connected through the process enactment domain, which supports and controls the process performance domain through process models.
Therefore, in order to enable enactment and performance capabilities, an CAME environment's capabilities, normally, are complemented by a so-called application engineering part, as, for example, proposed in [Rol97] or [NR08]. An architectural overview about such a more comprehensive CAME environment is given in Figure 2.19. The figure shows, that a complete CAME environment can be subdivided into two parts: a core part, which provides mere method engineering functionality, and an CASE part, which offers means for application engineering, i.e., for the generation of CASE tools and process support.
Once a method is obtained from the CAME core part, it will be fed as input to the CASE part, which generates a project-specific CASE tool from the product part of a method. In parallel, process support is generated based on the process part of methods. Depending on the CASE part's focus, i.e., the product model or the process model of a method, CAME environments are classified into product-oriented or process-oriented environments. Most of the existing CAME environments fit into the first class, due to their emphasis on modeling a method's product part and the generation of various CASE tools to enable the construction and modification of method-specific products [NR08]. These product-oriented environments use the product-related parts, which result from the method engineering activity, and their associated product meta-models, to derive method-specific user interfaces, i.e., editors or CASE tools, which provide relevant capabilities only.

The process-oriented environments go by different names: Process Support Systems (PSS), Process Sensitive Environments (PSE), Process Centered Environments (PCE), or Process-Centered Software Engineering Environments (PSEE) are some of the alternative terms used in the literature [MR12]. Although, many requirements for such an environment are discussed in literature [ACF97, Zam01, Gru02, DO05, MR12], the most general and widely accepted set of requirements, which should be fulfilled, are summarized in [ADOV02]. Here, Arbaoui et al. state, that an PSEE at least provides the following functionality:

- It should support enactment to enable the automating of human and non-human activities.
- It should support the dynamic management of software process activities. Therefore, in [DF94] they distinguish four different styles: passive guidance, where information is provided to developers only on request; active guidance, where the process model specifies when and how information is provided; process enforcement, where agent have to perform process parts in a specific way; and process automation, where process parts are under the control of the enactment mechanism completely.
- It should support the software process distribution to enable communication, co-

ordination, cooperation and negotiation between both user performers and automated process elements.

- It should support the software process evolution to manage the redesign and adjustment of processes.

Furthermore, Matinnejad et al. expose, that PSEE Technology should be distinguished from related technologies, such as Workflow Management (WFM) and Computer-Supported Cooperative Work (CSCW). While PSEEs are software systems, that assist in the modeling and automation through enactment of software development processes [ADOV02], WFM systems are mainly concerned with modeling and automation of business workflows and industrial processes, and CSCW systems merely provide assistance for groups of developers in collaborating and coordinating their activities. In particular, according to [Gru02], an PSEE-enabled CAME environment provides essential advantages, such as:

- the enforcement of consistency between documents,
- the guidance of software developers,
- the knowledge of the state of the software process, and
- the automation of process parts (e.g., in configuration assembly, testing).

In literature, many prototypes have been proposed to illustrate the concepts of flexible CAME environments. Due to their particular relevance to this thesis, where we introduce a hybrid CAME environment, which considers process and product facets likewise, we provide a comprehensive overview and discussion about related work in chapter 7.

2.5 Software Product Line Engineering

In the beginning of the software area, a software product was small and each product variant either included all possible features one might ever need, or it was developed from scratch for individual customers, which made software products rather expensive. Due to the increasing complexity of modern software and the increasing diversification of software products induced by individual customer needs, a new approach was required very quickly. Therefore, inspired by the notion of a production line, as initially invented by Henry Ford to enable more cheap production for a mass market in the automotive market, the term product line was established in the software domain to achieve various benefits. While a detailed overview about benefits and efforts, which arise from product line engineering, is given in [CN01], Pohl et al. summarize the most essential benefits, as follows [PBV05]:

- **Reduction of Development Costs** through reusing of predefined artifacts from an asset base
- **Enhancement of Quality** through more extensive quality assurance of reusable artifacts

- **Reduction of Time to Market** through assembly-based combination of new products from an asset base in contrast to individual solution
- **Reduction of Maintenance Effort** through the propagation of single artifact changes to the overall product portfolio
- **Coping with Evolution** through the influences of innovative improvements in the asset base on all products
- **Coping with Complexity** through systematic management of software products and reuse of common product parts
- **Improving Cost Estimation** through reduced risks in asset assessment
- **Benefits for the Customers** through reduced prices for consistent quality and a stable look & feel

According to Clements and Northorp [CN01], a **product line** or product family is defined as a set of software-intensive systems sharing a common, managed set of features, that satisfy the specific needs of a particular market segment or mission and that are developed from a common set of core assets, aka. as platform, in a prescribed way. Thereby, the term **core assets** is a generic term, which includes various artifacts relevant for product development, such as requirements, design documents, software components, tests, budgets, schedules, or process descriptions. The probably most important core asset, though, is a **reference architecture**, aka. domain architecture, which determines the structure and the texture of a specific application, aka. product family member, in the software product line [PBV05]. It provides common rules guiding the design, realization, and the combination of core assets to form product family members.

To efficiently achieve benefits as mentioned above, various initiatives and projects, such as the AMPLE Project, the Cafe Project, the Families Project, or the SEI Framework for Software Product Line Practice were conducted during the last decades, to further **software product line engineering**, which is a paradigm to develop software applications using platforms and mass customization, i.e., the large-scale production of goods tailored to individual customers' needs [Dav87, PBV05]. In general, this paradigm provides high reusability through the management of commonalities and variabilities of core assets in a family of software systems in a common way. In [PBV05], this kind of management is referred to as the concept of managed variability.

An essential vehicle in software product line engineering is the development process, which is commonly separated into two parts [WL99, Van02, PBV05]: a *Domain Engineering* part (aka. core asset development [CN01]), which concerns the development for reuse; and an *Application Engineering* part (aka. product development [CN01]), which concerns development with reuse. Additionally, a cross-cutting part can be identified to support management activities considering technical and organizational concerns of a product line [CN01].

2.5.1 Variability Engineering

As already mentioned, a software product line is affected with the management of variabilities in a family of software products. Contrasting general variability, which can be defined as the ability to change or customize a system [VBS01], software variability refers to the ability of a software system or artifact to be efficiently extended, changed, customized or configured for use in a particular context [SVB05]. This may concern, for example, features, processes, data, policies, user interfaces, system interfaces, or various quality attributes of a software system. In [PBV05], they, additionally, distinguish between variability in time (cf. [EBLSp10]) and variability in space, as well as, between external and internal variability. While the latter refers to variability, which either is visible (external) or invisible (internal) to customers, the former refers to different versions of an artifact, which are valid at different times (time), or to different shapes of an artifact (space) at the same time.

2.5.1.1 General Variability Model

In general, variability can be defined either as an integral part of development artifacts or in a separate variability model [PBV05, MP07]. Although, various approaches exist, which integrate variability into development artifacts directly, such as use case models, feature models, message sequence diagrams, or class diagrams [VBS01, KLD02, BFG+02, BHP03], this causes different drawbacks concerning, e.g., consistency relationships between the various variable elements spread across various artifacts. Therefore, to overcome these and other drawbacks, as discussed in literature (cf. [GB02, MA02, BGL+04, BLP04]), some approaches propose the notion of a comprehensive variability model, which stores variability information in a separate model. Such a so-called orthogonal variability model, as, e.g., presented by Pohl et al. in [PBV05], defines the variability of a software product line by relating the defined variability to other software development artifacts. The meta model presented by Pohl et al. specifies the main concepts of an orthogonal variability model, as depicted in Figure 2.20.

The main concepts of the model are *variation points* and *variants*. While a variation point answers the question of what does vary, a variant answers the question of how does an element vary and why. A variation point either can be internal or external in order to indicate the relevance of variability to internal or external stakeholders, as discussed above. In order to relate variable elements with potential realizations, variation points must be related to at least one variant.
This relationship, or variability dependency, is specialized differently: either as mandatory variability dependency, which defines that a variant must be selected for an application if and only if the associated variation point was selected before, or as optional variability dependency, which states, that a variant can (but does not need to) be a part of a product family member. Additionally, to define the minimum and the maximum number of optional variants to be selected from a given group of variants, the meta model provides the alternative choice class, which groups a set of optional variants of the same variation point and defines the range for the number of optional variants, which have to

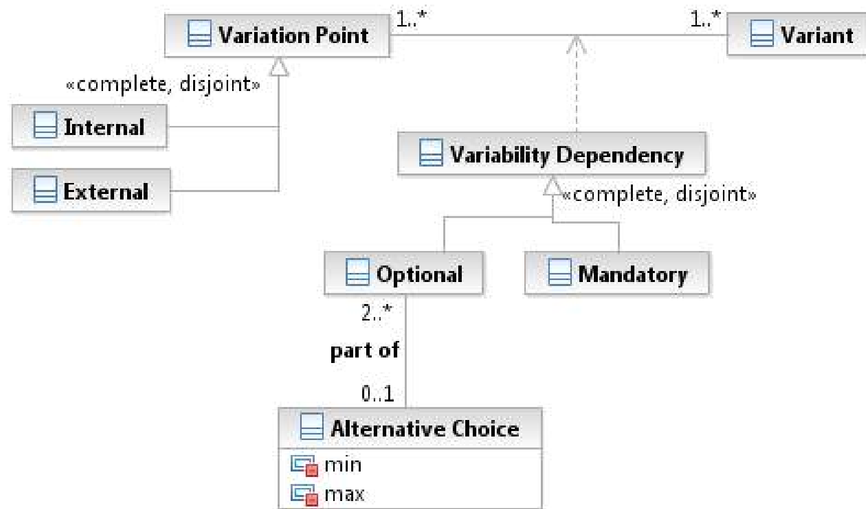


Figure 2.20: Variation point, variant, and the variability dependency in the variability meta model according to [PBV05]

be selected from this group.
To additionally consider particular dependencies between variants and variation points, the variability model can be complemented by so-called variability constraints. These constraints can be defined between variants, variation points, or between a variant and a variation point to **exclude** or **require** a dependent element based on the selection of another. For example, the selection of a variant V_1 may exclude/include a variation point VP_1, a variant V_1 may exclude/include a second variant V_2, or a variation point VP_1 may exclude/include a second variation point VP_2. That way, variation point constraints, variant constraints, and variant to variation point constraints are defined.

To relate the variability defined in the variability model to software artifacts specified in other models, textual documents, and code, variants in the variability model must be related with software artifacts, such as requirements, design, realization, and test. This can be made at different levels of granularity by introducing a dependency between variation point or variants and a concrete representation of a development artifact.

The moment, when variability is resolved to derive a product family member, often is called the binding time of the variability. Different binding times or configuration mechanisms are conceivable to bind variants before, during, or after an individual process step [PBV05]:

- Before Compilation: This mechanisms is applied to resolve variability before source code is compiled by using, for example, code generation techniques and parameterization, aspect-oriented programming, model-driven approaches, or software factories.
- At Compile Time: This mechanism is based on the used compiler functionality itself

and uses pre-compiler makros or conditional compilation statements.

- At Link Time: A resolution at link time is based on sequences of compilations and linkages activities. Depending on the situational parameters, different sets of compilations and linkages are performed.
- At Load Time: When several executables and dynamic link libraries are combined into one system, a configuration file is used to locate and initiate all files that should be loaded.
- At Run-Time: Runtime variability concerns the target system, which hosts a central registry, in which a component registers individual information about interfaces. By querying for that information, components interaction can be defined variably.

2.5.1.2 Feature Modeling

The described orthogonal variability model, basically, conforms to the notion of feature diagrams, as initially developed by Kang [KCH+90] as means to capture commonalities and variabilities at the requirements level. Due to their particular relevance to this thesis, we discuss this formalism in more detail in the following.

Feature models are one of the most commonly employed modeling and representation techniques in software product line engineering, that are used to capture both variability and commonality. They provide a tree-like structure, whose root feature represents a domain application and other nodes represent the features of products of the domain. Basically, a feature model allows different types of features and various relationships between them, from which the most relevant ones are detailed in the following:
There are mandatory features, which must be included in the description of its parent feature, and optional features, which may or may not be included in its parent description. Additionally, alternative feature groups are used, if one and only one of the features from the feature group can be included in the parent description. In contrast, an Or-feature group is used, if one or more features from the feature group can be included in the description of the parent feature. Similar to the above orthogonal variability model, feature models define cross-tree dependencies between features referred to as integrity constraints. Two widely-used integrity constraints are: *includes* – the presence of a given (set of) feature(s) requires the inclusion of another (set of) feature(s) ; and *excludes* – the presence of a given (set of) feature(s) requires the exclusion of another (set of) feature(s).

Furthermore, there are various alternative approaches for feature diagrams available, which extend the initial semantics defined in the context of Kang's FODA (Feature Oriented Domain Analysis) [KCH+90] method. In [KKL+98], they extended the scope of feature diagrams from requirements engineering to software design by applying features on different abstraction layers and by introducing relationships for generalization or specialization. The FeatureRSEB approach [GFD98], is a combination of FODA and a use-case driven reuse process, where variability is captured by structuring use cases and

object models with variation points and variants. Van Gurp et al. [VBS01] specialized feature diagrams with regard to binding times and external features to refer to technical possibilities offered by the target platform of the system. The Generative Programming paradigm was adapted by Czarnecki et al. [CE00,CHE04] to provide staged configuration and distinguishing between group and feature cardinalities. Furthermore, to provide a high level view of a product family, in [EBB05], they combined feature diagrams with use case diagrams, and, in [BAGS10], they extended feature diagrams with capabilities for capturing business oriented requirements or preferences in the form of fuzzy linguistic variables. Beside these approaches, a multitude of different approaches can be found in literature: [VK02], [Rie03] [CHE05], [Bat05], or [BTRC05].

2.5.2 SPLE Development Process

The software product line engineering paradigm separates two processes: Domain engineering, which is the process phase in which the commonality and the variability of the product line are defined and realized; and application engineering, which is the process phase in which the applications of the product line are built by reusing domain artifacts and exploiting the product line variability [PBV05]. Both phases consist of different subprocesses, which can be performed in appropriate order to match organizational needs. Thereby, product line engineering activities must be aligned with already existing organizational business processes. Although, many reference frameworks are proposed for software product line engineering in literature [BFK+99,CN01,KLD02,Van02,KKSB06], some activities are essential to it. Therefore, by following the framework introduced in [PBV05], we discuss the most common activities in software product line engineering. That framework is illustrated in Figure 2.21 and distinguishes the two main phases domain engineering and application engineering. The figure shows the distinct activities, which must be performed in respective phases, as well as, the separation between domain artifacts and application artifacts, which both are discussed in the following.

2.5.2.1 Domain Engineering

The upper part of Figure 2.21 depicts the Domain Engineering sub-process or phase of the software product line engineering process. During this phase, basically, a family reference model is designed and implemented by identifying variability and commonalities in the domain of interest. Therefore, appropriate and reusable artifacts, that accomplish the desired variability, must be defined and subsequently constructed as part of an asset base. Domain engineering consists of five sub-activities and associated outcomes, which go by different names or compositions in various approaches.

Product Management

A product management activity deals with economic concerns in a product line. It analyzes an organization's strategy and defines the product scope for a company or business

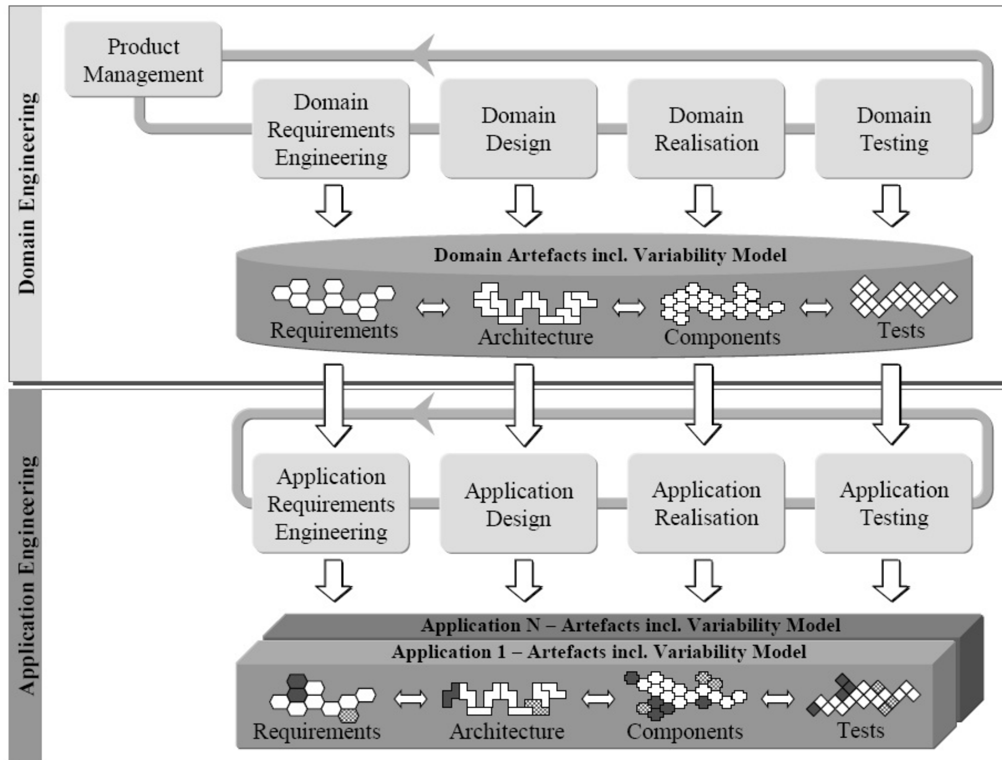


Figure 2.21: The software product line engineering framework from [PBV05]

unit. Therefore it uses scoping techniques (cf. [CN01, LKL10]) to define the boundaries of the product line. The output of that activity may be a product road map, which defines variability in features provided in existing and future products.

Domain Requirements Engineering

A domain requirements engineering activity deals with general tasks, which must be accomplished to elicit and document common and variable requirements and features in the product line. Therefore, it uses textual or model-based techniques, such as feature modeling, scenarios/use cases, state machines, or class diagrams for capturing the functional, structural, and behavioral domain requirements. In each formalism, variability must be documented in an unambiguous and systematic way. Therefore, various extensions to existing approaches were proposed in literature to integrate variability (cf. [KLD02, BHP03, PBV05]). In contrast, the aforementioned orthogonal variability model can be used for that purpose, likewise.

Domain Design

Based on identified requirements, a domain design phase deals with the technical definition of a reference architecture, which provides the common structure for all product line applications. For architecture design, various views, which concern, for example, the decomposition of processes, structural entities, or code, are used to satisfy individual stakeholder needs. Any applied view or representation must provide adequate variability mechanisms or it must be associated with the orthogonal variability accordingly to

provide a viable reference architecture, aka. domain architecture.

Domain Realization

A domain realization phase uses the reference architecture to derive a detailed design and an implementation of all identified architectural parts. This, in particular, concerns interfaces, components, algorithms, protocols, resources and their variabilities. The output of this activity is a set of design and implementation assets documenting reusable software components in the asset base.

Domain Testing

During a domain testing phase, test artifacts are created to validate and verify reusable components and to reduce the efforts for application testing. These test artifacts, at least, should contain test plans, test specifications, and test results, which cover variable scenarios.

2.5.2.2 Application Engineering

The lower part of Figure 2.21 depicts the application engineering sub-process or phase, that captures the requirements of a target application and derives applications from the reference architecture based on identified requirements. Thereby, an as high as possible degree of reuse of the domain assets should be achieved by exploiting the commonality and the variability of the software product line and binding the variable components according to the application needs from requirements to test cases. Application engineering consists of four sub-activities and associated outcomes, as discussed in the following.

Application Requirements Engineering

The application requirements engineering activity reuses the results from the the domain requirements engineering activity to define the application requirements artifacts, which serve as a basis for application design. By analyzing the differences between application requirements and domain requirements, reusable platform assets and additional requirements for the new application are identified. By determining the gap between platform requirements and aimed product, additional adaptation efforts and their impacts are estimated. The application requirements activity is essential for the communication with internal and external stakeholders, which are interested in an application and its individual features.

Application Design

An application design activity uses the application requirements to derive the application architecture, wherein the required parts of the reference architecture are selected and application-specific adaptations are incorporated. This activity requires to take care of a consistent selection of variants and variation points, which do not influence each other negatively.

Application Realization

During the application realization activity, the desired product or application is created. Therefore, reusable software components are selected from the asset repository and application-specific artifacts are created separately. Finally, a running application together with all requirement and design artifacts is created.

Application Testing

An application testing activity verifies and validates a realized application against its specification. Therefore, the test artifacts, which are provided by domain testing activity, are used in combination with application specific tests.

2.5.3 Tool Support for Software Product Line Engineering

The operation of product line engineering not only requires the introduction of new activities, but also must be incorporated with existing organizational standard processes regardless of whether development, management, or support. The transition process to change individual solution development to software product line development requires time, budget, and expert knowledge, before return on investment can be expected. However, even if product line engineering is established, new challenges must be mastered, as discussed, e.g., in [BFG+02,DSB04]. In particular, due to the high complexity of variation points and variants spread across different abstraction levels, avoidance of human errors, asset management, or consistent product derivation, must be supported by automation and tool support. In [GS07], indeed, they propose an approach for automating individual software product line engineering activities and product derivation, but, in general, comprehensive tool support for managing variability across all development artifacts is very weak [PBV05]. Therefore, it is one of the key challenges for future research.

3 Software Process Line Engineering

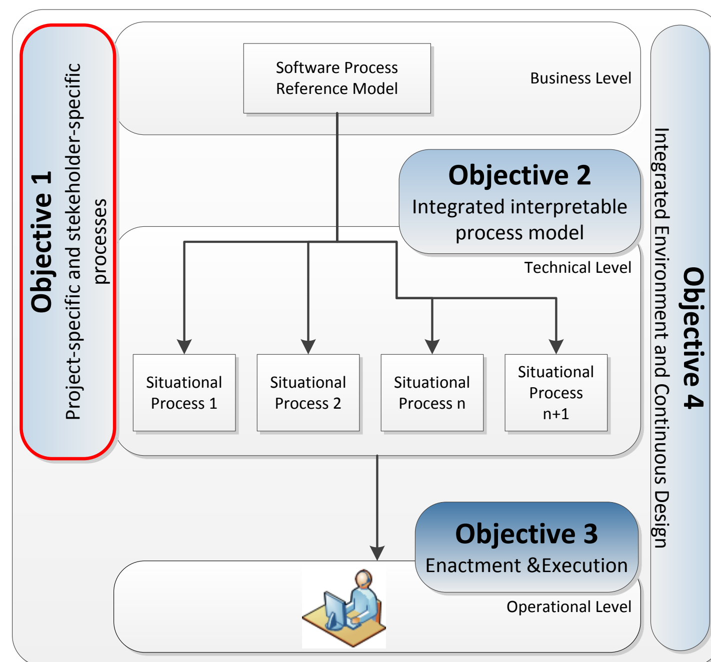


Figure 3.1: Objectives overview

Organizations are required to effectively manage their processes. Therefore, Process Engineering and Method Engineering are popular techniques to design and customize processes and associated information in form of methods, techniques, guidelines, or best practices. However, there is no integrated technique, which considers both, the variability of the process life-cycle and the situational needs of associated information. In particular, there is no approach, which considers the variability of executable development processes, at the same time. Therefore, since execution of development processes (one main objective of this thesis, which is discussed in subsequent chapters) increases conventional process design efforts to a considerable degree, an effective management of process information and their variabilities is needed to minimize these efforts. Therefore, this chapter focuses on *objective 1*, as illustrated in Figure 3.1 and sets up the necessary prerequisites to enable effective design, reuse, and configuration of executable development processes.

3.1 Motivation

Today's process variability techniques support prescriptive adaption of business project plans or other static process models and methods. However, there is no technique, which considers structural variability of a process life-cycle in parallel with behavioral variability of contained methods, which are more affected by qualitative and situational development concerns. Popular PDLs, such as SPEM or BPMN, do not provide mechanisms to support variability, which go beyond manual adaption or re-creation. Similar to other approaches, SPEM's process variability is based on reusable process patterns and *variability elements*, which support variability types, such as *replacement*, *extension*, *contribution*, *extension-contribution* [OMG08a]. Therefore, reuse, extension, and adaption of already available GEs (cf. Figure 3.4) are more enabled, than an effective support for managed variability, as known from Product Line Engineering (PLE). However, well-established product line engineering techniques can be combined with situational method engineering techniques (cf. Section 2.4) to manage a family of processes and methods enabling more than "paperware" capabilities. Therefore, the process variability management must consider structure-driven, behavior-driven, and resource-driven dimensions, as discussed in the following:

3.1.1 Structure-driven Dimension

The structure-driven dimension concerns the variable composition of processes and methods from smaller building blocks.

Process Structure Variability

Structural process variability mainly concerns structural decisions on necessary or dispensable process activities and their general order. The variability mechanism must consider, that multiple stakeholders, such as project partners, regulations, standards, such as ISO26262 [ISO10] or RTCA DO-178B [Rad92], and particular project situations influence the process structure, i.e., practice areas, activities, as well as, their logical or temporal order.

Method Structure Variability

The basic structure of methods is defined by the means of an MC, as described in Section 2.4. An MC is composed of fragments, such as input/ output information, responsible roles, and applications. Method structure variability concerns variability of MCs and a situational combination of different fragments, as addressed by a multitude of ME techniques.

3.1.2 Behavior-driven Dimension

The behavior-driven dimension concerns the variable realization of processes and methods at runtime, i.e., on operational level.

Process Behavior Variability

Process behavior variability refers to the different characteristics of a process at runtime. It concerns unforeseeable environmental effects, such as deferred project schedules, change requests, or other underestimated project risks, which influence the performance of a process. Such effects strongly influence the decisions on, e.g., the number of iterations, the duration of development phases, or the functionality of one release. The effects cannot be anticipated and must be handled at process runtime. Therefore, appropriate monitoring information must be provided to support humans in making correct decisions. For example, process mining techniques help to analyze such decisions and to incorporate this kind of knowledge into future processes.

Method Behavior Variability

The structure of an MC is realized by different ways of thinking, which we refer to as the technical realization of an MC. By the means of this separation, the structural characteristics of an MC are complemented by computer-interpretable information enabling the performance of an individual method at runtime. Similar to the structure-driven variability of an MC, which mainly concerns the composition of MFs on business level, fragments and MCs vary on technical level, as follows:

- Artifacts: On technical level, artifact-centric MFs differ in their data format, the concrete representation format, the degree of detail, or quality aspects.
- Roles: As the execution of a development activity requires different skills, responsibilities, and access rights, abstract role descriptions are refined by computer-interpretable information, on technical level depending on organizational rules and IT platforms.
- Applications: Applications differ in provided functionality, used data, and exchange formats, which must be detailed on technical level for the situation at hand.
- Guidelines: A method's behavior is strongly influenced by used guidelines, such as naming conventions, product validation rules, or quality design patterns. Although, guidelines are documented on business level informally, for automated evaluation and support, guidelines have to be detailed on technical level. These guidelines depend on the project situation at hand, which concerns IT platforms, customer relations, project schedules, and company specific standards.

3.1.3 Resource-driven Dimension

Concrete development projects depend on factors, which concern environmental effects and available resources. The factors influence the structural and behavioral dimensions of methods and processes, equally. Affected resources, for example, are:

- Timing resources: depending on contractual obligations, project timing restrictions vary from project to project. This also influences the time, which can be spent for individual development activities.

- Quantitative human resources: depending on the importance of the project and the actual company specific situation, the number of employees differs from project to project.
- Qualitative human resources: the execution of development activities depends on developer skills. As each method requires individual knowledge, the developers's knowledge determines the feasibility of activities.
- Budget resources: financial restrictions constrain the man-months and sometimes the infrastructure used during the project. Consequently, the feasibility of development activities depends on the budget.
- Infrastructure resources: especially the availability of tooling to support development or project management activities influence the process and feasibility of activities.

As the individual characteristics of a process and associated development activities depend on project resources, this kind of variability has to be factored into the process variability management, as well.

3.2 Overview: Software Process Line Engineering

Our approach, as discussed in subsequent sections, faces above dimensions in a model-driven architecture. The structure-driven dimension of methods and processes is faced by the proposed Software Process Line Engineering (SPLE) approach, where we set up a variable reference process, which is configured for the situation at hand using ME techniques, product line techniques, and Artificial Intelligence (AI) planning. Especially, method behavior variability is faced by the introduction of an additional design level and a repository, which consists of a variety of method variants (each realizing an individual behavior), which are selected based on an annotated usage scenario and situational needs. In contrast, process behavior variability is discussed in subsequent chapters due to missing runtime information at process design time. Finally, the resource-driven dimension is considered, when situational processes are derived from the process line. Herby, we validate situational process family members with respect to their feasibility.

Typical (Situational) ME (cf. Section 2.4) starts by analyzing situational requirements, e.g., through goals, and refines them to high-level methods, which are refined to more detailed methods. Afterwards, methods are integrated with processes, which are tailored by using similar techniques. As method engineering does consider the variability of the process life-cycle or the configuration of situational processes with different methods insufficiently, we distinguish between structural variability and behavioral variability.
While structural variability concerns the composition of processes and methods from smaller building blocks, behavioral variability concerns the realization of methods. In general, structural variability is handled on an abstract business-oriented design level by applying general ME and product line engineering techniques. Additionally, to face

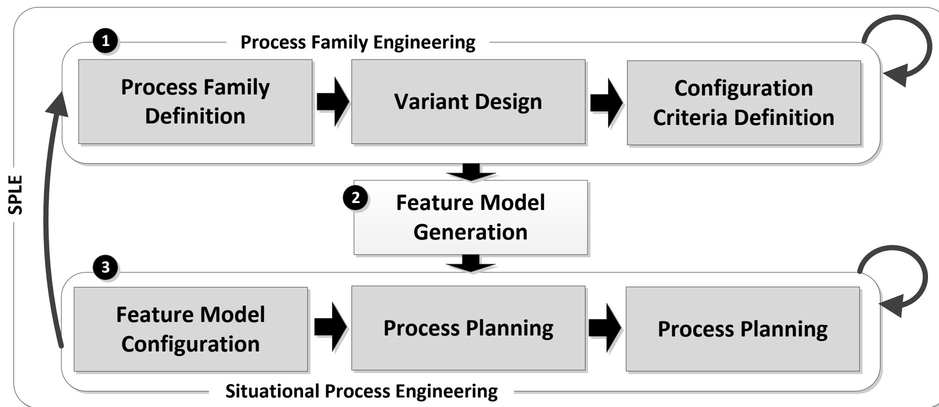


Figure 3.2: Overview: Software Process Line Engineering

behavioral variability and project-specific characteristics, a situational process is configured by combining structural information from a business level with behavioral information, which is defined on a technical design level. Thus, situational characteristics, which describe the usage scenario of individual features of a process, and available resources provide the basis for interconnecting the two levels and finding an optimal configuration.

As illustrated in Figure 3.2, our proposed approach encompasses three main processes namely 1) Process Family Engineering, 2) Feature Model Generation, and 3) Situational Process Engineering. During the Process Family Engineering phase (Section 3.3), we start on an abstract business-oriented level without technical details, by which commonalities and variabilities of (domain-specific) development processes are captured to define a Reference Process (RP), which is composed of GEs and so-called MC interfaces to face structural variability. The reference process is basis for project management activities and defines the static workflow of development processes represented by process modeling techniques, such as SPEM, BPMN, or JWT. Thereby, it includes Variation Points (VPs), which are variable places in design, that have to be bound with situational variants. Variants realize MC interfaces as concrete MCs, which are detailed by technical details for a particular application scenario, i.e., a situation in which a variant should be selected, using situational characteristics. We employ feature models for encapsulating the knowledge of the reference process and also for visualization, which caters combined view of VPs and variants. During the Feature Model Generation phase, feature models are generated as an intermediate model to represent the configuration space of a process family (Section 3.4). Next, during the Situational Process Engineering phase, (Section 3.5), Hierarchical Task Network (HTN) planning, an AI planning technique, is leveraged to configure the feature model, i.e., to bind variabilities, automatically. Therefore, appropriate technical variants (MCs) are selected from the variant repository by analyzing their annotated characteristics with regard to available resources within the respective organization. Afterwards, selected variants are bound to VPs of the reference process to realize a concrete behavior. Finally, the generated process is validated regarding structural or resource-driven hazards, before platform-specific code is derived for deployment.

The work, which is presented in this section, was developed in collaboration with our partners from the Athabasca University and the Simon Fraser University in Canada. While some ideas also were seized in a master thesis [Sol12], other ideas are discussed in [AHM+13], which is under review at the time this thesis was printed.

3.3 Process Family Engineering

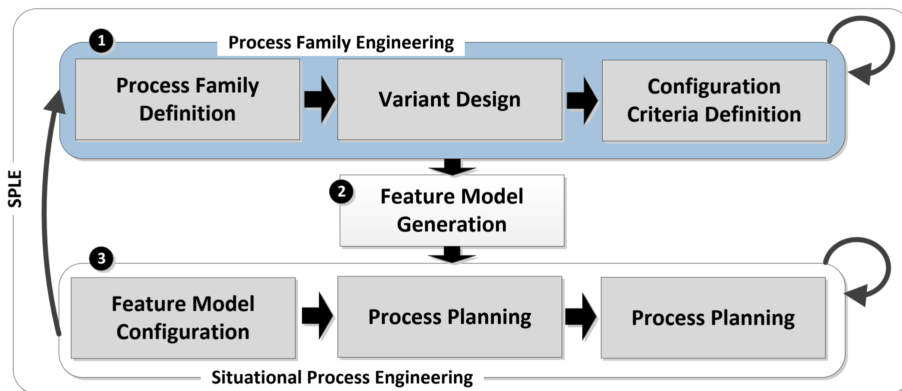


Figure 3.3: Software Process Line Engineering: Process Family Engineering

Process family engineering analyzes and sets up a process family, from which a distinct situational member is derived to enable an individual project. Similar to software product families, for software process families there are also various information types, which distinguish members of the family from each other. Basically, product family members differ in their provided functionality, which a product can have or not. Additionally, software products, which are derived from one product family, may also differ in control-flows, used data formats, dedicated access rights and user interfaces, or individual non-functional properties of functional features or the complete product.
For software process families, similar distinctions can be made: Primarily, process family members differ in functional characteristics, which may represent individual development concerns, such as Requirements Engineering (RE), functional design modeling, or testing. In addition, processes and contained methods differ in control-flows, i.e., the sequence of process actions, in data formats, i.e., data structures of processed artifacts, in access rights, i.e., process responsibilities, in user interfaces, i.e., supportive tools, and in non-functional properties of functional features, i.e., qualitative characteristics of methods, which are used during a process. Table 3.1 contrasts the distinguishing features of products and processes.

To enable these types of process variability, we basically distinguish two levels: the level of the abstract process design (business level) and the detailed method design (technical level). On both levels, relevant process line assets, as illustrated in the meta model

shown in Figure 3.4, are identified and implemented. The abstract process design level (Section 3.3.1) defines a variable basic structure of the process life-cycle and contained methods. Therefore, a set of reusable GEs is defined and used to set up a process skeleton, the RP. The RP defines the structural boundaries of the process family. It is hierarchically structured and consists of common and variable sets of abstract practice areas, activities, and methods, which are used as VPs, i.e., as proxy objects for variants.

The detailed design level characterizes technical details, which are necessary to execute the development process on operational level. It complements the structural, economic information given by the RP on business level with the variants, named MCs. MCs provide the abstract process skeleton with detailed technical information to enable the computer interpretation of method-specific data, which will support developers in their daily work, when the process is automated. In Figure 3.6, the correspondence between structural and behavioral assets is shown. It shows, that VPs define a workflow and are associated with a set of behavioral variants. To derive a situational member of the process family, VPs and variants are characterized by a particular usage scenario, which is defined by a set of configuration criteria. On the one hand, the scenario determines, if an individual VP must be fulfilled or not. On the other side, the scenario enables the selection of one proper variant from the set of available variants, which match the actual situation best.

This section describes the set up of a process family, from which concrete process family members are achieved for the situation at hand. Therefore, the following describes the three main activities, as illustrated in Figure 3.3: *Process Family Definition*, *Variant Design*, and *Configuration Criteria Definition*.

3.3.1 Process Family Definition

The *Process Family Definition* activity defines the problem space, for which a process line has to be established. While the problem space defines a dedicated domain at large, its counterpart, i.e., the solution space, refers to individual sub-problem of that domain. Therefore, various information sources are used to abstract several isolated domain-specific process solutions and to analyze and define the process family problem space.

Product Type Variability	**Process Type Variability**
Functional Feature	Engineering Aspect
Workflow	Action Sequence
Data Format	Artifact data structure
System Access	Responsibility/Role/Skill
User Interface	Tool Functionality
Non-functional Properties	Method Characteristic

Table 3.1: Variability Comparison: Product and Process Components

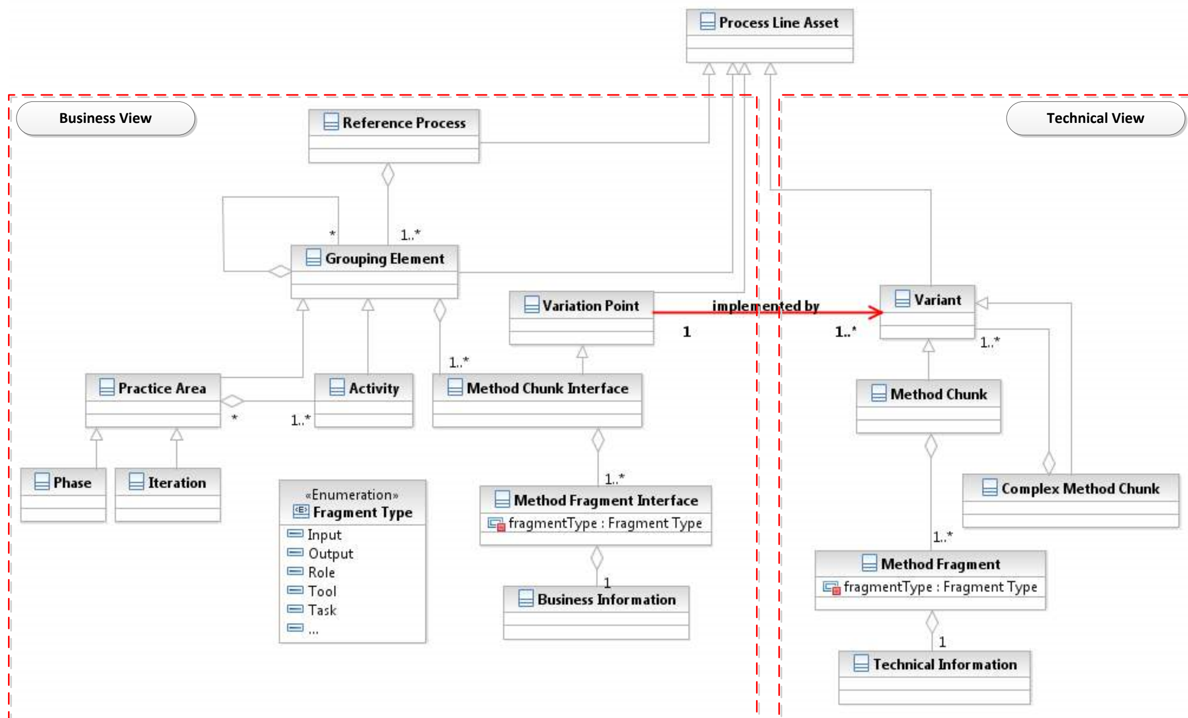


Figure 3.4: Process Line Engineering Assets

Exemplary information sources, from which information can be taken to define the problem space, are:

- Process Domain: Most companies provide a specific product range, which represents an independent domain. Normally, the domains require specific knowledge and focus on special product requirements which also influence respective development processes and activities.
- Product Domain: Besides the differences in process domains, there are also differences concerning the concrete product. As each product has individual characteristics, a respective process or method must consider this variability by providing adequate guidelines and product information for development, configuration and integration.
- Finalized projects: A multitude of past projects provide a good base for variability. By analyzing the differences in past strategies, criteria, which indicate the situational needs for individual activities, can be identified.
- Process Assessment Standards: Standards, such as CMMI or SPICE, recommend the usage of best practice activities. These practices consist of development activities or artifacts to be produced. To be conform with one of these standards means to implement respective practices as proposed. Depending on the respective standard, which must be fulfilled, some practices must be considered or not.
- Standards & Regulations: There are a lot of internal or external standards and regu-

lations, which standardize the accomplishment of individual activities or artifacts. Additionally, depending on a required compliance level, processes must or must not follow these standards and/or regulations.

- Customer Relationships: As most of the companies have multiple customer relationships, every single relationship causes other requirements concerning the exchanged data, such as individual information needs, data formats, or required tooling. Therefore, variability is caused through customizing the development process with respect to produced and consumed project specific information. This variability concerns artifacts and corresponding information content to be produced/consumed, the data format, and used tooling for seamless integration between collaborating parties and tools.

According to software product lines, where *a common, managed set of features satisfy the specific needs of particular mission are developed from a common set of core assets in prescribed way* [CN01], a process family uses a common set of process building blocks to specify the particular, situational mission of software product development. Therefore, the *Process Family Definition* activity identifies and implements all relevant components, i.e., assets, of a process domain. Afterwards, the assets are correlated to define an RP. The RP defines the temporal and/or logical order of variable and mandatory assets to set up a process family from the structural point of view, and ensures a member of the process family to comply with particular structural dependencies or correlations.
By defining the assets, an evolutionary repository, which aggregates various process and method related assets, is created. Thereby, new assets or „lessons learned" can simply be integrated into the repository by adapting the reference process or assets themselves. Executable processes are prepared by building up the reference process business assets and providing proper technical assets, which are configured during the situational process engineering phase (cf. Section 3.5).

The tasks, which have to be performed during the *Process Family Definition* activity, are illustrated in Figure 3.5. The figure shows, that this activity is subdivided into scoping tasks to identify building blocks (left part of the figure) and design tasks to define building blocks (right part of the figure). Both parts and the relationships between them are discussed in the following.

3.3.1.1 Process Base Analysis and Design

The *Process Family Definition* phase starts with an analysis of the process domain, where parts of the process family are identified to be logically integrated into a coherent entity in a subsequent step. The main task of this analysis is the identification and definition of GE and MC interfaces, as depicted in Figure 3.4. Therefore, we follow a top-down approach, as proposed in [BJJ+01], and distinguish the following three scoping activities: *process portfolio scoping*, *domain scoping*, and *asset scoping*.

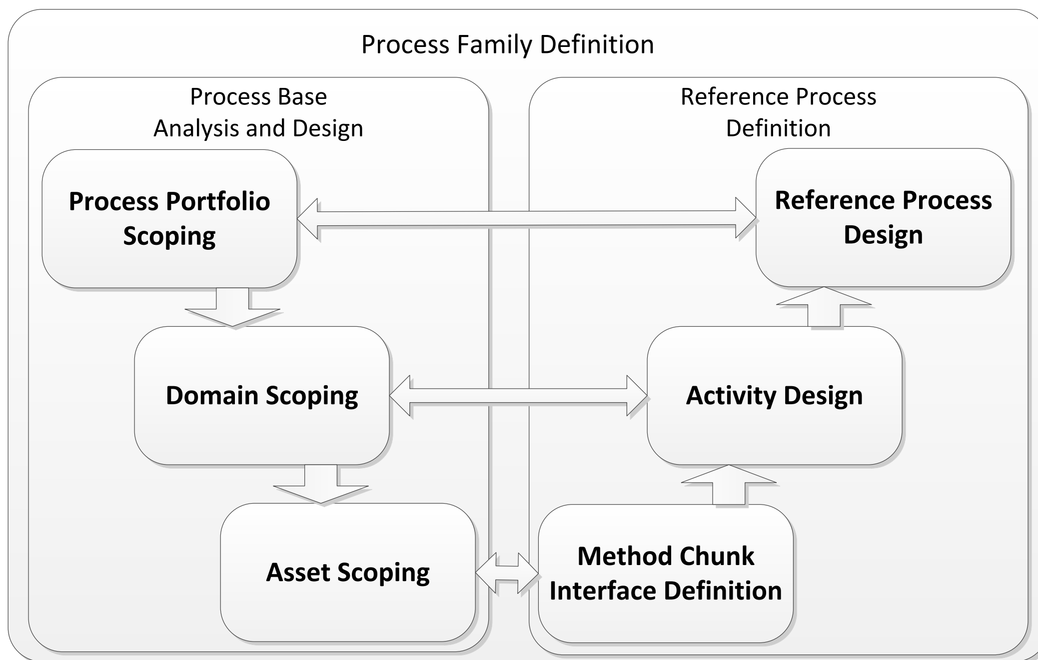


Figure 3.5: Overview: Process Family Definition Phase

Process Portfolio Scoping During *process portfolio scoping*, relevant practice areas, which the process family must consider, are identified. Aligned with Clements et al. ([CN01]) a practice area is „a body of work or a collection of activities that an organization must master to successfully carry out the essential work [..]" of a development process. Similar to feature-oriented domain analysis [KCH+90], various information sources, such as an enterprise product portfolio, affected domains, other reference processes (V-Model, RUP,..), assessment standards (CMMI, SPICE,..), existing processes or domain experts may help to define the scope of the process family. Thereby, complete processes and definable sub-processes, which vary in qualitative product and process characteristics, are identified. For example, within the automotive domain, processes to develop basic software, application software, hardware, systems, or software configurations can be identified. Furthermore, to enable a large number of process family members, as much as possible practice areas and processes should be identified to cover as much as possible situations. Identified practice areas and processes are used for further analyses in subsequent steps, i.e., they are decomposed into components, which afterwards are used to set up the all-encompassing RP in a well structured manner.

Domain Scoping *Domain scoping* systematically analyzes the identified practice areas and processes with regards to contained development activities. Activities are definable, short-dated, and constituent parts of a software development process, which are composed of a collection of closely-related methods relevant for individual product development. For example, timing analysis or software architecture design are different

development activities of a development process, whereas each activity is realized by an appropriate set of methods. Beside an unique identifier for an identified activity, the required development phase, such as RE, analysis, design, or implementation, and the affected modeling concern, such as requirements, function design, safety analysis, timing, or test, must be defined. That way, the domain scoping phase allows an easier identification of activities, which are redundant in different practice areas.

Asset Scoping During *asset scoping*, available activities are analyzed in detail to identify assets, which are necessary to realize development activities of a process. Contrasting the two phases before, where we identified abstract hierarchical structural building blocks or GEs, such as practice areas and activities, here we are interested in concrete actions, which must be realized by humans or technical resources. That means, we analyze identified activities with regard to contained methods, i.e., concrete actions, and associated MFs, such as (input and output) artifacts, roles, or applications, which are required for a method's enactment. For example, an RE activity could be decomposed into three de facto methods: elicitation, documentation, and analysis. Each method requires additional information about affected input and output, responsibilities, and tool-support. Therefore, following situational method engineering techniques (cf. [RBKJ06]), methods and associated method fragments are identified, to set up a method base, from which methods are assembled and evolved. Since asset scoping focuses on the general structure of methods, i.e., Method Fragments, we refer to the resulting description of a method as an MC interface.

Until now, the three scoping activities have identified a comprehensive set of structural or functional requirements in form of the GEs, methods and Method Fragments. During the derivation of situational process instances (Section 3.5), it must be decided, which functions are relevant for the process or not. Therefore, characteristics are required to detail the usage scenario of a particular asset, by which the process family is configured. To prepare the characteristics, during the three scoping phases it is necessary to identify all criteria, which caused the decision to define a new GE or MC interface. Each information source for the scoping phases, such as reference processes, a standard, a modeling aspect, or an individual development phase represent potential characteristics, which the process family must face. For example, the requirement that some processes must comply with an assessment standard, such as CMMI or SPICE, leads to particular development activities, which must be considered by the process family. In that case, CMMI and SPICE are candidates for functional characteristics of the process family and associated assets. Identified characteristics must be substantiated and recorded appropriately for later application.

3.3.1.2 Reference Process Definition

Similar to the V-Model XT or the Rational Unified Process (RUP), a reference process allows us to define a basic structure of a process, which considers multiple project sit-

uations on an abstract level. It is an ordered sequence of potential GEs, which may be indicated either as mandatory or optional to set up development aspect variability. During the *Reference Process Definition* activity, we order GEs, methods, and MFs, which were identified during the *Process Base Analysis and Definition* activity. However, while concepts are identified top down (from processes to methods), the RP is build from bottom up (from methods to process). Therefore, we firstly define methods, which are composed of identified method fragments. Secondly, we group the methods into activities and define a logical order (workflow) of MCs. Finally, we group the activities according to relevant practice ares, i.e., phases and iterations, of the process family, whereupon we define a workflow between practice areas to set up the overall RP. To reduce the complexity of the reference process design, we do not require complex control flow patterns. Instead, a $happens\text{-}before$ relation to indicate the predecessor of an asset is sufficient. During the final process derivation phase (Section 3.5.3), we demonstrate the automated derivation of essential control-flow patterns (cf. [WfM99b]) from that $happens\text{-}before$ relation. That way, we are in line with most of existing reference methodologies defined, e.g., by AUTOSAR, TIMMO, or MAENAD, and we are enabled to simply integrate their process as an RP into our approach.

The RP to be defined is disconnected from concrete realizations of the process and establishes a business-oriented perspective on the process. It defines the abstract workflow of the process family from a functional and structural point of view. To let the RP cover as much as possible project situations, the main objective on this level is to define the complete process domain faced by the process line without considering their concrete behavior. While the structure of the RP is adapted during the situational process engineering phase, contained structural method definitions serve as variation points, which are bound to a concrete behavioral method realization (or variant), which matches the situation at hand best. Therefore, as depicted in Figure 3.4, we distinguish MC interfaces, i.e., VPs, and MCs (realization), i.e., variants. The basic idea is to detach a well-proven basic process from the large set of applicable methods, which must be distinguished with regard to their usage scenario and behavior. Therefore, to achieve an RP, the *Reference Process Definition* is subdivided into three tasks, as discussed in the following:

Method Chunk Interface Definition To manage structural and behavioral variability of methods, we distinguish the interface level and the implementation level of methods. During the *Method Chunk Interface Definition* activity, we define MC interfaces to express required structural information of a method by combining the MFs identified during the *Asset Scoping* activity. The fragments, which are classified into input artifacts, output artifacts, roles, tools, and tasks, are combined using the relationships discussed in Section 2.4.2. Furthermore, MFs are detailed by individual business-oriented information, which may support management activities. Therefore, natural language text can be used to explain the purpose of the respective element and the asset's relevance. For individual cases, particular Key Performance Indicators (KPIs) or metrics are related with MFs to define their qualitative characteristics, such as an artifact's level of completion, the duration of a task, or the required tool compliance with a particular certification authority.

Activity Design During the activity design phase, MC interfaces are integrated into activities, as identified during the *Domain Scoping* activity. In parallel, a network showing the sequence of MC interfaces is defined to show precedence of work. We abstract the design from a concrete PDL, and only require the usage of a general $happens\text{-}before$ relationship to logically order MCs. The $happens\text{-}before$ dependency is simplified in Figure 3.6, where the *control-flow* relationship indicates, that VP_X $happens\text{-}before$ VP_Y.

Reference Process Design To complete the RP, we define practice areas by aggregating defined activities and dependencies between them. For practice areas, we distinguish phases and iterations. While phases indicate a large stage of the overall process, iterations are used for long-lasting repetitive development cycles. Similar to the steps before, a simple $happens\text{-}before$ relation between practice areas and activities is sufficient to define a control-flow on this level of abstraction.

The so far modeled RP only manifests the ordered sequence of functionalities in a process family. Finally, the functional variability of the RP is defined by identifying variable parts of the process family, which are relevant for particular process members only. Therefore, all components are analyzed with regard to common, i.e., mandatory, and variable, i.e., optional, development activities. If the RP contains activities, phases, or iterations, which are relevant for any member of the process family, the component must be indicated as mandatory. In contrast, optional components are allowed to be removed from individual process instances depending on situational requirements. However, as GEs aggregate other assets (cf. Figure 3.4), the defined variability of an asset influences contained assets. While the mandatory characteristic of an asset does not influence contained mandatory and optional assets, optional assets must be developed with respect to situational characteristics and potential conflicts. Since an optional asset is allowed to be removed from the process at configuration time, a contained optional or mandatory asset would be removed, too. Therefore, one has to decide about the proper hierarchical arrangement of optional assets.

3.3.2 Variant Design

The structure of the RP and VPs in form of MC interfaces is defined, during the *Process Family Definition* activity. The MC interfaces, were defined by combining the tasks with other method fragments, which describe product-related information (artifacts), responsibilities, and tool features. As discussed earlier, the interfaces describe the basic characteristics of a development activity from business-oriented point of view. They represent abstract classes of situational methods, which are realized during the variant design phase on a technical design level. On this level, different situational variants, i.e., concrete MCs, are defined for each MC interface, as indicated by the $implemented_by$ association of Figure 3.4. We follow this approach, since each development activity can be

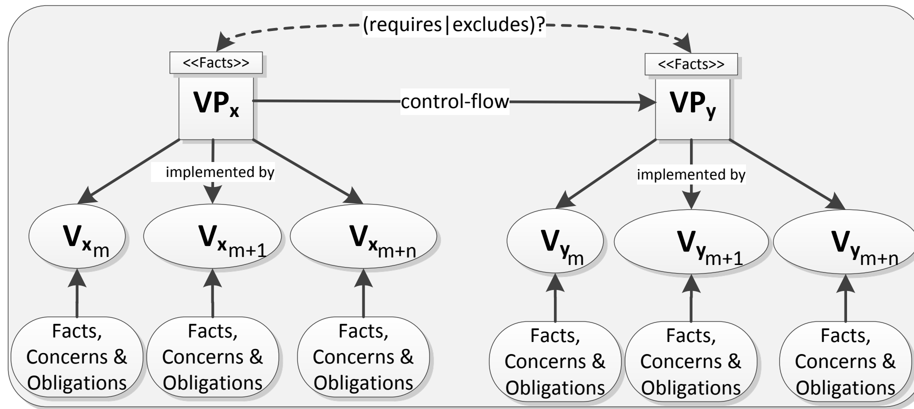


Figure 3.6: Design Level Correspondence

realized differently by applying particular techniques, which support different ways of thinking and other qualitative concerns. Hence, VPs are analyzed by considering project characteristics and other potential situations. The correlation between VPs and variants is also depicted in Figure 3.6, where VPs are associated with a set of variants V.

3.3.2.1 Method Chunk Design

Similar to MC interfaces, MCs are composed of method fragments by using well-known SME techniques, as before. The difference, though, concerns the information content, which is provided with fragments on the respective level. As depicted in Figure 3.4, while MF interfaces are associated with business-oriented information, MFs are refined by technical information. Although, technical design is detailed in chapter 4, the following gives a brief overview about fragments from the technical point of view, whose main goal is to extend the business-oriented level, with computer-interpretable information, which make processes executable. On technical level, we extend four basic fragment types:

- Artifact: For the definition of data formats, which are relevant for a particular development activity as input and output data, meta model information is provided with artifact fragments. This enables us to effectively provide developers with appropriate data, which can be manipulated at execution time.
- Role: For detailing responsibilities and relevant skills, which are necessary to execute a method, access rights, required capabilities and skills of developers are provided with the role fragment.
- Tool: To support developers and the execution of a method best, tool support is required. Therefore, features, which are required from the tool, are annotated with a tool fragment to either identify tools out of the box, or to derive customized proprietary tools (automatically).

- Task: For the definition of necessary best practices or guidelines, which guide developers in using input information to produce required output of a method, interpretable constraints to evaluate the produced output and recovery strategies are provided with the task fragment. This supports developers in doing their work by enabling the evaluation of guideline or best practice in the context of a specific method.

MCs on technical level are combined from above fragment types, either from scratch or by reusing already defined fragments stored in a repository. Therefore, fragments are combined into new MCs using specific relationships or existing MCs are extended by using techniques, as discussed earlier (replacement, extension,..). Furthermore, complex MCs can be defined from MCs, as illustrated in Figure 3.4. A complex MC is composed of multiple MCs and other complex MCs. On technical level, this enables us to define MCs with a complex control-flow semantics to detail the logical and temporal order of several more simple MCs, using control-flow nodes explicitly, in contrast to the simple *hapens-before* relation applied on business level. Since complex MCs are small, self-contained units, whose control-flow behavior can be defined with more reasonable expenditure than the overall process, an explicit definition of the control-flow can be realized without large efforts, and resulting complex MCs can be applied in resulting process family member one to one.

3.3.2.2 Variant binding

Finally, a binding mechanism between VPs and variants must be defined. As VPs are proxy objects, it must explicitly be defined which variant realizes a specific VP, before a subsequent analysis can identify the optimal variant from the overall set of variants. We call this variant binding, and identify two ways for defining which variant belongs to a specific VP: the *explicit* and the *implicit* bindings. Explicit binding is implemented by an explicit 1-1 association between an VP and a respective variant, and must be realized manually by a method engineer at design time. In contrast, for implicit binding, MC interfaces are matched with the fragment structure of MCs automatically, as demonstrated, e.g., in [RPR98] or [MR05]. As depicted in Figure 3.4, we apply the explicit binding strategy by providing the *implemented by* association, which binds each variation point to a set of potential variants.

3.3.3 Configuration Criteria Definition

The variabilities identified so far only manifest the existence of variability within the RP in form of optional and mandatory GE and variants. The model does not provide reasons, which cause variations or mutual conditions between variable elements. To distinguish variants from each other and to decide about the usage scenario of VPs or GEs, all assets are annotated with configuration criteria, i.e., situational characteristics, as depicted in Figure 3.6. Characteristics are qualitative properties, which concern e.g., a method's dependencies, feasibility, applicability, or behavior at process runtime. They are identified

in parallel with the *Process Family Definition* activity, as discussed above. Since the decision for the creation of an asset is based on individual reasons, the reasons are recorded and subsequently used to derive meaningful concepts for the diversification of usage scenarios. This holds for GEs, VPs, and variants equally. For example, if the RP must enable process members to be SPICE compliant, SPICE requires individual practice areas (process groups) and activities to be added to the RP. Since SPICE is the reason for the creation of individual assets, it should be used as requirement for process members and evaluation criteria for the selection of associated assets. While the process family definition phase identifies concepts informally, the concepts are organized, during the *Configuration Criteria Definition* activity.

As depicted in Figure 3.7, we identified three types of characteristics, which enable our evaluation mechanism to match the usage scenario of applicable variants and the required VPs with situational characteristics: Integrity *Constraints*, *Facts*, and *Concerns*.

To define mutual conditions between variants, we use the term *Integrity Constraint*. Integrity constraints are applicable to variants and define, whether one selected variant $requires$ or $excludes$ other variants. Based on a preliminary decision about the necessity of one component, it is a logical implication on the availability of other components. For example, a variant $V1$, could $require$ a variant $V2$, i.e., $V1 \implies V2$, or $V1$ could $exclude$ another variant $V3$, i.e., $V1 \implies \neg V3$.

The other types of characteristics, i.e., $Facts$ and $Concerns$, are more generic and cannot be defined without knowing the application domain of the process line. Instead, a general characteristics framework is provided, which can be reused for refinement in different domains. The following details the basic characteristics, as depicted in Figure 3.7, and provides examples to set up domain-specific characteristics, which can be annotated with assets to detail their usage scenarios.

Both characteristic types influence the ranking of process line assets during situational process engineering. The main difference between facts and concerns is that while facts can be decided in a binary way (i.e., a fact is either fulfilled or not), concerns are fuzzy properties, which should be factored optimally regarding the actual situation and the overall process context. Figure 3.7 shows that facts are further split into hard facts and soft facts.

Hard facts are mandatory characteristics, which must be factored during the process configuration phase, if they are required for the actual situation. That way, hard facts perfectly match the need to decide if a GE, a VP, or a variant, which was indicated as optional, is required or not. Hard facts, for example, ensure that a particular variant complies with a specific *standard*, such as International Organization for Standardization (ISO), Institute of Electrical and Electronics Engineers (IEEE), or International Electrotechnical Commission (IEC), or a reference process, such as CMMI or SPICE. Another example for hard facts are *process*-related characteristics, that provide assets with information about a particular process phase, for which it has been developed explicitly (e.g., analysis or design), or a covered design concern (e.g., timing, safety, or verification).

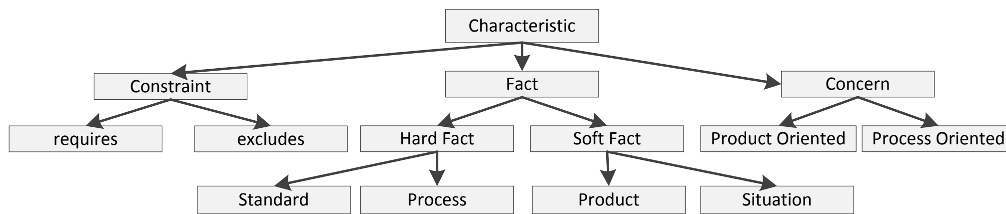


Figure 3.7: Characteristics framework

Soft facts can be either *product-specific* facts that mark variants, which are specialized for the development of a particular product from the product portfolio of a company, or *situational* facts, which may concern the application domain, or particular customer relationships. These soft characteristics should be considered during the process configuration phase where possible. Variants, which fulfill soft fact characteristics, can strongly improve the process, but their absence has almost no negative impact.

In contrast to facts, a concern represents a characteristic, which cannot be fully observed. This matches the semantics of variants only, since we are searching for variants, which best support dedicated situational criteria. We perceive concerns as non-functional properties, either concerning a process-oriented or product-oriented quality of a variant. The process-oriented quality aggregates properties to determine an enactment and execution behavior of process components, such as their time consumption or the required infrastructure prices. On the other hand, the product-oriented quality aggregates properties to determine the impact of a specific variant on the product under development. For example, to determine if a variant observes verifiability, scalability or other criteria regarding the product under development. In contrast to hard fact characteristics, it makes sense to weigh up concerns and soft-facts against each other, as in most of the cases the complete set of characteristics cannot be fulfilled at the same time. Therefore, to distinguish the level of detail, by which a variant fulfills an individual concern, we incorporate so-called qualifier tags for concerns, as described in [BAGS10]. For example, we can define *Low*, *Medium*, and *High* qualifier tags to qualify the costs of a specific variant. This is described in Section 3.5.

3.4 Feature Model Generation

Feature models are not only used to represent variabilities in different application scenarios, such as product line engineering. They are also a good means to represent the variability of a process family, in contrast to work breakdown structures or large-scale process models. Therefore, we apply feature modeling to represent process variabilities, which subsequently are resolved to derive situational members of the process family. Additionally, by using the feature model we integrate distributed information into one

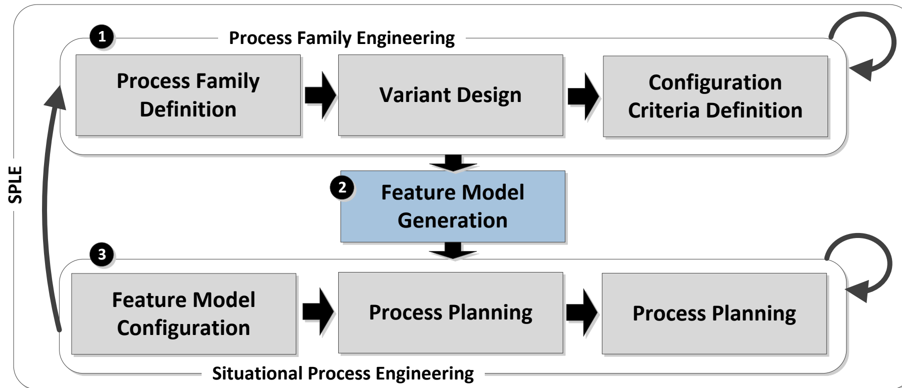


Figure 3.8: Software Process Line Engineering: Feature Model Generation

model, i.e., we combine the functional RP information with behavioral variants and situational characteristics. As a result, each time a data source changes, the feature model is generated as a stable intermediate data format between the definition domain of process family engineering and the planning domain of situational process engineering. As discussed in the following, the feature model is generated, automatically.

We defined an abstract mapping between process line assets and a cardinality-based feature model, which conforms to the definition of Czarnecki et al. [CHE05]. Similar to Feature-Oriented Domain Analysis (FODA)-based feature models [KCH+90], the process feature model is a tree, which hierarchically groups features and sub-features of the process family. Since any asset of the RP is required on feature model side, we use a structural relationship $consists_of$ to re-organize the aggregated set of process line assets in the feature model domain one to one, i.e., if an asset aggregates other assets of the process, the corresponding feature model uses the $consists_of$ relationship to relate a feature (asset) with sub-features (aggregated assets). Figure 3.9 exemplifies the mapping between parts of the RP and the corresponding counterpart in a feature model. The left part shows process patterns, which follow our process line engineering meta model, as depicted in Figure 3.4. The right part shows the mapping result based on the feature model notation. The first mapping rule (RULE 1) demonstrates the scenario, when an GE aggregates mandatory process assets. In the process line domain, one can see, that GE 1 aggregates four assets, i.e., the two GEs 1.a and 1.b, and the two VPs 1 and 2. On feature model side, the rule shows, that GE 1 $consists_of$ four assets, alike.
Furthermore, feature modeling enables the indication of optional or alternative features on each hierarchical level of the feature model. To represent optional GEs and VPs, the feature model uses optional features, as demonstrated in Figure 3.9 (RULE 3). In contrast, optional variants are not considered, as we require that for one VP exactly one variant must be selected from the overall set of bound variants based on situational project characteristics. This makes variants alternative implicitly.

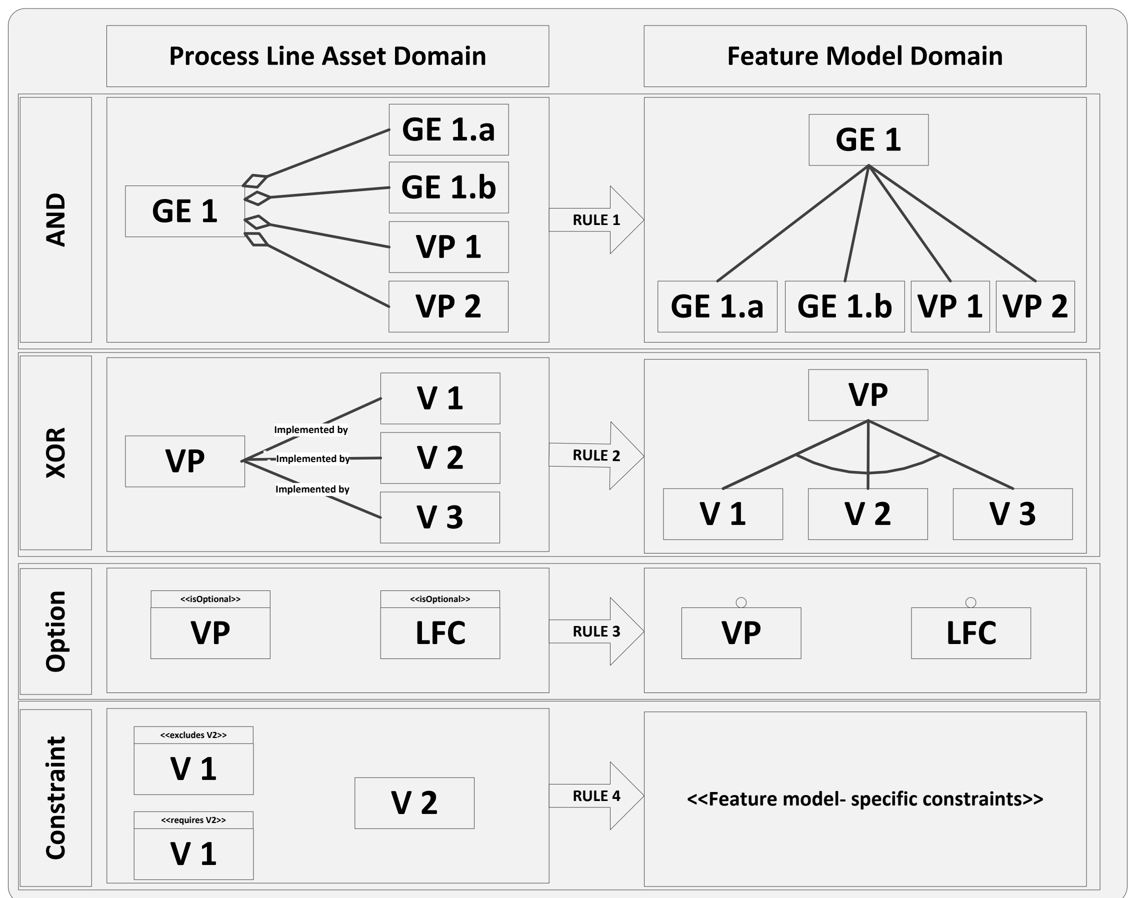


Figure 3.9: Feature Model Generation Rules

However, four different aspects of alternative variability and its resolution must be considered:

- Alternative design-time assets: this type of variability concerns alternative activities, i.e., GEs or VPs, inside of the RP, whose application depends on situational characteristics, which are already known during the process configuration phase. To reduce complexity of the feature model, we face this type of variability by optional assets and their annotated usage scenario. Alternative assets are resolved by the elimination of unnecessary optional assets by evaluating their usage scenario during the configuration phase.

- Alternative run-time assets: this type of variability concerns the different ways of thinking, which are applicable to realize a particular goal or development activity of a process at runtime. It is realized by the *implemented_by*-association between an VP and its associated variants. It is resolved by using the alternative feature notation, as depicted in Figure 3.9 RULE 2.

- Alternative design-time control-flows: this type of variability concerns alternative paths of an RP, which can be decided before run-time. To specify a control-flow of the RP at design-time the *happens-before* relationship is used to define logical and temporal dependencies. At configuration-time, the usage scenario of individual GEs and VPs is analyzed, whereupon some of them are eliminated depending on situational needs. The remaining components and their dependencies are used to derive basic control-flows based on particular transformation patterns, as described in Section 3.5.3, to derive a situational workflow.

- Alternative run-time control-flows: this type of variability concerns decision-based alternative paths, which are represented by an exclusive choice pattern (cf. [VTKB03]) within a process. The decision depends on information, which is only available at process execution time, such as artifact states or individual produced data. As this information is not available at design-time, we neglect this type of variability during process line engineering.

Finally, we must consider integrity constraints, which are annotated with assets to define the effect of the selection of one asset on other assets. Since there is no standard formalism for integrity constraints, RULE 4 of Figure 3.9 only indicates the transformation of integrity constraints into the feature model, as required from a respective feature model formalism. An example is given in the following, where we describe a general transformation of an RP and associated variants into a feature model:

1. First of all, the RP, which aggregates any other assets, is the basis for the mapping and maps to the root feature of the feature model;

2. Similarly, GEs inside the reference process are mapped to features and sub-features following the containment relationships given in the RP. As a result, a hierarchy of features and sub-features is produced;

3. For all features, except for the root feature, the optional or mandatory characteristic of an GE or an VP is mapped to the corresponding cardinality value of the respective feature. Cardinality values of 0 to 1 for optional and 1 to 1 for mandatory features;
4. VPs within the RP are mapped to a feature, which contains a mandatory feature to represent a group of available variants, which are identified in the following step;
5. For an identified feature group element, the associated variants, which are contained in the variant repository, are queried. This set is applicable as a realization of a specific VP (i.e., a feature group of the feature model);
6. Finally, the integrity constraints (i.e., $requires$ and $excludes$) are mapped to constraints of the feature model. For example, using the notation introduced in [AC04], the following templates express exemplary integrity constraints of a feature model: Let $f1$ and $f2$ be two variant features. To specify that variant $f1$ requires variant $f2$, the following syntax is used: $if\ (//f1)\ then\ (//f2)\ else\ true()$. Otherwise, if we want to express, that $f1$ excludes $f2$ the following is used: $if\ (//f1)\ then\ not\ (//f2)\ else\ true()$. In both cases, the template allows to replace $f1$ and $f2$ with actual variants.

Following this methodology, process models based on general meta models, such as SPEM or BPMN, can be transformed into a tree-based feature model and subsequently be integrated with our approach. While leaf nodes (atomic features) of the tree are MC variants, inner nodes are GEs or VPs, i.e., practice areas, activities and MC interfaces of an RP.

3.5 Situational Process Engineering

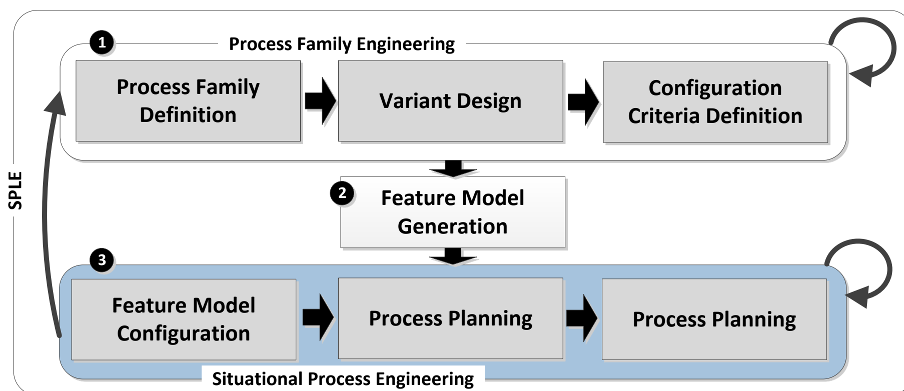


Figure 3.10: Software Process Line Engineering: Situational Process Derivation

Situational process engineering aims at deriving a target process from a process family, based on characteristics of the project at hand. Hence, project characteristics are captured, which form the requirements of the target process. According to the project characteristics, a method engineer decides on hard facts, which need to be satisfied. Moreover, soft facts and concerns are decided and their relative importance is indicated by a method engineer using qualifier tags. We assume, that a method engineer can decide about relative importance of soft facts and concerns based on requirements of the target project. For example, a method engineer could specify that scalability of the process is important more, than development time. After deciding the requirements, a situational method is generated through configuring the feature model, instantiating a reference process, and populating it with proper variants. Therefore, situational process engineering encompasses feature model configuration and situational process generation phases.

For realizing these two phases and, in particular, for planning situational processes using a feature model, we closely collaborated with researchers from two partner institutes in Canada. Thereby, we adopted the work of Asadi et al. for feature model configuration using AI planning to our needs of software process lines and exemplified it in a case study. As details about the applied planning approach can be found in [BAH+11], the following shortly sketches the main idea behind our planning approach.

3.5.1 Feature Model Configuration

At project start, project managers together with method engineers identify project-specific process requirements or goals. They either concern interim-goals of individual development stages or global goals of the final process. As any process is composed of functional process components and specialized behavior of different methods, both, i.e., functional and non-functional aspects, must be taken into account.
The configuration process starts with ranking atomic features (i.e., variants or leaf nodes of the generated feature model) based on their annotations and followed by selecting features, which satisfy hard requirements and optimize overall ranks of features. Feature ranks are calculated by employing a ranking process called Stratified Analytical Hierarchy Process (S-AHP) [BAGS10], which is based on Analytical Hierarchy Process (AHP) [Saa80]. We selected S-AHP because it enables the handling of preferences formalized in terms of relative importance and it is easy to use. S-AHP performs a pair-wise comparison between the characteristics (i.e., soft facts and concerns) by considering their relative importance indicated by method engineers. As a result, the absolute ranks of required characteristics are calculated, which is a value greater than or equal to 0 and less than or equal to 1. Next, the rank of each feature is computed based on the rank of soft facts and concerns, which are assigned to the feature. The rank of features is calculated based on the following utility function: $\sum_{i=1}^{n} w_i \times M_i(QT(f))$, where w_i is the weight of a characteristic i calculated in the first stage of S-AHP, and $QT(f)$ is the mapping function, that maps soft facts or qualifier tags of the concern, which are assigned to feature f to real numbers greater than or equal to 0 and less than or equal to 1. For example, if scalability

and cost concerns are qualified into high, medium, and low qualifier tags, the following mapping functions can be defined by method engineers:

$$M_{Scalibality}(QT) = \{Low: 0.2, Medium: 0.6, High: 1.0\}$$
$$M_{Cost}(QT) = \{Low: 1.0, Medium: 0.4, High: 0.1\}$$

Similarly, for soft facts, method engineers can define a mapping function which maps a soft fact to a numeric value between 0 and 1. Hence, if a feature contains some facts, the mapping function returns a numeric value, which is used in the utility function for calculating an overall rank of the feature. After calculating the rank of the features, we transform the feature model into the planning domain, and the configuration problem into the planning problem. To find an optimal plan and select features based on the returned plan, we employed the Simple Hierarchical Ordered Planner 2 (SHOP2) planner [NCLMA99] in our case-study . We chose HTN planning because it fits well with hierarchical domains, such as feature models, and planners in this domain are able to find optimal plans in a reasonable time [NCLMA99].

3.5.2 Generating planning domain

The planning domain encompasses operators, methods and tasks. Before transforming the feature model into the planning domain, we need to perform some pre-processing steps. First, we define a dummy feature f_d and then replace every optional feature f_o with a feature f_o^p, which is decomposed into two alternative features f_d and f_o, i.e., f_d XOR f_o. We should note, that the rank of the dummy feature is set to 0. Furthermore, we invert the feature ranks for finding a minimal solution. Therefore, we find the maximum rank in the feature model, and replace all the feature (including dummy feature) ranks with their difference from the maximum rank. For example, if the maximum rank is 5 and the rank of feature f is 3, then the rank of feature f is replaced with 2. After these two pre-processing steps, we generate the planning domain using the following transformation rules [BAH+11]:

- Hard facts are translated into domain predicates. For example, for the four hard facts CMMI and SPICE (to indicate that a variant is required to ensure the compliance with the standard reference process of CMMI or SPICE), as well as, STRUCTURE and BEHAVIOR (to indicate that a variant is required for structure or behavior modeling), the domain predicates CMMI, SPICE, STRUCTURE, and BEHAVIOR are generated. Moreover, one domain predicate is created for every atomic feature in the feature model. A propositional formula (called attainment formula) is created for each non-atomic feature according to the relations, which are located between the sub-features of the non-atomic features.

- An atomic feature f is translated into the operator o_f. Hard facts, which are assigned to a feature f (e.g., CMMI) are translated into a pre-condition of the corresponding operator $o_f \rightarrow precondition = CMMI$. The rank of a feature f (i.e., $R(f)$) is translated into a cost property of operator $o_f \rightarrow cost = R(f)$. A post-condition

(i.e., effect) of the operator receives the value of the domain predicated corresponding to the feature (i.e., $o_f \rightarrow effect = v_f$).

- An intermediate feature f is converted to a task t_f. According to the feature type (i.e., AND- decomposition and XOR-decomposition), one or more methods may be created. For an intermediate feature $f = AND(f_1, \cdots, f_n)$, one method m_{t_f} is generated, where all the tasks which correspond to sub-features are added to a list of the method subtasks (i.e., $m_{t_f} = t_{f_1}, \cdots, t_{f_n}$). For an alternative feature $f = XOR(f_1, \cdots, f_n)$, n methods $m^1_{t_f}, \cdots, m^n_{t_f}$ are generated. Next, task t_{f_i}, which corresponds to feature f_i is assigned to the subtask list of method $m^i_{t_f}$.
- Constraints (i.e., requires and excludes relations) are translated into preconditions of methods or operators depending on the type of a feature (i.e., intermediate or atomic features). We assume, that both features are atomic features; however, the general translation works for all combinations of atomic and intermediate features. If feature f_1 requires f_2, we add domain predicate v_{f_2} to the preconditions of operator o_{f_1}. When a feature f_1 excludes f_2, the negation of the domain predicate v_{f_2} is added to the precondition of operator o_{f_1}.

After generating the planning domain, we generate the planning problem, which includes initial states $\mathcal{S}$ and initial tasks $\mathcal{T}$. For the initial states, we set the domain predicates corresponding to hard facts as true. For example, if a method engineer asks for CMMI and STRUCTURE modeling, we assign the following values to domain predicates corresponding to qualifier tags of CMMI and STRUCTURE: $CMMI = STRUCTURE = true; SPICE = BEHAVIOR = false$. We also add a task corresponding to the root feature into the initial task set.

Having defined a planning domain and a planning problem, we employ the SHOP2 planner with required inputs (i.e., planning domain and planning problem). The SHOP2 planner returns a plan with minimum cost, that satisfies hard facts determined by method engineers. Afterwards, we select features corresponding to operators in the returned plan and their ancestors.

3.5.3 Final Process Derivation

Once a feature model is configured using the result of the SHOP2 planner, a process instance is generated directly from that feature model configuration, during the process derivation phase. The RP is adapted, by removing unnecessary features (i.e., GEs and VPs), and by eliminating unqualified variants, i.e., MCs. Afterwards, a workflow with an explicit control-flow is generated from qualified assets by clarifying the $happens_before$ relation semantics, as described below. This results in a situational process, which subsequently is validated and deployed.

As already mentioned, for documentation purposes, it is sufficient to model logical and temporal dependencies between components of the RP using the $happens\text{-}before$

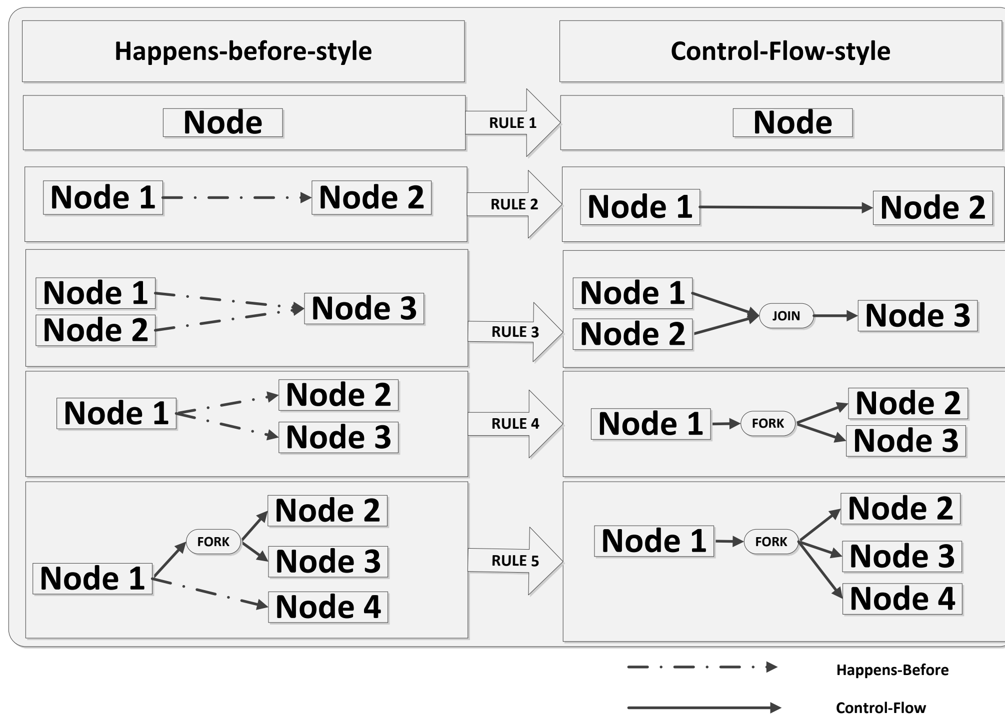


Figure 3.11: Basic Adaption Rules

relationship. While this modeling-style represents a network showing sequence of activities, which suffices most of the management activities, it is not suitable to define a explicit process execution semantics, as required for development process execution using a workflow management system, as discussed in chapter 5. For example, as the *happens-before* relationship only supports simple dependencies to correlate functional assets, no clear statement can be made about how to proceed (e.g., in parallel or choice-based) after the finalization of an activity, which has more than one successor. Therefore, we developed some simple adaption patterns for a process, which is based on a common *happens-before* semantics, to derive an explicit control-flow, as depicted in Figure 3.11. This pattern-oriented transformation is based on a configured feature model and the correlated RP. While the feature model provides information about relevant assets, the respective RP provides information about dependencies between these assets. In combination, these two information sources allow the application of adaption rules to generate the situational process with an explicit control-flow.

The previous step of (automated) process family feature model configuration simplifies this activity significantly. As the main goal of this configuration was to identify relevant practice areas and development activities, we can assume, that no alternative or optional paths, which would be caused by eliminated alternative/optional GEs and VPs of the RP, must be considered during the transformation. In other words, as variabilities were bound during feature model configuration, remaining assets and relationships indicate obligatory dependencies between development activities. From that point of

view, the three basic control flow patterns *sequence*, *parallel split*, and *synchronization* (cf. [VTKB03]) substantiate our control-flow behavior.

To derive an explicit control-flow from a process model, which is based on the *happens-before* relationship, Listing 3.1 shows an algorithm in pseudo-code. This algorithm applies different rules, as depicted in Figure 3.11, to create an appropriate control-flow. Each rule matches a particular constellation of ingoing or outgoing relationships on the side of an *happens-before* based graph, i.e., process, and creates a corresponding control-flow pattern on workflow side. As depicted in Figure 3.11 (RULE 2), two single nodes, which are connected via *happens-before*, correspond with a sequence on workflow side. In contrast, if a node has more than one ingoing or outgoing relationships, a control-flow node must be introduced. This is demonstrated in Figure 3.11 (RULE 3), where a node (*Node 3*) with more than one ingoing relationships has multiple obligatory predecessors (*Node 1* and *Node 2*), which all must be synchronized in *Node 3*. Therefore, a *JOIN* node, which synchronizes the finalization of all predecessor nodes, is introduced. On the other side, if a node (*Node 1*) has more than one successors, as depicted on the left side of Figure 3.11 (RULE 4), *Node 1* must be followed by any of these successor actions without priority. That means, that all succeeding actions can be executed in parallel, which is indicated by the parallel split node, named *FORK*.

```
1  input: happens-before graph g
2  deriveControlFlow(g){
3          removeDirectBypasses(g)
4          set actualNodeSet := nodes without outgoing arcs
5          for each Node n in actualNodeSet do
6            //Start node
7            if(n.ingoingEdges.size() = 0){
8                  continue;
9            }
10           else if(n.ingoingEdges.size() > 1){
11             //Create synchronization
12             Join join = createJoinNode
13             setEdgeFromTo(join,n)
14             for each Edge pre in n.ingoingEdges do
15               setEdgeFromTo(pre.source,join)
16               actualNodeSet.add(pre)
17             od
18           }
19           //Sequence
20           else{
21             setEdgeFromTo(n.ingoingEdges.first.source,n)
22             actualNodeSet.add(n.ingoingEdges.first.source)
23           }
24           //if a node aggregates a sub-process
25           if(n instanceof GE){
26             deriveTechnicalProcess(n)
27           }
28         od
29 }
30
31 setEdgeFromTo(Node source, Node target){
```

```
32        boolean forks = doesFork(source)
33        boolean mustFork = hasNormalOutgoingEdge(source)
34        if(forks){
35          //Find parallel split
36          Fork fork = source.outgoingEdges.first.target
37          edge.source = fork;
38          edge.target = target;
39        }
40        else if(mustFork){
41          //Create parallel split
42          Fork fork = createForkNode
43          edge.source = fork;
44          edge.target = target;
45        }
46        else{
47          //Create sequence
48          edge.source = source;
49          edge.target = target;
50        }
51 }
```

Listing 3.1: Generate workflow semantics

The algorithm in Listing 3.1 applies these adaption rules to a $happens\text{-}before$ process to derive a process with an explicit control-flow. As input it takes a directed graph (the process), which we assume to be acyclic. Before the algorithm starts, some simplifications are made on the graph:
Basically, there can be multiple paths of different length between two nodes. Although, as the semantics of the edges on each respective path is an obligatory $happens\text{-}before$ relationship, we are not allowed to shorten the process by bypassing individual nodes. For example, if there is an edge between node 1 and node 2 ($1 \rightarrow 2$) and another edge between node 2 and node 3 ($2 \rightarrow 3$), the path ($1 \rightarrow 2 \rightarrow 3$) states, that 2 must precede 3 and 1 must precede 2. Another edge, which directly connects node 1 and node 3 would shorten the path, but it would neglect the obligation, that node 2 must be visited (or executed) before node 3. That way, direct edges between nodes, which bypass other nodes, can be removed from the graph, before the algorithm starts to match adaption rules. This is realized in Listing 3.1 by calling the *removeDirectBypasses* function.

After simplifying the graph, the algorithm initializes a node set with all final nodes (i.e., nodes without outgoing edges) of the process, and traverses the graph backwards for matching the rules. The algorithm takes one node out from its actual node set and follows its ingoing edges to identify preceding nodes, by which the node set is actualized. The algorithm introduces new edges and control-flow nodes, as shown in the listing. At the same time, all visited $happens\text{-}before$ relationships become obsolete. If the algorithm identifies a node, which contains a sub-workflow (i.e., applies for GEs), it uses the GE as container node and processes the sub-workflow using the above algorithm recursively. The algorithm stops, when the node set is empty, i.e., it has reached start nodes, which must not be further processed. The resulting control-flow consists of multiple parallel paths, which are composed of nested split and synchronized control-flow sequences.

Besides the above basic transformation rules, which we demonstrated in Listing 3.1, more complicated control flows can simply be taken into account based on the individual semantics of certain GEs. According to Figure 3.4, different types of GEs are possible, which depend on the applied PDL. Using the example of an Iteration, i.e., a practice area which can be repeated several times, we demonstrate the concept of an advanced adaption rule in Figure 3.12. On the left side of the rule an GE is indicated to be an $Iteration$. During the algorithm, which is given in Listing 3.1, such a node can be handled as container node, for which a sub-workflow is created separately. Only the iterative characteristic of that container node must be considered by introducing proper control flow nodes, which can be connected with other nodes, as described above. Figure 3.12 shows, that the resulting decision-based pattern on the right side has one ingoing edge and one outgoing edge. For that reason, it simply can be combined with rest of the control-flow on one level.

3.5.4 Resource-oriented Process Analysis

After the derivation of a situational process, various characteristics must be validated to ensure successful process execution. Some basic characteristics of the process are:

- Completeness: The process must be complete with respect to activities and artifacts. That means, that not only any required features, i.e., development activities, which were required during the process configuration phase must be fulfilled, but it also requires, that the final and intermediate products are produced.
- Reachability: The reachability of a process concerns the generated control-flow of the process. Since the control-flow was derived automatically, it must be ensured, that starting from an initial node a final node can be reached. Therefore, the process is required to have at least one start and at least one final node. Additionally, it must be ensured, that there are no unreachable or dead nodes.
- Decidability: This concerns the validity of decision nodes within the control-flow. As decision nodes must enable to make explicit statements for selecting exactly one alternative path from a set of paths, they must be annotated with conditions splitting the decision space into disjunct parts, where each part represents an alternative path.

Different techniques, such as simulation [JVN06], flow analysis [SB10], or graph-based algorithms [KKGL10], are available to identify and/or eliminate above drawbacks and can be implemented to meet the specific requirements of an individual PDL.

In contrast, Sadiq et al. [SOSF04] identified several conflicting situations, such as missing, mismatched, or inconsistent data, whose availability must be ensured before process enactment, as well. Therefore, this section focuses on the validation from a different point of view, which is often neglected, when talking about workflow validation.

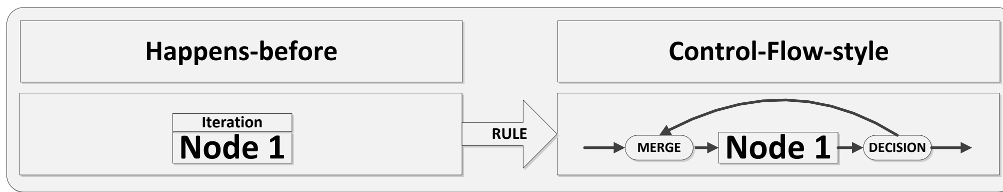


Figure 3.12: Example: Advanced Adaption Rule

As the realization of a workflow significantly depends on available resources, the following sketches basic ideas of an approach, which we developed for validating the feasibility of a workflow regarding the available resources of an enterprise or other involved parties. While details about the approach can be found in [BEFH12], most relevant parts from the published paper were directly integrated with this section.
In a nutshell, our approach for analyzing the feasibility of processes is the following: given a model of the process and a semantic model of the available resources within a company or department, we utilize semantic technologies, especially logical reasoning, in order to come up with infeasible MCs and the respective missing resources. Therefore, both MMTS and OTS need to be integrated via a mapping. Figure 3.13 depicts an overview of the concept with its three building blocks: *Process Model (PMod)*, *Resource Ontology (ROnt)*, and a *mapping* between these two worlds.

3.5.4.1 Method Chunk Resources

Beside product and process-related method fragments, MCs are also composed of Resource Fragments (RFs), which state the requirement for individual human or technical capabilities, which are required to realize an activity. On the other hand, an RF may provide a cardinality attribute to indicate the required quantity of the respective resource. For example, an MC, such as double-blind review, needs two roles of the same type to realize the four-eyes principle.

The left part of Figure 3.13 shows the PMod, which represents a concrete process model. The process is realized by using MCs, which are composed of different types of method fragments. The figure only illustrate resource-oriented method fragments (RF), such as e.g., tools and roles, which are assigned to the respective MC via a requires relation. The RF symbols show both, the cardinality as a hash symbol # on top and the name of the resource below. Note that, for clarity reasons, we are distinguishing between roles, i.e., employees with specific skills, and tools, i.e., infrastructural capabilities. However, since the approach presented here is generic regarding the concrete PDL, this is not a requirement for a potential method definition approach.

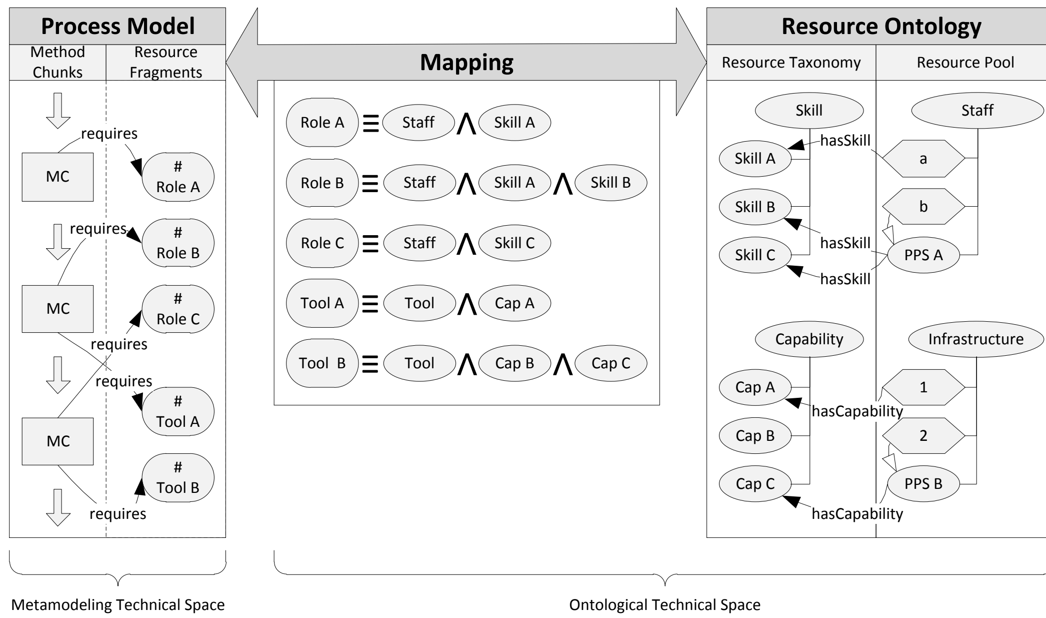


Figure 3.13: Concept of the resource-oriented analysis of processes.

3.5.4.2 Resource Ontology

The right part of Figure 3.13 depicts the ROnt, i.e., a model of the resources within a company or department represented in the OTS. On the one hand, this model conceptualizes company-internal resources, and builds up a *resource taxonomy* of skills and infrastructural capabilities of interest. On the other hand, it also comprises the *resource pool* of the company, i.e., concrete employees and infrastructure. By putting both in one ontology, it is possible to assign each employee to the skills he or she has, and to assign each tool to the capabilities it provides, respectively.

The resource taxonomy is a hierarchical definition of the skills, and the capabilities which are available or interesting within a company. Thereby, skills refer to concrete knowledge, capabilities, or access rights which an employee can have, e.g., tax law knowledge, project management skills, or access rights to particular design documents. In order to come up with a reasonable taxonomy of the skills, they should be categorized and aggregated to umbrella terms. For instance, the skills Java programming and C programming are classified under the skill programming languages. The same ideas apply to the taxonomy of the capabilities. A capability refers to a functionality of infrastructure components within the company. The information for building the resource taxonomy can be gathered from experiences or knowledge of experts, or from bodies of knowledge, such as [AMB+04].

The resource pool is a collection of concrete members of staff and instances of tools (depicted as hexagons in Figure 3.13) within the company or department. For instance, there can be a representation of the real employee Bob and the Laboratory X. Each member of staff and tool is linked to the skills and capabilities it provides via a hasSkill or has-

Feature relationship, respectively. This allows the concrete specification of the abilities of the company's resources. Furthermore, there can be concepts, which define a specific set of skills or capabilities, respectively. They are called *Predefined Property Sets (PPSs)* and depicted as ellipses in Figure 3.13. For instance, a concept software architect can define that an assigned employee has both a programming language skill and project management skill. This way, it is possible to represent company-specific job titles. Actually, PPSs can be seen as shortcuts for skill assignment by assigning a member of staff to them instead of single skills.

All in all, the ROnt acts as a skill database for human resources and, additionally, contains equipment and tools including their respective capabilities. This knowledge base is developed for a company once, and has to be kept up to date. Subsequently, it can be exploited in our approach to validate the feasibility of processes from a resource-oriented point of view.

3.5.4.3 Mapping

In order to exploit the ROnt for a feasibility analysis of a process, a translation between the RFs in the PMod and the skills and capabilities in the resource taxonomy is necessary. This is provided by the mapping depicted in the middle of Figure 3.13. It is a separate ontology and, therefore, technically belongs to the OTS. The basic idea of the mapping is to define an RF as an aggregation of concepts from the resource taxonomy, i.e., skills or capabilities.

A mapping between a concrete RF and concrete skills or capabilities is represented as an ontological equivalence. Each RF is represented as a distinct concept in the mapping. Thereby, the matching between RFs in the OTS and in the MMTS can be based on, e.g., name equality. Now, the ontological RF is defined to be equivalent to either an employee with a set of skills or a tool with several capabilities. An example of a mapping is:

```
software architect is equivalent to Staff
                                    and hasSkill software_architecture
                                    and hasSkill project_management .
```

To enable the usage of the resource taxonomy in the mapping, it imports the whole ROnt. Hence, no specific matching between skills and tools in the mapping, and in the resource taxonomy is necessary because both reside in the OTS.

In general, the set of skills and capabilities for an RF is interpreted as a conjunction, i.e., the employee or tool has to have *all* stated skills or capabilities. However, depending on the expressiveness of the utilized ontology language, this simple semantics can be extended by more complex structures, such as disjunctions. This would allow to express a logic like a requirements engineer has a skill in use cases or user stories. Note, that the mapping defines a translation for RFs, hence, the cardinalities for the RFs are neglected.

The mapping can be seen as the definition of a matching between the RFs, of a specific process definition language and the resource taxonomy of a company. Hence, the

approach of decoupling both PMod, and ROnt allows reusing the company's body of knowledge for analyzing the feasibility for processes independently from the used process definition language. It is solely necessary to define the mapping once for each combination of process definition language and resource taxonomy. Note that, however, it has to be ensured that the names of the RFs in the language are unique, i.e., no two different RFs have the same name.

3.5.4.4 Ontology-based Resource Analysis

Once, the ROnt and the mapping are defined as outlined above, this knowledge can be exploited to analyze the feasibility of a specific PMod or process family member w.r.t. the required resources given the available resources within a company or department. Thereby, the definition of the ROnt and the mapping in the OTS turns out to be an enormous advantage, since this allows utilizing reasoning which enables the automation of this task, as detailed in [BEFH12]. In a nutshell, it checks whether each and every MC in the process definition can be executed, i.e., whether there are enough resources in the resource pool for the RF assigned to the MC. Specifically, the feasibility is analyzed by querying all resources from the resource pool, i.e., staff or tools, which match the required RFs and comparing their count with the cardinalities. This results in a set of infeasible MCs, that cannot be fulfilled by the resources in the resource pool. This information can be utilized by the user to either re-plan or adapt the process by, e.g., replacing the infeasible MCs, planning a training program for developing the missing skills, or purchasing new tools.

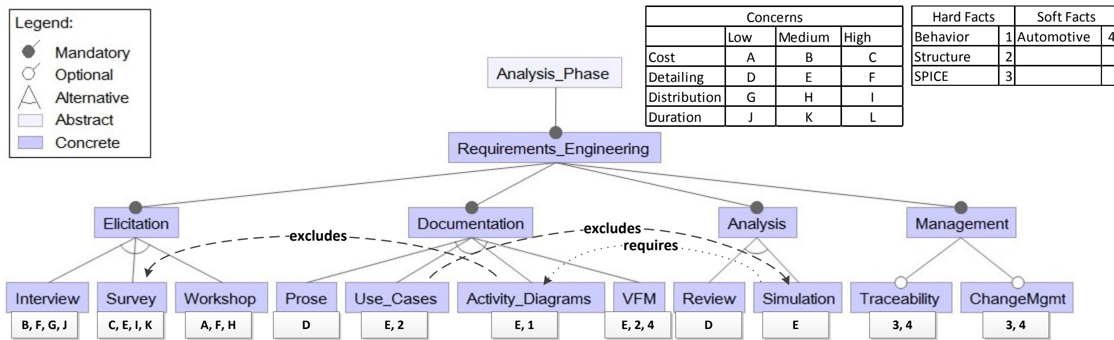
3.5.5 Deployment

The final deployment means the transformation of situational process instance on operational level. This is detailed in chapter 5.

3.6 Case Study

We employ RE to demonstrate and validate our approach. During the *Process Definition Phase*, we apply an iterative process, which is composed of four VPs, i.e., four abstract RE activities [KS98]: *elicitation*, *documentation*, *analysis*, and *management*. The model serves as a reference process, whose concrete realization must be configured with variants supporting the current situation.

During *Variant Design Phase* various variants are identified as potential realizations of the aforementioned VPs. For example, requirements documentation can be realized by using prose, use case modeling, or activity diagrams or traceability and change management may support the abstract management task of RE.



	Concerns		
	Low	Medium	High
Cost	A	B	C
Detailing	D	E	F
Distribution	G	H	I
Duration	J	K	L

Hard Facts		Soft Facts	
Behavior	1	Automotive	4
Structure	2		
SPICE	3		

Figure 3.14: Requirements Engineering Feature Model

After variants are bound to their respective VP, configuration criteria are defined. For the *documentation* and *management* VPs, the variants provide hard facts to describe situations, where the variant is mandatory for the result process. For example, traceability and change management are mandatory tasks for processes, which must conform to the SPICE reference process, and use cases must be incorporated if structure modeling is important. The Vehicle Feature Model (VFM), which is an automotive-specific variant for documenting requirements, is annotated by the soft fact *automotive*. This indicates that the variant should be preferred, if the process should be applicable to the automotive domain. In contrast, if there is no support for the VFM modeling in the repository, other variants are allowed to satisfy the requirement, as well. Besides hard and soft facts, all variants are annotated with concerns and their relative importance. As described in [BAGS10], concerns, which are annotated with variants, are qualified using high, medium, and low values, to describe how good a variant fulfills a concern. For example, as the level of detail for the interview variant is high, the variant is applicable to projects, where the level of detail plays a major role. On the other hand, the interview variant provides little support for situations, where the project lacks time and project partners are globally distributed. Finally, the dependencies between variants indicate, that a particular variant $excludes$ or $requires$ some other variant, if it is selected. For example, if Activity Diagrams are used to document the requirements, then a survey must not be selected for the elicitation step.

Figure 3.14 exemplifies the corresponding feature model, which is generated from the reference process and the variant repository, as described before. For all VPs, i.e., reference process activities, various variants are available. The features annotated with characteristics, as introduced in Section 3.3, and obligations are transformed in constraints into the respective feature model syntax.

In order to derive a situational requirements analysis process, requirements for the process are defined. We assume, that a method engineer requires the process to be compliant with SPICE, and defines relative importance between soft facts and concerns, as shown in Table 3.2. For our example, we apply the S-AHP method and compute the ranks of characteristics which are as follows: $Automotive = 0.10, Distribution = 0.33, Duration =$

	Automotive	Distribution	Duration	Cost	Detailing
Automotive	1				
Distribution	5	1			
Duration	7	3	1		
Cost	1	$\frac{1}{7}$	$\frac{1}{7}$	1	
Detailing	$\frac{1}{3}$	$\frac{1}{7}$	1	1	1

Table 3.2: Relative importance of characteristics for the current situation

$0.41, Cost = 0.06, Detailing = 0.10$. Next, we calculate the rank of each feature through the utility function defined in Section 3.5, considering the mapping functions for concerns and soft-facts, as defined by the method engineers. In our example, we consider the mapping function similar to the function as defined for scalability in Section 3.5 for distribution, duration, and detailing as well. If a feature has the Automotive soft-fact, then we replace it in the utility function with 0.5. Next, by applying the transformation rules, we generate the planning domain and the planning problem. The SHOP2 planner returns (an) optimal plan(s) and consequently corresponding features are selected. For the defined requirements, two optimal plans were returned by the planner: *Plan 1 (Survey, Use-case, Review, Traceability, and Change Management)* and *Plan2 (Survey, VFM, Review, Traceability, and Change Management)*. Accordingly, corresponding atomic features and parent features are selected. From existing configurations, a method engineer selects the final configuration. Afterwards, based on the configuration, a reference process is instantiated and proper process variants are bound to the process.

4 Computational Method Engineering

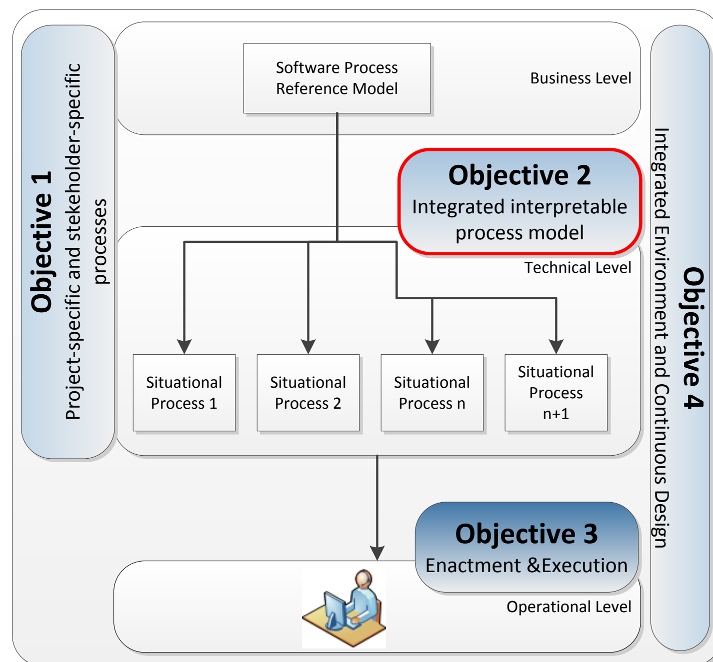


Figure 4.1: Objectives overview

After introducing our approach for the situational design of processes and methods on different abstraction levels in chapter 3, the following details the design on technical level to face *Objective 2* of this thesis, as illustrated in Figure 4.1. Therefore, we complement the management-related process information on business level, with computer-interpretable information, which particularly considers the realization of MC interfaces in combination with a flexible control-flow design. To realize this, we extend the MFs comprising an MC with a model, which enables the definition of computer-interpretable data structures, guidelines, and editor information. This chapter faces relevant design challenges and provides solutions to make development processes alive and to guide developers more effectively.

4.1 Motivation

Many quality norms, such as ISO15504 [ISO08a], ISO26262 [ISO10], ISO9000 [ISO05], CMMI [cmm08], or the IT Infrastructure Library (ITIL) [OGC07], require, that process-related information is captured by documents. To show an enterprise's standard compliance, today's development processes are documented by using either informal textual documents (cf. the OPEN Process Framework [OPE09]), or the syntax of an PDL, which more focuses general business processes, than the creative needs of a development process. More often than not, the circle is complete, when process models result in a set of documents, which summarize all the information using natural language text and illustrative figures. Indeed, CAME environments, such as IBM's Rationale Method Composer (RMC) [IBM10] or the Eclipse Process Framework (EPF) [EPF10], may provide means to enact modeled processes (i.e., they provide predefined project-specific development artifacts, such as templates or checklists), but further applications, such as process coordination or situational guidance beyond the provision of documents, are not enabled using a today's PDL. Especially, as development processes are creative and flexible more than general business processes, automation is a critical and unpopular task. Most people compare automation with control and restriction. However, automation can also be used to support the creativity and efficiency of developers by shifting a developer's focus from trivial or time-consuming tasks to real creative activities.

To realize this, process models must provide additional information. Therefore, the following requirements describe, what we expect from a framework to support the design and automation of development processes.

4.1.1 Process-centric Requirements

Today's development process models are more relevant for documentation, than for execution. They, indeed, provide simple indications about how to proceed in certain situations, but a preferable more sophisticated coordination and automated control of tasks and developers is hardly possible. This is due to the fact that most development process models do lack explicitly defined control-flows.
Therefore, our design must provide adequate interpretable process models, which enable us **to coordinate and to monitor development activities**. Such process model must enable

- **Process coordination** to define a flexible way, by which sequences of required activities can be managed
- **Traceability of tasks and artifacts** to ensure consistent data and to coordinate creative processes
- **Process validation** to check control and information flow of processes regarding their consistency and validity
- **Process deployment** to simply transform extended design models into platform-specific and executable units

4.1.2 Method-centric Requirements

Today's development tools are all-purpose tools and do not sufficiently affect developers work with respect to methods to be accomplished. Especially, for model-driven development it is difficult to differentiate method-specific information, i.e., artifacts, from the global information storage, the model. Additionally, the automation of situational guidance, artifact-specific data consistency, contextual failure analysis, or the management of responsibilities is out of scope for most of the cases.
Therefore, our design must focus the **methodological and context-specific aspects of a development process**, i.e., it must enable

- **Identification artifacts and their actual content** to provide situational guidance and validation for the development of artifacts, the
- **Provisioning of method-specific tooling support**, to not provide developers with unnecessary and confusing capabilities, the
- **Activity-specific guidance** and **product validation**, to shelter developers from careless mistakes, and the
- **Resource related information in the form of roles**, to shelter developers from misuse

As process definition, in general, is a regular project management task, we think, that resulting process models must influence development activities more efficiently than today. However, conventional PDLs and large-scale textual documents are not sufficient for this. Considering development processes, process models only set up an order of activities. In rare cases, this order is used to automatically assign activities and associated documents to a developer, which is considered to become familiar with the provided documents. As this is a time-consuming and complex task, we aim at the integration of additional, formalized knowledge into the process model. Therefore, in our layered process modeling approach, we complement the business layer information with additional information on a technical level. To support developers' work, the refined model provides an integrated view on computer-interpretable information. This section describes the design of interpretable process models and relevant information, by which platform-specific code is generated afterwards to shift the enactment of processes to real execution on operational level and to provide an efficient guidance system for developers.

4.2 Overview: Computational Method Engineering

Our approach keeps existing knowledge by (re-)using existing process models. However, as process models lack a detailed computer-interpretable information concerning process-, product-, and human-centric information, models must be extended by this additional concerns.

First, for process automation, we complement business-oriented process models with explicit control-flow semantics. As detailed in Section 3.5, this is prepared by transforming the business-oriented view into the technical view, which introduces additional control nodes. In this section, a new process execution paradigm is developed to enable the automation and coordination of creative development processes. Therefore, in Section 4.3, we identify new requirements for development process automation and coordination, and detail a mechanism for flexible workflow management. Afterwards, in Section 4.4-Section 4.7, we detail the technical refinement of MCs and associated fragments to provide developers with situational knowledge automatically. Finally, a case study will show the application of our approach in Section 4.8.

Instead of reinventing the wheel and building a new process (meta) model, we enable the extension of existing ones with additional information to centrally keep relevant information in one model. On the technical design level, this is realized by the application of the aspect-oriented modeling paradigm [SRF+] to weave necessary information to the given core of a process language in the form of aspects. The advantage is, that aspects extend models without affecting or changing the original PDL semantics. In spite of extending the meta model directly, aspects represented by a third party meta model are loosely coupled with individual target meta model elements on demand.

Based on the aspect-oriented meta model extension mechanism, which is described in [Lau10], we developed an aspect meta model, by which a custom PDL is extended. The meta model, which is depicted in Figure 4.2, defines aspects, which are woven with individual product line assets, as introduced in chapter 3. By extending the general concepts with technical aspects, we ensure that the extension remains applicable to most of known PDLs.
Each extension is identified by the name and the version number of a profile, which aggregates an extensible set of relevant aspects. For demonstration purposes, we defined two aspects to combine process-centric guidelines and product-centric data with process models. The *MCGLAspect* extends the process-related MF with a guideline model, which specifically holds for one MC. The product-centric part is covered by the *MF-DataAspect*, which targets product-centric MFs, i.e., artifacts. This aspect is used to extend business-oriented information of a method's in-/output by additional information, which describes the underlying product more formally. Further aspects, concerning e.g., the human-centric information, can indeed be added, but go beyond the scope of this thesis.
In summary, we aim at the definition of an Executable Process Guidance Model, which we define as follows: An Executable Process Guidance Model (EPGM) is a process documentation model, whose provided information is formally enriched by information which enables computer-assisted

- Context-sensitive assistance for the application of methods and processes
- Precise artifact descriptions, which enable unambiguous identification and further processing of relevant data
- Method-oriented user interfaces

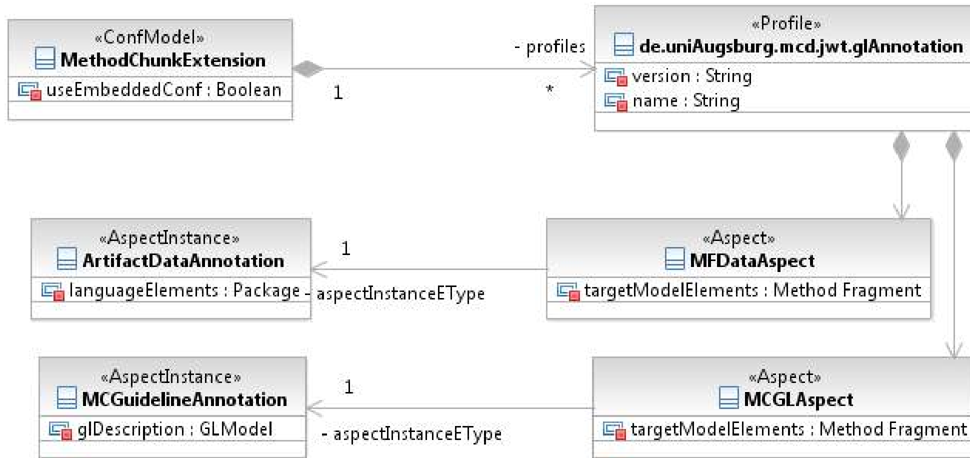


Figure 4.2: Aspect-oriented implementation for semantically enriched process models

- Management of method and artifact-specific data access
- Coordination of defined process activities
- Context-sensitive validation of used and produced artifacts

The following sections show process engineering on the technical level in detail. We introduce meta models and design techniques to realize additional information models, which enable executable process guidance.

4.3 Technical Process Design

Before introducing the design of MCs from the technical point of view, we focus on particular details of a technical control-flow design, which is required for a subsequent automation of a development process. Contrasting the business level, which is mainly used to support management activities and documentation, processes on a technical level must define explicit control-flows, which supports long-lasting, iterative, creative, and sometimes unpredictable processes. Control-flow semantics must enable workflow management systems to make unambiguous decisions about the processing of a development process and to allocate tasks with associated MCs correctly. Therefore, this section details the minimal set of syntactical and semantical requirements, which an PDL must fulfill from a design point of view. Section 5.6 will detail the application of that requirements for process execution.

4.3.1 Requirements for Development Processes on Technical Level

Basically, any documented process can be used to coordinate a set of activities. Since development processes are business processes too, established business process manage-

ment methods and workflow patterns [VTKB03], whose origins are workflow management systems [JB96], can be ported to coordinate development activities. Unfortunately, development processes introduce additional requirements, which differ from general business processes. Therefore, well-known workflow management principles cannot be applied one to one:
General business processes are short-dated and characterized by workflows, which can be repeated several times in the context of a specific business scenario, such as the order or reversal of goods. Various processes are defined for specific, isolated business case scenarios to simplify employees' work by the sequential assignment of tasks and customized user interfaces. Such a scenario considers the processing of standardized data, where creativity and interleaving with other scenarios of one spanning main process, are less important, than simple execution of an ordered set of tasks.

In contrast, development processes are characterized by long-lasting lifetime and an iterative incremental development. Typically, one main process is subdivided into phases, sub-phases, activities, and creative tasks, which are strongly correlated and processed by different developers in parallel and iteratively. Therefore, a development process can not be processed straight-forward, but must react on environmental events, such as design decisions, change requests, project situations, and customer requirements. At the same time, consistency of artifacts and a faultless interaction of different aspects of the entire system must be ensured.

Therefore, the management of development processes faces different challenges caused by a higher need for creativity and flexibility:

- Higher coordination needs because of multiple developers are working on different dependent artifacts distributed across the long-lasting development process
- Higher need for the iterative rework of individual activities and tasks contained in the process
- Higher needs to ensure the consistency of dependent artifacts across the overall process
- Higher needs for flexible repetition of the overall process or sub-processes in the case of, e.g., change requests.

To overcome the above challenges, we identify a minimal set of core constructs an PDL must provide and introduce particular responsibilities to flexibly control and manage development processes.

4.3.2 Technical Process Modeling: Core Concepts

For process modeling, almost any PDL is applicable, if it provides a sufficient set of language elements. The term "sufficient" depends on the respective application area and enterprise-specific requirements to address particular workflow issues concerning e.g., branching, synchronization, events, and exceptions (cf. [VTKB03]). According to the general process line assets, which were introduced Figure 3.4, we focus on a most relevant

subset of process modeling elements. Therefore, we, basically, require nodes and directed edges to obtain a directed graph, which serves us as workflow. Additionally, we distinguish three types of nodes: Grouping Element nodes, Method Chunk nodes, and control nodes. (Note, that these element are in line with the process line assets as presented in the meta model of Figure 3.4)

4.3.2.1 Grouping Element Nodes

GE nodes are structured process components, which are composed of other nodes, subprocesses and methods, to specify a temporal and logical order of related activities. For our purpose, we identified a minimal set of two types of GE nodes to create and manage processes efficiently.

- Practice Area: A root element to represent processes and sub-processes; it is a long-lasting period of a development process, which aggregates related practice areas, activities, MCs and control nodes. That way, structuring large processes is enabled by building hierarchies of self-contained sub-processes. Standard development phases, such as analysis, design, architecture definition, or integration are examples for long-lasting practice areas.
- Activity: A particular scope of a practice area is represented by activities. Activities mainly consist of MC nodes and represent a self-contained, time-limited workflow of related methods, which are necessary to fulfill a particular task or to produce one essential (intermediate) product. For example, the RE discipline, is an iterative and short-dated activity inside of an analysis practice area. This activity aggregates various methods, such as elicitation, documentation, or validation of requirements.

4.3.2.2 Method Chunk Nodes

In contrast to practice areas and activities, MC nodes are atomic executable units. As detailed in Section 2.4, MCs not only detail the process part of a method, but also define other MFs, which detail the product-related part of a method as well as its tool- and role-specific parts. As mentioned above, requirements elicitation is a particular method of the RE activity and is realized by an MC node.

4.3.2.3 Control Nodes

Control-flow nodes represent a set of node types, which can be used to dynamize the flow between various GE and MC nodes in different ways by enabling the construction of different control-flow patterns. Control-flow nodes also can be connected with other nodes by the means of directed edges. Although, there are more control-flow patterns available, in the following, we focus on basic control flow patterns, as initially proposed by the WfMC [WfM99b]: sequence, parallel split, synchronization, exclusive choice, and simple merge. To realize these patterns, we identified the following minimal set of required control-flow node types.

- Initial Node: To model an entry point of a process an Initial node is used.
- Final Node: To model an end point of a process a Final node is used.
- Fork Node: To model parallel flows of a process (i.e., all subsequent paths are taken) a Fork node is used.
- Join Node: To synchronize parallel process flows, which were initiated by a fork node, a Join node is used afterwards.
- Decision Node: To model an exclusive choice (i.e., exactly one subsequent path is taken) of different process flows a Decision node is used.
- Merge Node: To synchronize or merge flows initiated by a decision node a Merge node is used.

4.3.3 Process Modeling Requirements

Processes on a technical level, are designed by using aforementioned node types and standard execution semantics. However, to support the creative flow of large-scale, distributed, and parallel development processes at runtime, such processes are not sufficient to express necessary flexibility. Situational rework, adaption, change requests, iterative development, and human factors influence development processes more than other process types. Contrasting general workflow management systems and PDLs, which only support strict execution of nodes as defined by a control-flow, we distinguish Strict-managed Process Component (SPC)s and Flexible-managed Process Component (FPC)s, which serve as two essential design patterns to make development processes more flexible, when they are executed.

We split responsibilities for the execution of the overall process into strict units and flexible units and combine well-known workflow management techniques with advanced coordination capabilities. Therefore, we distinguish the correlations between practice areas with higher needs for flexible process management, from single activities or other MCs, which are ordered by a control-flow. For the latter ones, flexible coordination and control-flow management is less important, than high creativity of developers themselves. Creative and human-centric process periods do not require high-flexible coordination efforts, but require a high combination of brainpower and the application of best practices in strict order as provided by common process management systems. In contrast, long-lasting global processes must be provided with flexibility to overcome challenges mentioned above. While the realization of this separation is discussed in Section 5.6, the basic idea of Strict-managed Process Components and Flexible-managed Process Components is described in the following.

4.3.3.1 Strict-managed Process Component

The SPC pattern corresponds to strict workflow management, as known from general workflow management systems (cf. [VTKB03]). This suits the automation of short-dated,

well-known, and deterministic development activities. The respective workflows must not consider random external events, human-based design decisions or high coordination effort between several parties, but usually supports one user in doing his job in a correct order by providing situational information without to forget obligatory steps. Therefore, short-dated process components, which allow for a deterministic execution, such as activities or grouped MCs, are interpreted as self-contained parts of a superior process conventionally. An example of such an activity is the definition of an use case model in context of an analysis phase. The definition of use case, is subdivided into various sub-tasks, which have to be performed in a distinct order for efficient development.

4.3.3.2 Flexible-managed Process Component

In contrast, the FPC pattern is used to coordinate interrelated long-lasting process periods, whose execution depends on e.g., changing project situations, iterative/incremental rework, unplanned and planned change requests, concurrent development or other external and random events.
FPCs ensure flexibility by coordinating the overall process and contained FPCs depending on situational needs and coordination strategies, which are detailed in Section 5.6. Thereby, FPCs aggregate different SPCs. Based on the information of an SPC, which are monitored during their execution, and the current project situation, potentially affected FPCs can be identified and performed on demand. While on operational level SPCs are managed by a general workflow management system, an additional component is introduced to coordinate FPCs. This component evaluates information resulting from the execution of SPCs and uses the information to decide about FPCs, which are affected and have to be performed next. Each time an FPC is identified to be executed, contained SPCs are initiated by executing them on a conventional workflow management system. That way, the advantages of well-known workflow management systems are loosely coupled with additional functionality, as required for development processes. Example for FPCs, are long-lasting phases of a development process, such as the analysis and design phase of the V-Model.

4.4 Artifact Design

A variety of artifacts is produced and consumed, during a development process. Especially, to automate the processing (e.g., transformations or analyses) of artifacts, viewpoints (cf. [Com00]) and DSLs in the form of meta models are developed. Similar to viewpoints, which "encapsulate partial knowledge about the system and domain" (cf. [FKN+92]), DSLs span a domain-specific vocabulary, which covers different development concerns for the specification of artifacts to design functional and non-functional product properties on different logical levels.

The disadvantage of meta models is, that they are designed to support multiple steps of a generic development process, at the same time. This results in more and more complex meta models, which are applied for the definition of artifacts in various projects

differently. Additionally, while both methods and meta models are mostly documented informally, a combined documentation is rarely available. The more and the larger meta models exist, the more difficult it is to understand, how individual meta model elements must be applied for artifact design correctly. However, these are not the only reasons, which usually cause the discrepancy between documented process specifications and their application to real projects:

Insufficient Methodology Support

If the structure of inputs/outputs is specified insufficiently, only vague guidelines or best practices for the application of individual methods can be defined. By the means of a method-specific vocabulary or syntax, more specific guidelines would enable enhanced support of individual tasks

Missing Artifact Traceability

As modeled information is assigned with respective artifacts insufficiently, it is difficult to recognize artifact changes and associated inconsistencies or other impacts on dependent process segments. The explicit assignment of model information with relevant artifacts would enable traceability analysis and artifact-based change impact analysis across the overall process.

Information Overhead

Activities, usually, are provided with an all-encompassing model, instead of task-specific artifact information. This makes it difficult for developers to identify or extract relevant information in a chain of multiple manipulating and creating activities. By restricting models to individual situational views, the focus of developers can be shifted to relevant information.

Long-lasting Training Periods

Especially, new employees have difficulties to become familiar with new meta models. Caused by few documentation for the usage and the allocation of meta model elements on individual fields of function, a goal-oriented training period often is difficult. By the means of an explicit integration of relevant meta model information with individual tasks of the process, training periods can be reduced drastically, as more contextual information is provided.

Missing Evolution Support

Meta models, artifacts, or used methods change over the time. Through the loose coupling of these elements, such changes are hard to maintain and difficult to relate, i.e., changes of one part may have undefined side effects on other parts of the process. A process model as central integration point for artifact and meta model information would enable the evolution of the overall process as well as the enhancement of future processes.

Missing Project Impact

As artifact descriptions are mostly written in natural language, they are not machine-interpretable or analyzable. Thus, artifacts can be described, but an automated evaluation of artifacts is not enabled. Additionally, changes within artifact descriptions have no impact on the real application of artifacts within the actual process in progress. Instead, propagation of procedural changes and expected benefits is a long-lasting task. By formalizing artifacts, meta model or artifact changes can be incorporated into an actual process instance simply.

To overcome these challenges, the gap between the two engineering levels of process management and product development must be closed. Therefore, we formalize a mechanism for linking the two worlds of product-specific meta models and process-specific methods. We introduce Meta Model Views, which provide us with the capabilities to define customized viewpoints on already existing meta models. These viewpoints encapsulate partial knowledge of meta models and are fully-integrated with the process model, i.e., with product-specific MFs (artifacts) of an MC. In a second step, Meta Model Views (MMVs) are used to generate customized editors, to automate the monitoring of development activities to ensure the consistency between artifacts, and to validate the content of artifacts.

4.4.1 Meta Models & Views

In general, to formalize the product-specific information inside of process models, there must be a link between the process and a relevant meta model of the product, which has to be developed. However, in most cases, individual meta models support different processes or parts from it, e.g., MCs, at the same time. For example, the UML meta model is applicable to different development activities, such as RE and software design. As a result, the correct application of the individual parts from UML to a respective development activity depends on additional knowledge. Especially, if the meta model is intransparent and large, such as UML or the even more complex AUTOSAR, the abundance of different language elements is confusing and causes misuse and misinterpretation. Therefore, to combine these two information sources, we annotate individual MCs, i.e., associated artifacts, with extracted segments of a meta model. We call such an extract MMV.
In a nutshell, an MMV is a correlated extract of a meta model definition, which contains only method-relevant elements, i.e., classes and features of the original meta model. The idea is sketched in Figure 4.3. The original meta model on the left side provides a simplified DSL for system modeling. It defines a *Requirements Model* for documenting the *Requirements*, which are refined into *Functions* of a *Design Architecture*. Assuming that a particular development activity is the definition of *Functions* and their timing behavior in the *Design Architecture*, the corresponding MC and its output artifact requires no detailed knowledge about other classes and features. By restricting the original meta model, as indicated by the red dotted line in Figure 4.3, we remove irrelevant classes and features. The resulting MMV information is linked with an respective artifact of the process model

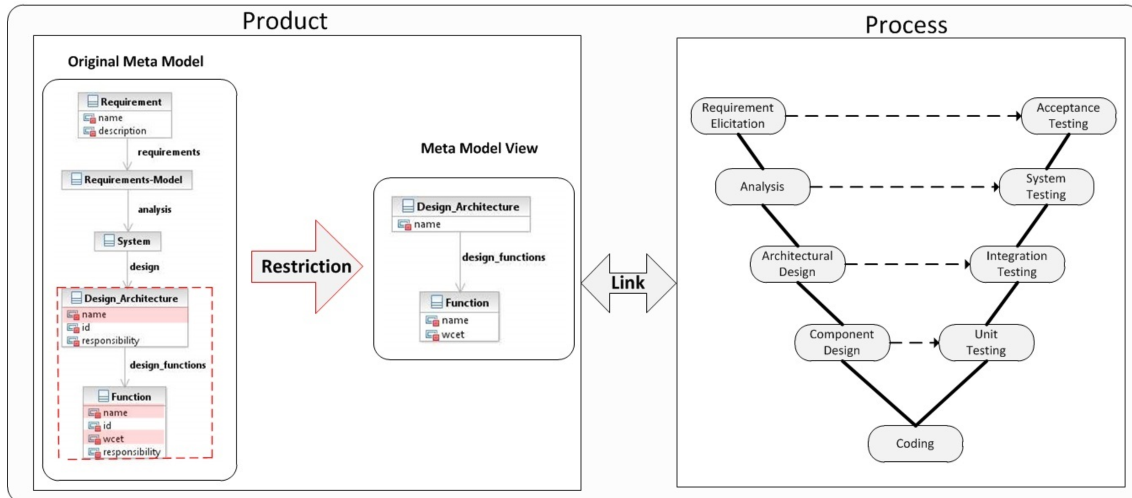


Figure 4.3: Overview: Relationship between Meta Model View and Process

(i.e., to an appropriate MF, which in this scenario would be contained in the architectural design phase), as depicted on the right side of Figure 4.3.

Before we define the linkage between a meta model information and an MC (Section 4.4.2), we formalize the concept of MMVs (Section 4.4.1.2) based on a general definition of meta models (Section 4.4.1.1).

4.4.1.1 Meta Model

Different specifications, such as MOF [OMG06b], UML [OMG11a], Eclipse Modeling Framework (EMF)'s Essential MOF (EMOF) or the associated Ecore meta model [EMF11], provide similar language constructs to define customized meta models. Therefore, we are enabled to provide a general basic definition for meta models, which is sufficient to existing realizations and to our approach, in equal shares. The following formalization is general enough, so that the definition of a meta model view and subsequent definitions based on it can be applied to arbitrary meta model formalisms.

As prerequisite for subsequent definitions, we firstly introduce some general terms. We formalize **identifiers** used for the naming of entities, **primitive data types** to provide a classification mechanism for necessary entities, and **cardinalities** to make statements about the quantity of individual entities.

Definition 1 (*Identifier*)

An identifier is a sequence of characters used to name individual entities. Let Σ be a non-empty finite set of characters. A word of Σ is a finite sequence of characters from Σ. The set of words over Σ of length n is denoted Σ^n and the set of words over Σ with

any length is denoted Σ^*, i.e.,

$$\Sigma^* := \bigcup_{n\in\mathbb{N}} \Sigma^n \tag{4.1}$$

An Identifier is defined as

$$IDENTIFIER := \Sigma^* \setminus \Sigma^0 \tag{4.2}$$

Data types are used for classification purpose to define a set of possible values, on which specific operations are defined. For meta models, data types are distinguished into primitive and complex data types to type individual features, i.e., attributes or associations (cf. definition 14). Primitive Data Types (PDTs) are generally accepted basic data types as provided by most of existing programming languages. Based on primitive types more complex composite types can be created. Complex Data Types (CDTs) or object data types are used recursively to create more complex data types. Basically, complex data types are used as domain concepts of a meta model. Therefore, they are discussed explicitly as meta modeling elements in definition 4.

Definition 2 (*Primitive Data Type*)

Let $BOOLEAN$ be a logical type, $INTEGER$ a number type, $STRING$ a character type, and $ENUM$ a selective composition of particular values typed by a Primitive Data Type. An PDT is defined as follows

$$\text{PDT} := \{BOOLEAN\} \cup \{INTEGER\} \cup \{STRING\} \cup \{ENUM\} \tag{4.3}$$

Meta models, in general, allow for the definition of numerical values or intervals to restrict the number of individual model elements. We call this definition a cardinality.

Definition 3 (*Cardinality*)

The set of cardinalities

$$CARDINALITY := \mathcal{P}(\mathbb{N}_0) \tag{4.4}$$

provides a selection of possible values and value intervals, that may be used to restrict the number of possible occurrences of a modeled entity.

Informally, a meta model is a set of interrelated Complex Data Types, aka. concepts or classes, which are grouped into a hierarchical package structure to constitute domain knowledge. CDTs are abstract entities, which collate associated features to detail domain-specific properties. Features are subdivided into a set of characteristic attribute values, which are typed with an PDT, and a set of associations to state a semantical relationships between CDTs. The following formalizes the notion of a meta model by providing general definitions and required constraints as foundation for subsequent definitions.

We start with a definition of general meta modeling elements, before we detail relevant properties.

Definition 4 (*Meta Modeling Elements*)

For meta modeling, three different main language elements can be identified, including

$$\text{an infinite set of concepts } := CONCEPTS \text{ (cf. definition 6)} \tag{4.5}$$

$$\text{an infinite set of attributes } := ATTRIBUTES \text{ (cf. definition 9)} \tag{4.6}$$

$$\text{an infinite set of associations } := ASSOCIATIONS \text{ (cf. definition 11)} \tag{4.7}$$

Accordingly, the set of available meta modeling elements is composed of three disjoint sets as follows

$$ELEMENTS := CONCEPTS \cup ATTRIBUTES \cup ASSOCIATIONS \tag{4.8}$$

For unique identification we require, that each element in $ELEMENTS$ must have a well-defined identifier.

Definition 5 (*Named Elements*)

Let e be an element of $ELEMENTS$, i.e., $e \in ELEMENTS$. Then e is associated with an identifier via the relation EN:

$$EN \subseteq ELEMENTS \times IDENTIFIER \tag{4.9}$$

So, for each $e \in ELEMENTS$, the function

$$\begin{aligned} &elementName : ELEMENTS \rightarrow IDENTIFIER \\ &elementName(e) := \{i | (e, i) \in EN\} \end{aligned} \tag{4.10}$$

determines its associated identifier, i.e., the name of the concept.

To structure domain knowledge, concepts are hierarchically composed into packages. This can be provided either by explicit package language elements of the respective meta modeling formalism, or adequate naming conventions for concepts. In the following, we apply the latter realization. That means, packages and sub-packages are uniquely identified by corresponding names or particular prefixes. Accordingly, we require the uniqueness of concepts names

Constraint 1 (*Uniqueness of concept names*)

Let c_0 and c_1 be two named concepts from $CONCEPTS$. Then the following holds:

$$\forall c_0, c_1 \in CONCEPTS : elementName(c_0) = elementName(c_1) \Rightarrow c_0 = c_1 \tag{4.11}$$

As mentioned above, concepts are CDTs of a domain to describe an abstract set of similar domain objects. In order to distinguish the infinite set of possible concepts from actual domain knowledge, we introduce the notion of modeled concepts.

Definition 6 (*Modeled Concepts*)

Let $CONCEPTS$ be an infinite set of concepts. Then we define

$$Co := \{c \in CONCEPTS | (c, i) \in EN\} \tag{4.12}$$

as the finite set of explicitly modeled meta model concepts, which is composed of all concepts, whose meta model membership and name is determined by an identifier.

For meta modeling, abstract concepts provide additional type system capabilities. Since concepts, which are indicated to be abstract, lack the instantiation capability, they only provide base characteristics, which can be extended by other concepts for a specialized usage scenario.

Definition 7 (*Abstractness*)

Let c be a modeled concept, i.e., $c \in Co$. Then the relation

$$ABSTRACT \subseteq Co \times BOOLEAN \tag{4.13}$$

associates a modeled concepts c with a boolean flag to indicate, whether it is abstract (i.e., flag is true) or not.

To access the information, the function

$$\begin{aligned} &isAbstract : Co \rightarrow BOOLEAN \\ &isAbstract(c) := b, \texttt{ such that } (c, b) \in ABSTRACT \end{aligned} \tag{4.14}$$

determines for a given modeled concept, whether it is abstract or not.

Furthermore, a modeled concept must be ensured to be either abstract or not. A mixing is not allowed. This is ensured by the following constraint 2.

Constraint 2 (*Explicit defined Abstractness*)

Let c_0 be a modeled concept, i.e., $c_0 \in Co$, then the following holds:

$$\forall c_0 \in Co : (isAbstract(c_0) = true) \veebar (isAbstract(c_0) = false) \tag{4.15}$$

Concepts define features to detail their characteristic properties and domain-specific relationships to other concepts. To express taxonomic relationships between concepts and owned characteristics, generalization hierarchies can be defined. The generalization mechanism enables a concept to inherit the features from an other concept, i.e., its super concept.

Definition 8 (*Generalization*)

To define, that a concept is super concept of another concept the relation

$$SC \subseteq Co \times Co \tag{4.16}$$

associates two modeled concepts by indicating that the first concept is super concept of the second one, i.e., the sub-concept.

In order to simply provide all super concepts of a given concept, we define a function called $superConceptsOf$:

$$\begin{aligned} &superConceptsOf : CONCEPTS \rightarrow \mathcal{P}(CONCEPTS) \\ &superConceptsOf(c) := \{c_1 | (c_1, c) \in SC\} \cup \bigcup_{s \in superConceptsOf(c)} superConceptsOf(s) \end{aligned} \tag{4.17}$$

The function not only determines direct super concepts, but also transitive related super concepts.
The inverse function of $superConceptsOf$ is $subConceptsOf$ and provides all sub-concepts of a given concept c. The function $subConceptsOf$ is defined as follows:

$$\begin{aligned} &subConceptsOf : C \rightarrow \mathcal{P}(C) \\ &subConceptsOf(c) := \{c_1 | (c, c_1) \in SC\} \cup \bigcup_{s \in subConceptsOf(c)} subConceptsOf(s) \end{aligned} \tag{4.18}$$

Moreover, we assume, that a concept is not super concept of itself. This is ensured by constraint 3.

Constraint 3 (*Anti-reflexivity of Generalization*)

Let c_0 and c_1 be two modeled concepts, i.e., $c_0, c_1 \in Co$, then the following holds:

$$\forall c_0, c_1 \in Co : (c_0, c_1) \in SC \Rightarrow c_0 \neq c_1 \tag{4.19}$$

That means, that a concept's super concept is different from itself.

A generalization hierarchy starts with the most abstract super concept and ends with the most specialized sub-concept. Along that hierarchy the concepts provide more and more differentiating features, which are inherited from super concepts to all of their sub-concepts. Features are subdivided into structural and behavioral ones, which describe dynamic and static characteristics of concepts and transitive related sub-concepts. As only structural features are used for meta modeling for most of the cases, we only focus on structural features in the following.

There are two types of structural features. The first feature type is represented by $attributes$, which are no self-contained model elements, but simple differentiating char-

acteristic values of a constitutive concept, such as its name or its identifier.

Definition 9 (*Attributes*)

Let Co be the set of modeled concepts, let $ATTRIBUTES$ be the infinite set of attributes, and let PDT the set of primitive data types. Then the relation

$$CAttr \subseteq Co \times ATTRIBUTES \times PDT \tag{4.20}$$

associates a modeled concept with a characteristic attribute and its Primitive Data Type.

For all existing tuples in $CAtttr$, the function

$$\begin{aligned} &attributesOfConcept : CONCEPTS \rightarrow \mathcal{P}(ATTRIBUTES) \\ &attributesOfConcept(c_0) := \{a_0 | (c_0, a_0, t_0) \in CAttr\} \cup \\ &\bigcup_{s \in superConceptsOf(c_0)} attributesOfConcept(s) \end{aligned} \tag{4.21}$$

determines the complete set of attributes (including inherited attributes), which are associated with a particular concept c_0.

Moreover, we assume the following three constraints to ensure the well-formedness and validity of attributes, as defined in some meta model.

Constraint 4 (*Each attribute belongs to exactly one modeled concept*)

Let a be an attribute and let c be a modeled concept. Then the following holds:

$$\forall a \in ATTRIBUTES, \exists_1 c \in Co : (c, a, t) \in CAttr \tag{4.22}$$

That means, that an attribute must not exist without a constitutive concept, i.e., each attribute must be associated with exactly one concept to describe a domain-specific characteristic.

Constraint 5 (*Uniqueness of attribute names*)

Let c be a modeled concept and let a_0 and a_1 be two attributes. Then the following holds:

$$\begin{aligned} &\forall c \in Co, \forall a_0, a_1 \in ATTRIBUTES : (c, a_0, t) \in CAttr \wedge (c, a_1, t) \in CAttr \wedge a_0 \neq a_1 \\ &\Rightarrow elementName(a_0) \neq elementName(a_1) \end{aligned} \tag{4.23}$$

That means, that in contrast to global unique concept names, we require the uniqueness of well-defined attribute names in the scope of the associated concept.

Constraint 6 (*Uniqueness of Concept Attributes*)

Let c_0 and c_1 be two modeled concepts, let a be an attribute, and let t_0 and t_1 be two

PDTs. Then the following holds:

$$\forall c_0, c_1 \in Co, \forall a \in ATTRIBUTES : (c_0, a, t_0) \in CAttr \wedge (c_1, a, t_1) \in CAttr \Rightarrow c_0 = c_1 \wedge t_0 = t_1 \tag{4.24}$$

That means, that attributes are explicitly defined via their associated concept. Additionally, this constraint enables the unambiguous identification of an attribute's PDT using the function:

$$\begin{aligned} &attributeType : CONCEPTS \times ATTRIBUTES \rightarrow PDT \\ &attributeType(c, a) := t, \texttt{ such that } (c, a, t) \in CAttr \end{aligned} \tag{4.25}$$

As we require, that attributes are only allowed to be associated with modeled concepts, we are now enabled to determine the set of all modeled attributes of a meta model.

Definition 10 (*Modeled attributes*)

Let $CAttr$ be a finite set of tuples, which relate modeled concepts with their respective attributes. Then the set At is composed of actual modeled attributes, which are used to describe a characteristic of exactly one concept:

$$At := \{a | (c, a, t) \in CAttr\} \tag{4.26}$$

Moreover, meta modeling requires the definitions of associations to model relationships between domain concepts. While associations may be allowed to connect two or more concepts at the same time, we restrict our associations definition to simple directed associations as follows.

Definition 11 (*ASSOCIATIONS*)

Let Co be the set of modeled concepts, and let $ASSOCIATIONS$ be the infinite set of associations. Then the relation

$$CAssoc \subseteq Co \times ASSOCIATIONS \tag{4.27}$$

associates a modeled concept with an association, by which associated concepts can be identified.

To receive information about domain-specific concept correlations, the function

$$\begin{aligned} &associationsOfConcept : Co \rightarrow ASSOCIATIONS \\ &associationsOfConcept(c_0) := \{r | (c_0, r) \in CAssoc\} \cup \\ &\bigcup_{s \in superConceptsOf(c_0)} associationsOfConcept(s) \end{aligned} \tag{4.28}$$

determines the set of related associations of a given concept, i.e., the set of associations

where a given concept c_0 or one of its super-concept participates as source concept.

Furthermore, additional constraints are required to ensure the validity of modeled associations:

Constraint 7 (*Each association belongs to a modeled concept*)

Let r be an association and let c be a modeled concept. Then the following holds:

$$\forall r \in ASSOCIATIONS, \exists_1 c \in Co : (c, r) \in CAssoc \tag{4.29}$$

That means, that an association must not exist without a constitutive concept, i.e., each association must be associated with exactly one concept to describe a specific domain-specific characteristic.

Constraint 8 (*Uniqueness of association names*)

Let c be a modeled concept and let r_0 and r_1 be two different associations. Then the following holds:

$$\forall c \in Co, \forall r_0, r_1 \in ASSOCIATIONS : (c, r_0) \in CAssoc \wedge (c, r_1) \in CAssoc \wedge r_0 \neq r_1 \Rightarrow elementName(r_0) \neq elementName(r_1) \tag{4.30}$$

That means, that in contrast to global unique concept names, we require the uniqueness of well-defined association names in the scope of the associated concept.

Constraint 9 (*Uniqueness of concept associations*)

Let c_0 and c_1 be two modeled concepts and let r be an association. Then the following holds:

$$\forall c_0, c_1 \in Co, \forall r \in ASSOCIATIONS : (c_0, r) \in CAssoc \wedge (c_1, r) \in CAssoc \Rightarrow c_0 = c_1 \tag{4.31}$$

That means, that we require, that associations are explicitly defined via their associated concept.

Now, the set of modeled associations is defined as follows:

Definition 12 (*Modeled associations*)

Let $CAssoc$ be a finite set of tuples, which relate modeled concepts with their respective associations. Then the set As is composed of actual modeled associations, which are used to describe a characteristic of exactly one concept:

$$As := \{r | (c, r) \in CAssoc\} \tag{4.32}$$

By now, an association is a relationship, that only defines the source of the relationship. However, as associations refer to a particular number of associated target objects in form of Complex Data Types, associations must be provided with additional properties regarding the referenced complex data type, a containment type, and the number of referenced objects. This is defined by the means of association properties, as defined in the following:

Definition 13 (*Association Properties*)

The relation

$$AssocProp \subseteq ASSOCIATIONS \times CARDINALITY \times BOOLEAN \times Co \tag{4.33}$$

refines the association information by the referenced data type, a containment type, and the number of referenced objects.

While the source concept of an association is fixed by the association itself, cf. constraint 7, an association's property relates its source concept with a target concept (the last entry of the property relation) by indicating, that the source concept has a direct relationship with the target concept. Additionally, the property links the association with a boolean flag to indicate whether it is a containment association or not. The containment or by-value aggregation is particularly important, as it identifies the parent or owner of a target instance, which implies the location of the object when persisted. If the containment flag is "true", the association's source concept is the container class of the target concept. Thereby, it is well-defined, whether an association is a containment or not. Additionally, the property links an association with a numerical value, which defines the number of target concept instances, that are allowed to participate in the relationship. The relationship between an $CAssoc$ and the $AssocProp$ is illustrated in Figure 4.4.

Finally, to ensure the unambiguity of associations, the following constraint must be fulfilled.

Constraint 10 (*Uniqueness of association properties*)

Let r be an association, let c_0 and c_1 be two modeled concepts, let m_0 and m_1 be two cardinality values, and let b_0 and b_1 be two boolean values indicating whether an association is a containment association or not. Then the following holds:

$$\begin{gathered}\forall r \in ASSOCIATIONS, \forall c_0, c_1 \in Co, \forall m_0, m_1 \in CARDINALITY, \forall b_0, b_1 \in BOOLEAN : \\ (r, m_0, b_0, c_0) \in AssocProp \wedge (r, m_1, b_1, c_1) \in AssocProp \Rightarrow m_0 = m_1 \wedge b_0 = b_1 \wedge c_0 = c_1\end{gathered} \tag{4.34}$$

To provide concept properties with general characteristics, we finally subsume above mentioned attributes and associations as so called features.

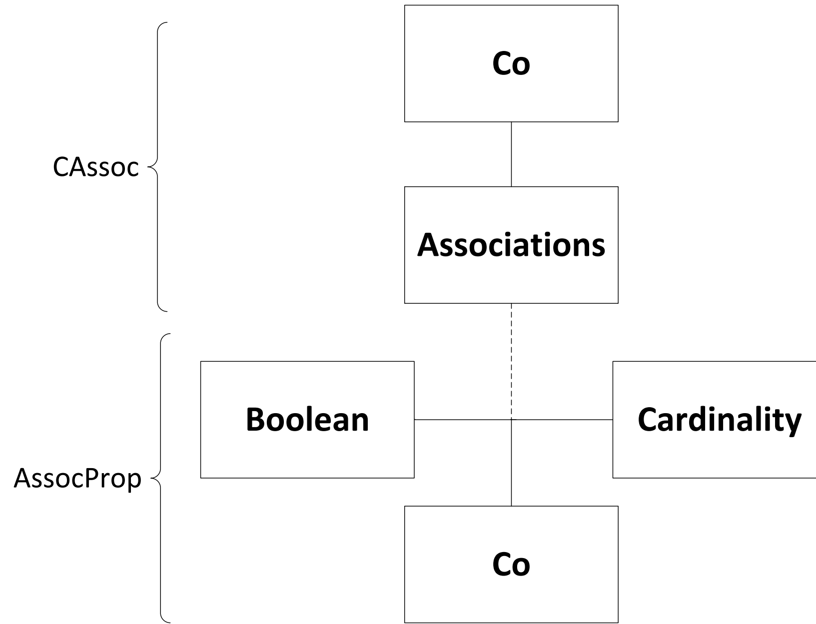


Figure 4.4: Meta model associations: Relationship between CAssoc and AssocProp

Definition 14 (*Features*)

Features are the union set of attributes and associations, i.e.,

$$FEATURES \subseteq ATTRIBUTES \cup ASSOCIATIONS \tag{4.35}$$

Based on that, the function

$$\begin{aligned}
&featuresOfConcept : Co \to \mathcal{P}(FEATURES) \\
&featuresOfConcept(c_0) = \{attributesOfConcept(c_0) \cup referencesOfConcept(c_0)\} \cup \\
&\bigcup_{s \in superConceptsOf(c)} featuresOfConcept(s)
\end{aligned} \tag{4.36}$$

determines the set of features, i.e., $f \in FEATURES$, of one single modeled concept, i.e., $c_0 \in Co$, which is composed of attributes and associations. To get the overall set of the concept's characteristics, features, which are inherited from corresponding super concepts, must be taken into account.

Based on the definitions 4 - 14 and associated constraints, a meta model definition which sufficiently fulfills necessary meta modeling capabilities is defined on the three domains of CONCEPTS, ATTRIBUTES, and ASSOCIATIONS, as follows:

Definition 15 (*Meta Model*)

Let EN be a relation of named domain elements, let SC be a relationship of concepts

and their super-concepts, let $CAttr$ be relationship of concepts and their attributes and let $CAssoc$ be a relation of concepts and their associations to other concepts, whose properties are well-defined vie the $AssocProp$ relation. Then a meta model is defined as five-tuple:

$$MM := (EN, SC, CAttr, CAssoc, AssocProp) \tag{4.37}$$

4.4.1.2 Meta Model View

Based on the meta model definition, we introduce MMVs as instrument to restrict meta models to artifact-specific needs. Before we formally define the view mechanism in detail, we shortly introduce the common indicator function, aka. characteristic function, on which our view mechanism is based.

Definition 16 (*Characteristic Function*)

A characteristic function χ for a subset A of set X, i.e., $A \subseteq X$, indicates the membership of an element from X in A, i.e.,

$$\chi_A : X \to \{0,1\}$$
$$\chi_A(x) := \begin{cases} 1 & \texttt{if } x \in A \\ 0 & otherwise \end{cases} \tag{4.38}$$

Accordingly, we define a MMV on a meta model MM as characteristic function. That function indicates the membership of an element or tuple from MM in a subset or view of that meta model. In practice, the characteristic function is defined in an user-centric task of selecting elements from the source meta model as subset for a specific view.

First of all, a characteristic function must be defined, that restricts modeled concepts of the source meta model to concepts, which are necessary for a particular view. That way, views may provide contextual concept information. For this purpose we define the Characteristic Concepts Function (CCF).

Definition 17 (*Characteristic Concepts Function*)

Let MM be a meta model, whose set of modeled concepts is Co. Let VC (ViewConcepts) be a subset of modeled concepts, i.e., $VC \subseteq Co$, which contains all modeled concepts, which are relevant for the view. Then the characteristic concept function is defined as

$$\chi_{VC} : Co \rightarrow \{0, 1\}$$
$$\chi_{VC}(c) := \begin{cases} 1 & \texttt{if } c \in VC \\ 0 & otherwise \end{cases} \tag{4.39}$$

The function associates each modeled concept of a meta model with the value 1 to indicate, that a concept is included in the view, or with the value 0 otherwise.

Regarding views, the set of relevant modeled concepts is different from the original set of concepts. Therefore, the set of view-specific modeled concepts must be determined differently based on the characteristic function χ_{VC}.

Definition 18 (*Set of View Concepts*)

Let mmv be a meta model view, i.e., $mmv = S\mid_{\chi}$, which is based on a meta model, whose set of modeled concepts is Co. Let χ_{VC} be the CCF of the view. Then the function

$$ViewConcepts : S\mid_{\chi} \rightarrow \mathcal{P}(CONCEPTS)$$
$$ViewConcepts(mmv) := \{c \mid c \in Co \wedge \chi_{VC}(c) = 1\} \tag{4.40}$$

determines the set, which consists of all concepts, which were defined for the view.

For a more detailed contextual restriction of meta model information, we also enable the restriction of a view concept's features. Therefore, we define a second characteristic function, which is called Characteristic Feature Function (CFF).

Definition 19 (*Characteristic Feature Function*)

Let MM be a meta model, where Co is the set of modeled concepts and $FEATURES$ is the set of features. Let mmv be the meta model view, which is defined to restrict MM. Let VF (ViewFeatures) be a subset of $FEATURES$, i.e., $VF \subseteq FEATURES$, which contains all features, which are relevant for concepts of the view. Then the Characteristic Feature Function

$$\chi_{VF} : Co \times FEATURES \rightarrow \{0, 1\}$$
$$\chi_{VF}(c, f) := \begin{cases} 1 & \texttt{if } (\exists m \in CARDINALITY, \exists b \in BOOLEAN, \exists c_1 \in Co : \\ & f \in attributesOfConcept(c) \vee (f \in associationsOfConcept(c) \wedge \\ & (f, m, b, c_1) \in AssocProp \implies c_1 \in ViewConcepts(mmv))) \\ & \wedge c \in ViewConcepts(mmv) \wedge (f) \in VF \\ 0 & otherwise \end{cases} \tag{4.41}$$

associates a tuple of modeled concept and its associated features with the value 1 to indicate, that a concept provides the view with a particular feature, or with the value 0 otherwise.

As we require, that the view concept's features are freely configurable from inherited and distinct features of the source concept, methods for determining the features, i.e., attributes and associations, of a view concept must be re-defined, likewise. Therefore, the view-centric functions now depend on the newly defined characteristic function χ_{FV}.

Definition 20 (*Set of View Attributes*)

Let mmv be a meta model view and let χ_{VF} be the CFF of the mmv. Then for each concept c the function

$$viewAttributesOfConcept : CONCEPTS \times S \mid_{\chi} \rightarrow \mathcal{P}(ATTRIBUTES)$$
$$viewAttributesOfConcept(c, mmv) := \{a | a \in attributesOfConcept(c) \wedge \chi_{FV}(c, a) = 1\} \tag{4.42}$$

determines all attributes from $ATTRIBUTES$ which originally are associated with a source meta model concept and indicated as necessary for the view by the characteristic function χ_{VF}.

While the above function explicitly determines view-specific concept attributes, we also re-define our function to determine concept associations.

Definition 21 (*Set of View Associations*)

Let mmv be a meta model view and let χ_{VF} be the CFF of the mmv. Then for each concept c the function

$$viewAssociationsOfConcept : CONCEPTS \times S \mid_{\chi} \rightarrow \mathcal{P}(ASSOCIATIONS)$$
$$viewAssociationsOfConcept(c, mmv) := \tag{4.43}$$
$$\{r \in R | r \in associationsOfConcept(c) \wedge \chi_{VF}(c, r) = 1\}$$

uses the characteristic function χ_{VF} to determine all associations from $ASSOCIATIONS$, which originally are associated with a source meta model concept and indicated as necessary for the view.

Based on that definitions above, we are now able to define a meta model view, as follows:

Definition 22 (*Meta Model View*)

Let $S = (EN, SC, CAttr, CAssoc, AssocProp)$ be a original source meta model, which consists of named elements, the super class relationships, and concept features with an associated set of properties. Then an MMV regarding the meta model S is defined as

$$S \mid_{\chi} := (S, \chi_{VC}, \chi_{VF}) \tag{4.44}$$

That means, a view is a triple which extends an original meta model S with two additional characteristic functions χ_{VC} and χ_{VF} to indicate the membership of relevant concepts and features in the view.

Similar to the source meta model, the restricted target MMV provides concepts and features. In contrast, the view mechanism only keeps information concerning individual concepts and a subset of associated features. For elements which are contained in the view, the identifier and the individual features, such as attributes and associations, remain the same. Consequently, the constraints 1 to 10 still hold. However, contrasting an original meta model, the inheritance mechanism is restricted, so that inherited features must be explicitly marked to be contained in an MMV (Note, that the characteristic function specifies the incorporation of features into the view, not inheritance)

For view definition, characteristic functions are essential. The function definition represents an user-centric task of the method engineer during the ME phase. In addition to the definition of artifacts, the engineers have to decide, which meta model elements have to be part of the MMV. During the MMV definition task, relevant elements of the source meta model are selected. This results in a derived meta model, which restricts domain knowledge of the original full-blown meta model to more contextual information as needed for specific tasks of a development process.

For different artifacts, which are associated as input/output of a tasks, separate MMVs are defined. To enable the combination of different views or artifacts, we define an union operation. This union operation takes one or more MMVs and combines their characteristic functions appropriately.

Definition 23 (*Union of Meta Model Views*)

Let $S_i \mid_{\chi_i}$ with $i = 1..n$ be a set of arbitrary meta model views, where S_i is an original meta model with the two associated functions χ_{VC_i} and χ_{VF_i}, by which the MMV membership of concepts and features is defined. Then the union operation

$$\bigcup_i S_i \mid_{\chi_i} := (\bigcup_i S_i, \chi_{\bigcup VC_i}, \chi_{\bigcup VF_i}) \tag{4.45}$$

combines meta model-specific relations and associated characteristic functions, i.e., the union of the characteristic sets VC and VF.

4.4.2 Process Model - Meta Model Relationship

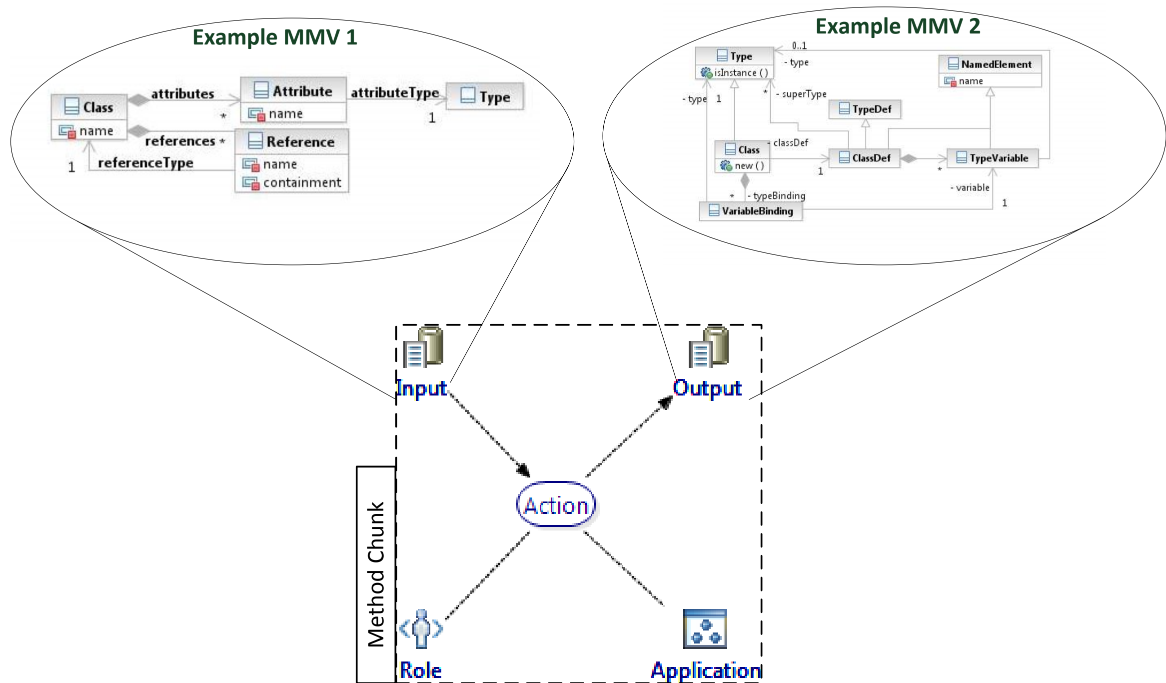


Figure 4.5: Method chunk-specific data annotation

We use MMVs, as defined above, to extend the semantics of artifacts by restricting the meta model for the situation at hand. Therefore, artifacts which were defined during the variant design activity of process line engineering (cf. Section 3.3), are now combined

with appropriate MMV information. This is exemplified in Figure 4.5, which illustrates an MC consisting of four MFs. The product-related MFs, i.e., Input and Output, are associated with a meta model view, as indicated by the rounded circles. When building more complex method chunks, which consist of multiple input and output artifacts, input-/output- parts are assembled by using the MMV union operation, automatically.
In the following, we formalize the relationship between product-oriented method fragments, i.e., artifacts, and MMVs.

Definition 24 (*Meta model Artifact Link*)

Let $Artifacts$ be the finite set available product fragments as defined in an asset repository of a process line and let $S\mid_\chi$ be a set of MMVs. Then the relation

$$ArtifactLink \subseteq Artifacts \times S\mid_\chi \tag{4.46}$$

associates a product-oriented method fragment, i.e., an artifact, with additional information of an MMV. For that relation, we require that only one MMV is linked with one specific artifact, i.e., for $artifact_1, artifact_2 \in Artifacts$ and $S_1\mid_{\chi 1}, S_2\mid_{\chi 2} \in S\mid_\chi$, the following holds:

$$\begin{aligned}
&(artifact_1, S_1\mid_{\chi 1}) \in ArtifactLink \wedge (artifact_2, S_1\mid_{\chi 1}) \in ArtifactLink \Rightarrow \\
&artifact_1 = artifact_2 \wedge \\
&(artifact_1, S_1\mid_{\chi 1}) \in ArtifactLink \wedge (artifact_1, S_2\mid_{\chi 2}) \in ArtifactLink \Rightarrow \\
&S_1\mid_{\chi 1} = S_2\mid_{\chi 2}
\end{aligned} \tag{4.47}$$

Only, if all views are associated with fragments properly, the function

$$\begin{aligned}
&MMVofArtifact : Artifacts \rightarrow S\mid_\chi \\
&MMVofArtifact(artifact) := \{S_{mc}\mid_{\chi_{mc}} \mid (artifact, S_{mc}\mid_{\chi_{mc}}) \in ArtifactLink\}
\end{aligned} \tag{4.48}$$

returns the MMV, which is associated with an artifact of an individual MC mc.

As each MC provides different number of input/output artifacts, we must provide functions to combine input and output MMVs to get an integrated view on these input/output information.
Let mc be an MC, and let inputs(mc) result in a set of input artifacts of mc, then the function

$$\begin{aligned}
&InputPart : MC \rightarrow S\mid_\chi \\
&InputPart(mc) := \bigcup_{a \in inputs(mc)} MMVofArtifact(a)
\end{aligned} \tag{4.49}$$

results in the aggregated input view, which is composed of all input artifacts of an MC, using the view union operation defined in definition 23.

Similarly, let mc be a MC, and let outputs(mc) result in a set of outputs artifacts of

mc, then the function

$$\begin{aligned} &OutputPart : MC \rightarrow S \mid_{\chi} \\ &OutputPart(mc) := \bigcup_{a \in outputs(mc)} MMVofArtifact(a) \end{aligned} \tag{4.50}$$

results in the respective output view, which is composed of all output artifact MMV information.

4.4.3 Artifact Content Propagation

Today, model-driven development uses the concept of viewpoints (cf. [{IEEE Computer Society}90]) to split models into concern-specific parts. Viewpoints enable developers to organize models according to development phases, stakeholder needs, and relevant meta model elements. However, while viewpoints are predefined by tools for most of the times, they do not provide means to realize customized artifacts of a project-specific development process. Due to the missing relationship between predefined viewpoints or model elements and an artifact, it is hardly possible to trace the evolution and dependencies of an artifact's content with regard to a development process automatically. Usually, one physically persisted model aggregates various artifacts, which are refined during a development process. Therefore, it is difficult to define the membership of individual model elements to an artifact retrospectively. This is further complicated, since even equally typed elements may belong to various artifacts. Standardized views or predefined rules, indeed, help to better organize models with regard to generic artifacts, but more often than not they are not in line with actual project needs and real artifacts. Without explicit guidelines, which define the structure of the model, explicit developer knowledge, or other means, it is impossible to identify the content of artifacts for validation or a purposeful change impact analysis.

By the means of MMVs, data schemata of artifacts are formalized at process design time. However, MMVs are not sufficient to assign model elements with artifacts unambiguously. Only in combination with the context of an MC, associated input and output artifacts enable to overcome this drawback. As discussed in Section 5.6, a workflow management system provides information about a currently performed MC, which enables the monitoring of MC-specific modeling events and the matching of affected model elements with respective artifacts, i.e., MMV information.

Based on an MC's input/output artifacts and associated MMVs, the following basic rules are obligatory to enable the assignment of model elements to individual artifacts:

- Since input artifacts are read only, by default, only output artifacts must be considered for the assignment
- Knowing the actual MC enables the recognition of newly created, modified, or deleted model elements. The matching of affected model element types with MMV concepts of an output artifact, allows an unambiguous assignment of elements to

one artifact. However, automated assignment only works for disjunct MMVs, i.e., for MMVs, which do not share an equal meta model element as concept. If output artifacts are not disjunct, additional guidelines, such as naming conventions or predefined package structures, must be provided.

The above obvious rules enable a basic assignment of data elements with artifacts using the MMVs. However, the rules neglect the assignment of elements, which belong to an artifact, while they are never used actively. To exemplify this, we sketch some scenarios in the following:

1. Figure 4.6 illustrates a process, which is composed of three tasks, i.e., MCs. Basically, the outcome of MC_n is a design architecture, whose components must be refined by timing information in MC_{n+1}. Finally, the components of the design architecture must be refined by safety-related information in MC_{n+2}. Following the above rules, we would assign all components, which are created during MC_n, with its corresponding output artifact. Subsequently, after performing MC_{n+1} based on the results of MC_n, we would expect, that an artifact, which is called *Design architecture + Timing properties*, contains architecture components and associated timing information. Due to the above rules, the artifact would lack the component information.
   All components would already have been assigned to the output of MC_n, which means, that the output of MC_{n+1} is allowed to contain timing information as well as components, which were created in MC_{n+1} only. Without additional information it would not be possible to validate timing properties of all architectural components in MC_{n+1}. Furthermore, since the second timing-related artifact does not provide architectural information, it can not be used as input for MC_{n+2}, when it's purpose is to annotate the components with safety properties. This is what we call a **propagation obstacle** for the assignment.
2. Figure 4.7 illustrates a process, which is similar, but not equal to the propagation scenario. Here the process is composed of three tasks. MC_n and MC_{n+1} each are responsible for architecture design of two different subsystems. Subsequently, they both use the same language, i.e., meta model elements. MC_{n+2} is responsible to in-

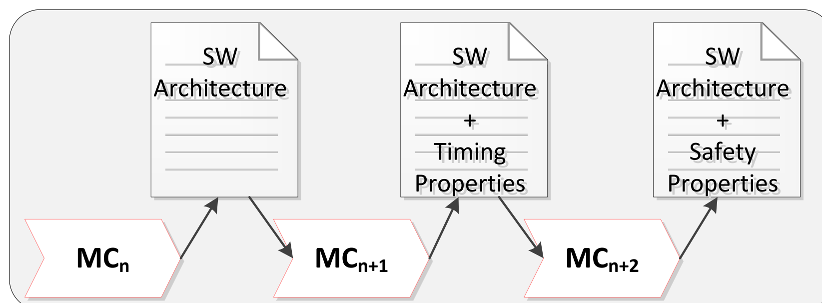


Figure 4.6: Artifact Assignment Propagation Obstacle

tegrate these two subsystems, which belong to respective artifacts of the preceding tasks. The combined artifact is intended to hold all information from both subsystems, but without modifying the subsystems during MC_{n+2}, it is undecidable, which elements belong to the integrated artifact. This is what we call a **combination obstacle** for the assignment.

3. Figure 4.8 illustrates a process, which is composed of one single task. The objective of that task is to use an abstract definition a software architecture for the definition of an architecture on more concrete level of detail. In that scenario, the content of the input artifact is isolated from the content of the output artifact, whereas both artifacts are based on the same meta model elements for the design of a software architecture. In other words, architectural elements of the input must not be assigned with the output artifact. For that reason, the above basic rules can not be extended with a rule, which generally associates similar types of the input with output artifacts. This is what we call an **abstraction obstacle** for the assignment.

The basic rules defined before, are not sufficient to face above obstacles. Therefore, to consider the so far unconsidered scenarios, we make explicit use of the relationship between input artifacts and output artifacts in the context of an MC. However, to realize this, we require the following assumption:

Except newly created elements, all elements, which are read, modified, or deleted in the context of an output artifact, must be provided by an respective MC's input artifact explicitly.

This assumption aims at the definition of a mechanism to decide about the simultaneous assignment of elements to an input artifact and an output artifact. We call this mechanism Artifact Propagation Mechanism (APM). The following example sketches the main idea of that mechanism: an undefined number of elements of type x serves an MC A as input information. The task of A is to complement input elements of type x by missing elements of the same type in a separate output artifact. While the output artifact should represent a combined view on input elements of type x and newly created ones,

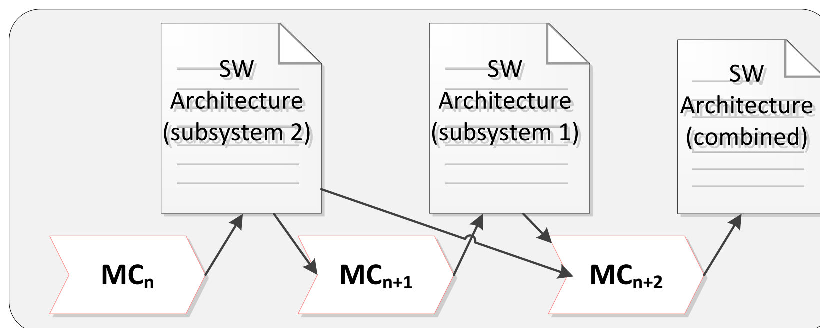


Figure 4.7: Artifact Assignment Combination Obstacle

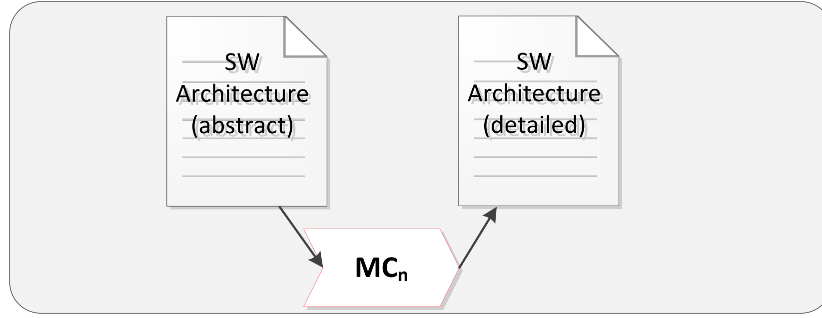


Figure 4.8: Artifact Assignment Abstration Obstacle

the above rules would only consider newly created elements. This, in particular, would cause negative side effects, if subsequent process steps would be based on the combined output artifacts, which misses some concepts. Therefore, our goal is the propagation of affected input data to output artifacts.

In a nutshell, this works as follows: In the context of each MC, all the elements of a particular type of an input artifact, which are relevant for the output artifact as well, must be known. To reach this, we extend the design of MMVs of one MC with additional information. Each MMV concept of an output artifact is linked with a list of input artifacts. The list maintains all input artifacts, which provide appropriately typed elements, which must belong to a respective output artifact, in parallel. In order to define, which concepts of an individual input artifact are relevant for an individual output artifact, the *FromTo* relationship is defined, as follows.

Definition 25 (*From-Artifact-To-Artifact relationship*)

Let Co be a set of modeled concepts (cf. Definition 6), let $S|_\chi$ be a set of MMVs, and let MC be a set of MCs, then the relation

$$FromTo \subseteq Co \times S|_\chi \times MC \times \mathcal{P}(S|_\chi) \tag{4.51}$$

relates a concept, which is contained in a particular MMV of an output artifact of an MC mc, with a set of input artifacts of mc, i.e., associated MMVs. As a result, the defined relationship identifies the input artifacts and particularly typed elements, which must be assigned with the output artifact.

Let mc be an MC, let $input_1 \cdots input_n$ be the input artifacts of mc whereof rel is a subset of MMVs, which are associated with the input artifacts of mc, i.e.,
$rel \subseteq MMVofArtifact(input_1) \cdots MMVofArtifact(input_n)$. Let $output_i$ be an output artifact of mc, where $ommv_i = MMVofArtifact(output_i)$. Let c be a concept, which was defined for $ommv_i$, i.e., $c \in ViewConcepts(ommv_i)$.
Then $(c, ommv_i, mc, rel) \in FromTo$ indicates, that already assigned model elements of type c of an artifact from rel must be assigned to $output_i$ likewise.

Based on this, the function $CopyFrom$ is defined.

Definition 26 (*CopyFrom*)

Let Co be a set of modeled concepts (cf. Definition 6), let $S|_\chi$ be an MMV, and let mc be an MC, then the function

$$copyFrom : Co \times MC \times S|_\chi \rightarrow \mathcal{P}(S|_\chi)$$
$$copyFrom(c, mc, ommv_i) := \{rel|(c, ommv_i, mc, rel) \in FromTo \wedge c \in ViewConcepts(ommv_i)\} \quad (4.52)$$

results in a set MMVs, which formalize related input artifacts of mc, whose already associated elements of data type c are relevant for the output artifact $ommv_i$ of the same MC.

The *FromTo* relationship enables us to decide about which elements must be propagated from input artifacts to a specific output artifact during the execution phase of the process. After we described the design in this section, the application of that information for an automated assignment of elements is discussed in Section 5.4.

4.5 Method-specific Editor Design

To support development methods, appropriate editor capabilities must enable the method-specific visualization, modification, and creation of output information, based on individual input information. In parallel, editors must consider the situation, that methods and processes evolve over time. To face these two requirements, we aim at a model-driven generation of method-centric editors from process models and contained MCs.
To reach this, MC knowledge is extended by appropriate design information. In contrast to conventional editor generation approaches, which are provided by frameworks, such as EMF or Xtext, method-centric editor design does not focus the realization of a meta model in general, but aligns it according to methodological needs. To address relevant information items, i.e., objects of model under development, and to provide situational editor capabilities, a different design technique is needed.
Therefore, method-centric editor design mainly encompasses three main activities: in Section 4.4, we already detailed MMVs as a binding mechanism, in which relevant meta model information are combined with input and output artifacts of MCs. As a result, relevant type information is provided to restrict an editor's capabilities to individually typed elements. Based on that, we specify the arrangement and labeling of bound elements in the context of an editor. Finally, we define the way of processing individual model elements by making their usage scenario explicit. The following details the theoretical background and provides general design rules, which are necessary to enable the method-centric editor design.

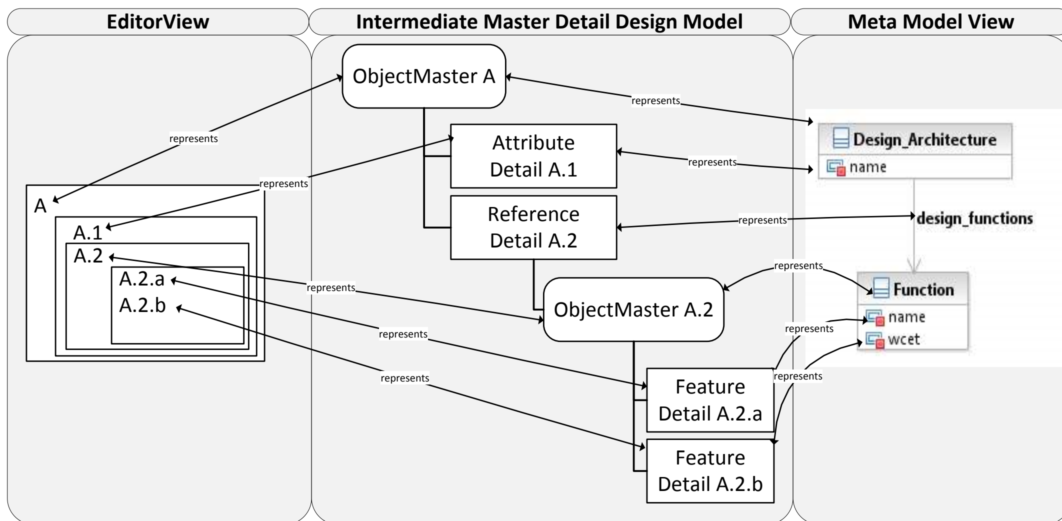


Figure 4.9: Master-Detail Meta Model Re-arrangement

4.5.1 Editor-specific Meta Model View Re-arrangement

Based on the established link between a meta model and a MC's artifacts, the elements of the annotated MMVs must be re-arranged according to methodological requirements of an editor. For this, we separately unify the method's input and output artifacts, i.e., its MMVs, to get a comprehensive view on relevant input and output information, as detailed in definition 23. For each aggregated view, a customized editor is designed and generated following the master-detail pattern (cf. [SN09]). The resulting editors subsequently are combined into one method-centric editor, whose input information can be distinguished from output information.

Concerning the master-detail pattern, a *master* section represents a set of root objects, for which relevant features, i.e., attributes and associated objects, are handled in a separate *details* section. This separation makes objects tangible and enables us to visualize an element for editing associated features. To enable a method-specific navigation of modeled objects, by which individual associated features and objects can be hidden on individual levels, relevant navigation paths between explicit typed objects are realized by respective interleaved master-detail sections. That means, a details section, which details associated objects of the master object, serves as master section for subsequent details. As a result, a tree-based hierarchy of nested sections is assembled, as illustrated in the center of Figure 4.9. To design an editor following the master-detail pattern, we re-organize aggregated input/output MMV in a Meta Model View Tree (MMVT) according to the following Backus Naur Form (BNF) (cf. [Nau63]).

$$
\begin{aligned}
\langle \text{ObjectMaster} \rangle &\models \textit{MMVConceptInformation}\ \langle \text{FeatureDetail} \rangle \\
\langle \text{FeatureDetail} \rangle &\models \langle \text{FeatureDetail} \rangle\ \langle \text{AttributeDetail} \rangle\ \mid \\
&\quad\ \langle \text{FeatureDetail} \rangle\ \langle \text{ReferenceDetail} \rangle\ \mid \bot \\
\langle \text{ReferenceDetail} \rangle &\models \textit{MMVAssociationInformation}\ \langle \text{ObjectMaster} \rangle \\
\langle \text{AttributeDetail} \rangle &\models \textit{MMVAttributeInformation}
\end{aligned}
$$

Starting point of every MMVT is an Object Master (OM) node. An OM node represents a master section to visualize instances of a particular MMV concept, i.e., a meta model class, and references one or more Feature Detail (FD) child nodes. FDs are split into Reference Details (RDs) and Attribute Details (ADs), and establish the information source of the OM's details section. An RD node represents an MMV association, by which other classes can be referenced as children OM nodes according to the respective MMV and the abstract syntax of an underlying meta model. An AD node represents an MMV attribute, by which attributes of a class are addressed.

As depicted in Figure 4.9, a link between the MMV and MMVT is established, by annotating MMVT elements (OM, RD, AD) with their MMV counterpart (Concept, Association, Attributes). That way, the MMVT not only provides structural editor information, but also sufficient information, which enables the generated editor to process, i.e., to identify and to extract individual model elements of a Model Under Development (MUD): An OM node references a main object type, which serves as entry point for the identification of referenced features. An RD feature is either a containment reference, i.e., a composition of elements, which are destroyed when the referencing object is destroyed, or it is an aggregation reference, i.e., a simple part-whole relationship between objects without side effects. In each instance, the referenced MMV association information of an RD provides information about the referenced object type. As this involves super class types likewise, the children OM node of an RD node must make the referenced object type explicit to enable a type-specific treatment of associated objects. Therefore, for each object type, which must be referenced as feature in a *details* section of a particular OM node, a separate RD node, which references an explicitly typed OM node, must be introduced. In contrast, object attributes are referenced from an OM node by using an AD node, which are leaf nodes typed with a respective primitive data type.

In addition to type information, the MMV association information of an RD node references the cardinality value, which indicates how many object instances may be related with the OM. This is relevant for the transformation of hierarchic master-detail design structures into a platform-specific target language. For a cardinality with a maximum of one, one single object must be handled within the respective editor section. Otherwise, a list of objects of the respective type must be handled, which impacts the used target

language constructs of the generated editor.

In summary, to define the structure of the target editor, we re-organize the flat structure of an MMV $S\ |_\chi$ (cf. Section 4.4) into a tree-based structure, for which the following rules must be ensured:

- OM nodes are linked to a view concept C, i.e., $C \in ViewConcept(S\ |_\chi)$
- An AD node, whose next parent OM node is linked with type C, only can be linked with one of its associated attribute features a, i.e., $a \in viewAttributesOfConcept(C, S\ |_\chi)$
- An RD node, whose next parent OM node is linked with type C, are linked with one of its associated association features r, i.e., $r \in viewAssociationsOfConcept(C, S\ |_\chi)$
- Two OM nodes must only be connected via an RD node, if this is valid regarding to the underlying meta model information. Let om_{parent} be an OM node, which is associated with an MMV concept C_{parent} and let rd be an RD node, which is associated with an MMV association r and defined as feature of C_{parent}, i.e., $r \in viewAssociationsOfConcept(C_{parent}, S\ |_\chi)$. Let rd be a child node of om_{parent}. Then for each child OM node om_{child} of rd, the referenced MMV concept C_{child} must correspond to the targeted class type of r, or one of its super classes, i.e., :

$$C_{parent} \in ViewConcept(S\ |_\chi) \wedge r \in viewAssociationsOfConcept(C_{parent}, S\ |_\chi) \wedge$$
$$(r, x, y, C) \in AssocProp \implies C_{child} \in ((C \cup superConceptsOf(C)) \wedge ViewConcept(S\ |_\chi))$$

The created tree-structure is best qualified for the design and persistence of editor information. Additionally, its platform-independent and generic characteristic enables the generation of various editor layouts. For example, as presented in the left part of Figure 4.9, the structure of an MMVT can be realized as nested composites, by which master and detail elements are edited. Composite-specific code templates, which are used to generate working areas for particular elements, as detailed in Section 5.3, can simply be replaced by other templates.

4.5.2 Element Usage Scenario Definition

Method-centric editors not only are required to visualize artifact elements, but also to provide capabilities, by which elements can be modified or created, while considering methodological needs. To realize this, the MMVT is complemented by information about how to process individual model elements and associated features, i.e., attributes and associated objects.
For these elements, we defined the notion of an Element Usage Scenario Attribute (EUSA) to annotate one of the following basic editor capabilities with an MMVT element: *create*,

modify, and *read-only*. *create* enables the editor to instantiate or to delete/remove object instances, *modify* allows to modify a particular feature, i.e., an attribute value or a referencing list of objects, and *read-only* visualizes a feature without the possibility for modification. The concrete EUSA semantics and its influence on the later editor depends on the respective MMVT node type and its context or parent node, as depicted in Table 4.1.
An EUSA either is defined for attributes, i.e., children nodes of OM nodes, which are AD nodes, or referenced objects, i.e., children nodes of RD, which are OM nodes. In contrast, since no statement is given about the concrete referenced object type, an EUSA is not defined for RD nodes, i.e., if different types or sub-types of a class are enabled by an association, a type-specific distinction of EUSAs is enabled. Finally, the root object is *read-only* by default.

As discussed earlier, an RD node either represents a containment association or an aggregation association, which influences the possibilities to annotate an referenced child OM node differently:
If the RD node is related to a containment association, potential EUSA values of an OM child node are *create* or *read-only*. The EUSA *create* enables the modification of the referenced object list, i.e., new objects can be instantiated and added to the list or already contained objects can be deleted and removed from the list. On the other side, *read-only* effects the visualization of contained unmodifiable objects and enables the selection of a navigable object as master section for a subsequent *details* section.
If the RD is related to an aggregation association, i.e., associated objects can not be created, a child OM node can be annotated either with *modify* or *read-only*. The EUSA *modify* implies, that the editor enables the adding/removal of already existing referenced objects to/from the reference list. Similar to the containment case, *read-only* implies an unmodifiable list of referenced objects, which can be processed in a subsequent *details* section.
Since attributes are instantiated using a default value, when their containing object is created for the first time, AD nodes are handled differently. Attributes either are unmodifiable, i.e., *read-only*, or they can be modified as part of the modification of some object instance, i.e., they are annotated with *modify*.

Besides the usage scenario, which concerns editor capabilities, the MMVT must pro-

Context		create	modify	read-only
-	Object Master	-	-	x
Object Master	attribute	-	x	x
Object Master	reference	-	-	-
Object Master	containment	-	-	-
Containment	Object Master	x	-	x
Reference	Object Master		x	x

Table 4.1: EUSA Annotation Alternatives

vide labeling information to visualize objects, which commonly are composed of more than one feature. For most of the cases, this information is managed by a *Label Provider*, which processes an explicit feature as representative of the object. Therefore, we require, that each OM node at least provides one Attribute Node (AN), which is indicated as object label to enable a later *Label Provider* generation.

Considering the MMV, which was defined in definition 22, the annotation of the EUSA information is formalized as follows:

Definition 27 (*EUSA Basic Structure*)

An EUSA relates a data element on type level with an usage scenarios, i.e., $create$,$modify$, $read$-$only$, and an additional boolean flag, which indicates, whether or not the data element, i.e., a feature, provides label information. This is realized by the following tuple:

$$EUSA \subseteq \{create, modify, read - only\} \times BOOLEAN \tag{4.53}$$

As an EUSA is associated with concepts and features of an MMV, the following formally defines these two relationships. We start with a definition of the relationship between a concept and a respective EUSA.

Definition 28 (*EUSA Annotation with Concepts*)

Let Co be a set of modeled concepts, let ev be an EUSA value ($ev \in EUSA$), let MC be a set of method chunks, and let $S|_\chi$ be a set of MMVs, then the relation

$$EUSAtoConcept \subseteq Co \times MC \times S|_\chi \times EUSA \tag{4.54}$$

relates each concept of an individual MMV of an MC's artifact with an EUSA.

Consequently, to query a given concepts for its EUSA, the function *getEUSAFromConcept* can be defined as follows:

$$\begin{aligned}
&getEUSAFromConcept : Co \times MC \times S|_\chi \rightarrow EUSA \\
&getEUSAFromConcept(c, mc, ommv_i) := \\
&\{ev|(c, ommv_i, mc, ev) \in EUSAtoConcept \wedge c \in ViewConcepts(ommv_i) \\
&\wedge \exists_1 artifact \in \{inputs(mc) \vee outputs(mc)\} : ommv_i == MMVofArtifact(artifact)\}
\end{aligned} \tag{4.55}$$

Given a concept c, which is contained in an MMV $ommv_i$ of an MC's (mc) output artifact. The function results in the EUSA, which is specified for concept c.

Finally, a definition of the relationship between a concept's feature and a respective EUSA is given.

Definition 29 (*EUSA Annotation with Features*)

Let FEATURES be a set of features (cf. Definition 14), Co be a set of modeled concepts (cf. Definition 6), let ev be an EUSA value ($ev \in EUSA$), let MC be a set of method chunks, and let $S|_\chi$ be a set of MMVs, then the relation

$$EUSAtoFeature \subseteq FEATURES \times Co \times MC \times S|_\chi \times EUSA \quad (4.56)$$

relates a concept's feature of an individual MMV of an MC's artifact with an EUSA

Consequently, to query a given feature for its EUSA, the function *getEUSAFromFeature* can be defined as follows:

$$\begin{aligned}
&getEUSAFromFeature : FEATURES \times Co \times MC \times S|_\chi \rightarrow EUSA) \\
&getEUSAFromFeature(f, c, mc, ommv_i) := \\
&\{ev|(f, c, ommv_i, mc, ev) \in EUSAtoFeature \wedge f \in featuresOfConcept(c) \\
&\wedge c \in ViewConcepts(ommv_i) \\
&\wedge \exists_1 artifact \in \{inputs(mc) \vee outputs(mc)\} : ommv_i == partitionOfArtifact(artifact)\}
\end{aligned} \quad (4.57)$$

Given a feature f of a concept c, which is contained in an MMV $ommv_i$ of an MC's (mc) output artifact. The function results in the EUSA, which is specified for feature f.

4.6 Guideline Design

Previously, we introduced MMVs to combine formal product information with a process-centric MF into one MC. Now, that the contextual knowledge about input and output artifacts of an MC is defined, it can further be used for the specification of Situational Method-centric Guidelines (SMCGs), which must be observed for individual development activities. Typical examples for guidelines guide the design of, e.g., UML class diagrams (cf. [Amb05]), as follows:

1. Indicate visibility only on design models.
2. Name interfaces according to language naming conventions
3. Always indicate the multiplicity of an association
4. Do not model $Implied$ relationships

Unfortunately, such guidelines are preferentially available as natural language text and are hard to match with individual activities of a development process. In contrast to general guideline repositories, where the assignment of that guidelines to a relevant application scenario is difficult, we provide guidelines with a computer-interpretable semantics and explicitly combine them with their application scenario, i.e., the MC. Therefore, we re-use annotated MMVs for the definition of rules, which are evaluated as guide-

lines or best practices in the context of an individual MC, or for the derivation of other helpful information for reaching a particular objective.

In the following, we first discuss typical guideline characteristics. Afterwards, we detail the requirements, which led us to the need for introducing a guideline formalism, which faces the needs of method-centric and executable guidelines. Finally, the main part of this section introduces a meta model and design principles for the design of Situational Method-centric Guidelines.

4.6.1 Guideline Characterization

"There exist various general strategies to help guide the [development] process. In contrast with general strategies[, such as divide-and-conquer or iterative and incremental development], methods are more specific in that they generally suggest and provide a set of notations to be used with the method, a description of the process to be used when following a method, and a set of guidelines in using the method." [AMB+04]

Accordingly, a guideline is an optional part of an MC to provide means in executing some method. On a more detailed level, a guideline can further be defined as "..a specific type of guidance that provides additional detail on how to perform a particular task [..], or that provides additional detail, rules, and recommendations on work products and their properties. Among others, it can include details about best practices and different approaches for doing work, how to use particular types of work products[..], discussions on skills the performing roles should acquire or improve upon, measurements for progress and maturity, etc. " [OMG08a]

The imperative or recommending characteristic of a guideline is similar to a rule, which restricts (or rectifies) development tasks. Due to this similarity, we were inspired by the rule classification schema of [Mil10] to identify the following five types of guidelines:

- Integrity guidelines are known as (integrity) constraints or rules. There are deontic integrity guidelines to express mandatory obligations and alethic guideline to express optional regards. Both types consist of a constraint assertion, which is a sentence in a logical language, such as first-order predicate logic or OCL. For example, the wake-up time of a SW-component **must** be lower than 10 milliseconds is a **deontic** integrity guideline.
- Derivation guidelines are rules, where conditions result in conclusions, which can be used for subsequent evaluations. For example, there is a safety-critical system, **if** the wake-up time of a SW-component must be lower than 10 milliseconds.
- Reaction guidelines answer to individual events by the application of particular actions. They consist of a mandatory triggering event expression, an optional condition, and a produced action or a post-condition (or both). An example of a process-related reaction rule is, when an artifact was changed in a first task, and this artifact is used by a second task, then the product of task two must be checked for consistency, i.e., it should be repeated.

- Production guidelines have a condition and a produced action, where condition is a logical formula. By the means of this guideline type derivation or reaction guidelines can be expressed alternatively. Regarding artifacts these guidelines may require the creation of obligatory output information implied by individual input information. For the focused model-based artifacts, this may concern:
  - creating new classes initially or the decomposition/refinement of other classes
  - setting individual attributes of available classes
  - setting individual associations of available classes

  While production guidelines can basically be used to guide human tasks by recommending conditional actions, they can be used as model transformation guidelines to adapt or prepare output artifacts, automatically.
- Transformation guidelines define the change of state of individual model elements. During the software development process, individual artifact elements are used and possibly adapted by different activities. To ensure individual states of the entire system or subsystems, allowed state changes could be described by state transformation guidelines. For example, the wake up time of an SW-component may be decremented but not incremented.

In this thesis, we focus on the above guideline types, which describe different application scenarios of what a guideline accomplishes and how a guideline works. However, a second dimension of guidelines is, that they all share an equal set of characteristics. From our point of view, regardless of its type, each guideline is characterized by four interrelated basic facets, as illustrated in Figure 4.10. Similar to an MC, which is characterized by various fragment types, a *product*-oriented facet of guideline characterizes data entities, for which the guideline is relevant for. They are further characterized by a *process*-oriented facet, which define the scope of a guideline's application from a procedural and temporal perspective. In parallel, guidelines are also influenced by a *role* or human, which has to observe the guideline. Finally, a guideline is characterized by the *effect* of its application, which either influences the product-oriented facet (i.e., the product itself) or the further course of development (i.e., the process-oriented facet). In summary, a guideline specifies, that something (What?) has to be analyzed in a specific situation (When?), provides somebody or something with relevant information (Who?), and causes an individual change (How?). While MCs compose method knowledge, guidelines re-organize the available knowledge from MFs to compose rules, which guide developers in using method knowledge. These facets are detailed in the following.

4.6.1.1 Product-centric Facet

Primarily, a guideline is characterized by the affected model or product. A guideline affects static and dynamic model qualities, whereof dynamic qualities can further be subdivided into qualities regarding the abstract syntax and the static semantics.

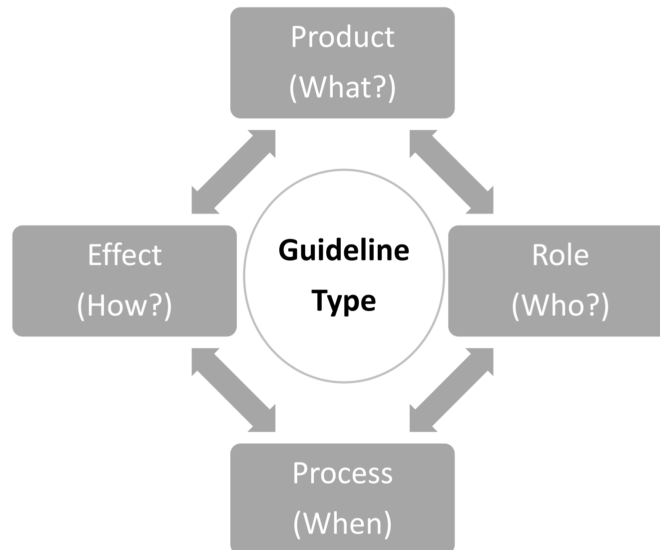


Figure 4.10: Guideline Facets

The **validity** of a model's abstract syntax is mainly ensured by the means of a meta model. As mentioned in Section 2.2, meta modeling deals with the formal specification of (modeling) languages by the means of concepts, associations and necessary construction rules, determining e.g., that only one association is allowed between class *A* and class *B*. Similar to the syntax analysis, as used in compiler construction [VSSU06], parsers automatically check whether a model is in accordance with the grammar and contained construction rules, i.e., the meta model.

While abstract syntax mainly concerns the minimal set of obligatory production rules of meta models, static semantics concerns the **correctness** of models and provides global restrictions with regard to effectively modeled elements and their relationships. For example, identifiers of individual model elements have to be distinct, or a concept must have an appropriate number and type of attributes. Such restrictions are hard to express using standard syntactic constructs, and are normally realized by the means of an additional/extensional formalism, which provide the overall meta model with a set of validation rules. As shown in [SB10], for behavior models, especially, an abstract interpretation is useful to gain information about static semantics concerning e.g., control or data flow by a partial execution of the model, i.e., without performing all calculations.

In contrast to static information, dynamic semantics concerns **integrity** of a model and focuses the behavior of a product at runtime, i.e., how an instantiated model transforms random input data via conditionally executed statements. For this reason, dynamic semantics more concerns product requirements, which must be validated using accepted testing strategies, than method-centric modeling guidelines or best practices.

While checking the abstract syntax is a standard function of todays' modeling envi-

ronments, checking dynamic semantics requires runtime data, which are not available at design time. Hence, they are both out of scope for this thesis.
In contrast, checking the static model semantics is a more crucial task, which can be enhanced by providing Situational Method-centric Guidelines. Therefore, we focus on the definition of validation rules for models under development from the static semantics point of view. As a result, we enable to enhance the model quality by providing a method-centric interpretable rule set (cf. Section 4.6.3) of restrictions and best practices concerning the creation, modification and adjustment of models and contained elements.

4.6.1.2 Process-centric Facet

Beside the product-oriented facet, guidelines can be characterized with regard to procedural and temporal properties. While the procedural property refers to the impact of an evaluated guideline on dependent process scopes, such as GEs and MCs, the temporal property refers to the impact of a guideline's evaluation time. For development process, the scope and the evaluation time of a guideline are important for the interpretation and the effect of a guideline.

Guideline Scope Guidelines are basically defined in the context of one MC to guide a development task. However, by composing various MCs into large-scale GEs, larger scopes of guideline application are created. While the scope of guideline application in the context of one MC clearly is defined by associated MFs, the scope of a guideline in the context an GE is defined by the sum of all contained MCs and their associated MFs. In contrast to MC guidelines, an GE guideline must hold for all contained GEs and MCs, likewise. For evaluation, an GE guideline can be evaluated once for the overall GE, or each time a contained MC guideline is evaluated. In the latter case, the set of MC guidelines is merged with superior GE guidelines.

Guideline Evaluation Time Basically, a guideline can be evaluated **before**, **after**, or **while** an activity is executed, whereas each alternative may influence method's execution differently.
If a guideline is evaluated **before** executing the activity, the correctness and completeness of all input information is validated. Based on a positive result, the activity is allowed to be performed. A negative evaluation result, though, implies, that preceding activities have produced incorrect or inconsistent output and must be repeated before the actual activity can be started.
In contrast, if a guideline is evaluated **after** the activity was performed, information, which was newly produced in the course of the respective activity, can be ensured to be valid for sub-sequent activities. Otherwise the activity immediately can be repeated to fix potential conflicts.
Guidelines, which are validated **while** the activity is performed, are triggered each time a single action was performed by a developer. This type of so-called "*online*" evaluation

triggers the overall set of specified guidelines and provides the disadvantage of uncomfortable information overload.

4.6.1.3 Effect-centric Facet

Guidelines are rules or policies, which are evaluated to accomplish an individual effect. If a condition is evaluated positively, i.e., a guideline is fulfilled and no conflicts were detected, the workflow can be continued normally. Otherwise, various strategies can be initiated. Depending on its severity, a detected guideline violation may influence:

- the further proceeding of the workflow. For example, by repeating the actual method or initializing a recovery workflow.
- the product itself. For example, by adapting the product automatically or manually.

In general, restricting, activating, or proposing human or automated activities are possible.

4.6.1.4 Role-centric Facet

Finally, we mention the role or human specific evaluation semantics. Since activities are normally executed by different developers, guidelines can also focus their individual skills, knowledge, permissions, and (technical) language. Although, such specializations are conceivable, the following will not consider the role-specific evaluation semantics in more detail.

4.6.2 Requirements for Situational Method-centric Guideline Design

Today's model validation formalisms are neither process- or method-centric nor support the definition of guidelines according to above facets. Instead, they provide an overall meta model with validation capabilities in a textual or script-based way. Indeed, such formalisms provide model validation mechanisms, but also generic and non-situational solutions. These solutions are normally global to one modeling environment and are not connected with any process model, which leads to hundreds of tasks covered by one single rule set. Such a global rule set is always active and can not be restricted to relevant tasks. That way, also irrelevant rules must be checked, by what falsified or confusing information under-/over-load is provided to the developers. Especially, management and side-effect-free adaption of such universally valid guideline sets is difficult without knowing underlying meta models and affected methods in detail.

To face that challenges, the following introduces an approach to eliminate drawbacks of today's validation languages, and to enable the definition, adaption, and management of Situational Method-centric Guidelines. Therefore, our approach must provide essential capabilities of a model validation language, i.e., :

- Mechanisms to navigate models and comprised elements for identifying the elements to validate
- Mechanisms to distinguish classes and respective instances
- Mechanisms to express conditions on sets of model elements
- Mechanisms to express complex boolean expressions on instance-specific model features

Additionally, in [DKH08] the authors state, that there are four essential questions, which any guideline framework must answer. Therefore, we first discuss these questions, before we introduce the guideline meta model in the next section.

1. How to share and represent various types of guidelines using a formal unambiguous representation?
   Section 4.6.3 introduces the guideline meta model, which is represented by an intuitive graphical concrete syntax in contrast to common text or script based syntax. And as modeled guidelines will be fully integrated with the process model via the aspect-oriented extension mechanism, it can also be shared between different parties by the means of a central process model.
2. How to acquire, verify, localize, execute and evaluate formalized guidelines and support systems in daily practice?
   Inspired by the idea of a continuous process improvement, our guidelines are intended to be developed in an iterative way, i.e., guidelines are (re-)designed and checked syntactically during the method engineering phase. During the process execution time, they can be localized by the means of corresponding relevant tasks which are under control of a process engine as shown in chapter 5. While the model, which has to be validated, is determined by actual variables of a running process, a concrete evaluation depends on the selected target evaluation language, which is generated from our guideline model. This will be shown in detail in Section 5.5.
3. How to interface guideline-based decision support systems with external meta model data?
   Guidelines are strongly connected with the the process model and other associated information, which enable sound decision making. The automated processing of development process models, which is discussed in chapter 5, enables processing of meta model information, which are annotated in form of an MMV, and the evaluation of guidelines.
4. How to provide decision support to developers in daily practice?
   Our system does not aim at suppressing developers' creativity. It rather supports developers in finding appropriate work units based on the underlying process. Therefore, while developers may further use familiar tooling environments applying familiar practices, each task is allocated by a process management system, which validates the modeled output subsequent to a finalized modeling task. (cf. Section 5.5) Thereby, the set of annotated validation rules or guidelines is evaluated to decide about proceeding the process. As a result, most relevant errors or problems can be recognized and consequently avoided where they occur.

4.6.3 The Generic Guideline Meta Model

Similar to MMVs, the guideline meta model, which is introduced in the following, is linked with a process model by using the aspect-oriented mechanism (cf. Figure 4.2). That way, MCs can simply be extended by guidelines and best practices. The aspect, called *MCGLAspect*, extends task-specific MFs with an aspect instance annotation, called *MCGuidelineAnnotation*. That annotation references a guideline model (*GLModel*), which details best practices and guidelines only in the context of one specific MC. Although, this section focuses on MC-specific guidelines, the aspect simply can be adapted to cover GEs, as well.

Although, there are well-known formalisms to validate an MUD, such as policy- or constraint-based languages, in the following, we introduce a guideline meta model to demonstrate the capabilities of an approach, which is integrated with the process model. In contrast to, e.g., OCL or EVL, the meta model enables a procedural combination of various statements (constraints) to be evaluated and conditional actions into a complex guideline. To focus the actual process context of an MC, complex guidelines use artifact information and combine different interacting statements either to initiate individual actions or to influence the course of a development process.

The guideline meta model is composed of two parts, which enable the design of complex guidelines from a set of more simple guidelines. Figure 4.11 depicts the main part of our guideline meta model. It, basically, consists of edges and different node types to define the process of a complex guideline. While most of the nodes are used for process definition, a *StatementNode*, which is detailed in Figure 4.13, is a particular node, which serves as a simple guideline, whose result can be used by other *StatementNode* nodes.

4.6.3.1 Abstract Syntax for Complex Guidelines

A complex guideline enables the definition of a logical ordered sequence of various statements and associates them with conditional reactions. In contrast to OCL or EVL, where the context of a statement is defined by one element and one result, complex guidelines combine the results of various statements in the context of different elements of the MUD.
Basically, each element of the guideline meta model, which is depicted in Figure 4.11, is a *ModelElement*, which provides a name and an unique identifier. A guideline model (*GLModel*) is linked with a scope, i.e., a MC, and aggregates multiple *Guideline*s, which consist of nodes (*GLNode*) and edges (*GLEdge*).
GLEdges are used to associate a source *GLNode* with a target *GLNode*, by which a directed guideline graph logically orders *GLNode*s. The *GLEdge*s propagate the result, which was evaluated in the source node, to the target node for subsequent processing and to enable the design of **derivation guidelines**.
GLNodes are more complex than the *GLEdge*s. Basically, there are three types of nodes:

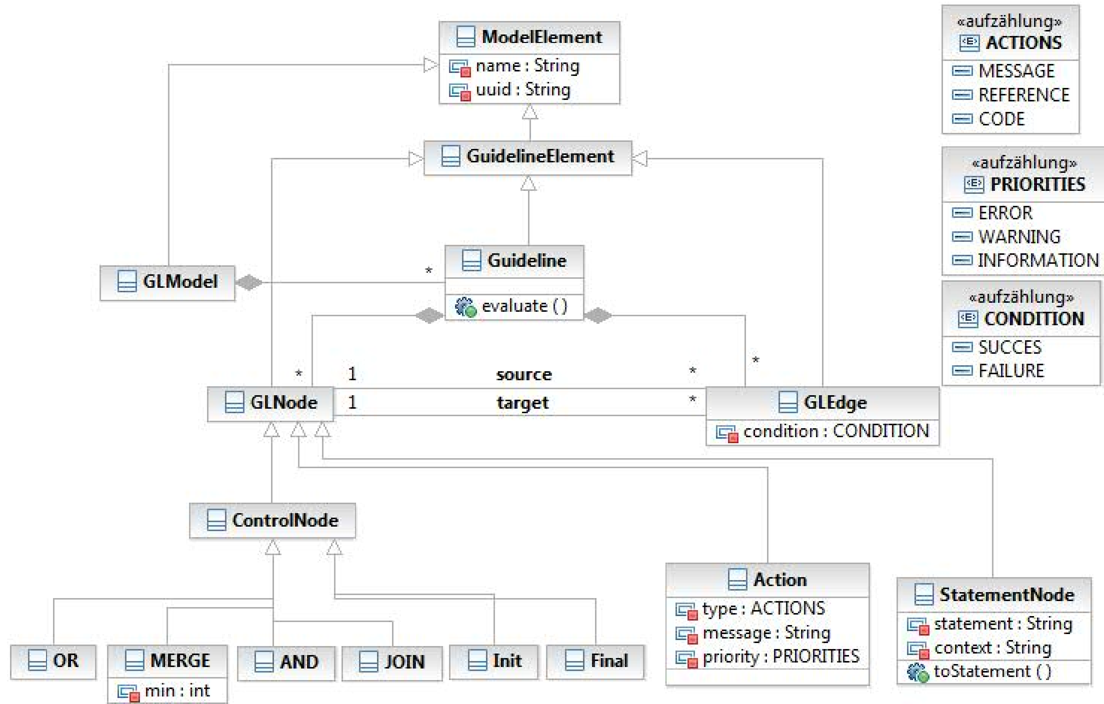


Figure 4.11: Guideline Structure

control nodes (*ControlNode*) for controlling the evaluation order, action nodes (*ActionNode*) for initiating actions, and statement nodes (*StatementNode*) for the specification of statements, which will be detailed in Section 4.6.3.2.

Control Nodes

Principally, a complex guideline is an ordered process of statements and actions. Similar to process models, more complicated control-flows between nodes can be enabled by introducing control nodes to detail the evaluation order. Therefore, we introduce four basic control-nodes, whose semantics is defined as follows:

- *Init*-nodes determine the beginning of a complex guideline. Only one *Init* node is allowed per guideline.
- *Final*-nodes determine the end of a complex guideline. Only one *Final* node is allowed per guideline.
- *OR*-nodes have exactly one ingoing and multiple outgoing edges. It initiates alternative paths in statement evaluation, whereof at least one alternative path, i.e., outgoing edge, must be processed. While complex expressions are possible to decide about which alternative path should be taken, for simplification, we only distinguish two alternatives to cover positive and negative results. Therefore, outgoing edges provide a conditional value, which either can be *SUCCESS* or *FAILURE*. An *OR* node, which either receives a logical *true* or a non-empty set of objects from a preceding node, propagates this message to all outgoing edges, whose condition is

set to *SUCCESS*. On the other side, the incoming logical value *false* or an empty set is propagated to all outgoing edges, whose condition is set to *FAILURE*.

- *MERGE*-nodes have multiple ingoing and one outgoing edge and are the counterpart of *OR* nodes to synchronize paths, which were introduced by an *OR* node. Therefore, it waits for a minimal number, which is indicated by the *min* attribute, of ingoing messages, before it evaluates ingoing messages and sends one message to its successor node. The concrete behavior of a *MERGE*-node depends on the type of ingoing messages. One must take care of the message type of all preceding nodes, which can be logical values or object sets. A *MERGE* node combines ingoing results. Therefore, it must be ensured at design time, that all ingoing messages have the same type, i.e., either logical values or object sets. For ingoing logical values $(b_1, .., b_n)$ all ingoing values are combined by using the logical operator *OR*, i.e., $b_1 \vee .. \vee b_n$. In contrast, if multiple sets $(s_1..s_n)$ arrive in an *MERGE*-node, the sets are aggregated by using the intersection operation, i.e., $s_1 \cap .. \cap s_n$. Afterwards, the aggregated set or the combined logical value is sent to the successor node.

- *AND*-nodes initiate multiple outgoing paths, whereof each must be processed in parallel and independent from an edge's condition. Therefore, it propagates an incoming messages to all outgoing edges.

- *JOIN*-nodes are the counterpart of *AND* nodes and synchronize paths, which were introduced by an *AND* node. Similar to *MERGE*-nodes, their concrete behavior depends ingoing message types: In the case of incoming logical values, a *JOIN*-node synchronizes the results $(b_1, .., b_n)$ by using the logical operator *AND*, i.e., $b_1 \wedge .. \wedge b_n$. On the other side, for object sets, a *JOIN*-node synchronizes the result sets $(s_1..s_n)$ by applying an union set operation, i.e., $s_1 \cup .. \cup s_n$. To trigger the behavior of the Join-node, all ingoing edges must have provided exactly one message. They synchronize parallel paths by waiting for exactly one message on each ingoing edge, before triggering subsequent nodes.

Action Nodes

To face the effect-centric facet, as discussed in Section 4.6.1.3, we introduce means to repair detected model inconsistencies and/or to show meaningful advices. This is realized by *Action* nodes, which are intermediate nodes with exactly one ingoing and one outgoing edge. Based on the ingoing message, which could be an empty message, a logical value, or an object set, different action types can be initiated. Presently, we differentiate three different types of *ActionNode*s, which can be spread across the complex guideline:

- A message action provides developers with textual guidelines based on the specific evaluation of a statement at specific stages of the complex guideline. In the case of an ingoing edge, which provides an object set, the message e.g., can refer to individual faulty model elements contained in the set.

- Similar to message actions, reference actions provide developers with guidelines. In contrast to message nodes, they do not provide customized textual advices, but link to additional documents with more context information or documented best practices.

- Individual objects which result from evaluated *StatementNode*s, can be modified/rectified by providing the *ActionNode* with code fragments or an M2M transformation. As the type of a pattern node's result set can uniquely be defined at design time, a typed transformation or program code, which is provided and executed during the guideline evaluation, enables the automated reparation of faulty model elements, i.e., it realizes **production guidelines**.

To indicate the relevance of a particular action and to realize the deontic and alethic characteristic of **integrity guidelines**, the priority values *ERROR*, *WARNING*, and *INFORMATION* are defined. While *ERROR*s have a high priority and represent mandatory action as well as some kind of "knock-out" criterion for the finalization of the validated task, medium prioritized *WARNING*s can be neglected if possible. *INFORMATION* actions have a low priority and have informative character to log basic or statistical information.

Figure 4.12 depicts a complex guideline which demonstrates the most essential constructs for guideline modeling. The guideline starts with an *Init*-node, which leads to a *StatementNode* to realize a statement, as detailed in the next section. Based on the result of the statement's evaluation, an *OR*-node switches between two alternative paths to proceed. For the *SUCCESS* case (i.e., the statement's result is true or a non-empty set of objects), the guideline defines no further action, meaning that the *OR* path can be finalized by sending a message to the *MERGE*-node, which synchronizes the two *OR* paths. Otherwise, if the *StatementNode* results in the logical value false or an empty set of objects the *FAILURE* path is taken. On that path, the *OR*-node calls the message action node on the left side. After the message was processed, the *MERGE* node is triggered to synchronize the alternative block. Finally, the guideline interpretation is finalized by calling the *END* node.
This, indeed, is a simple example to demonstrate basic elements of our meta model. More complex guidelines, though, can be specified by combining more *StatementNode*s, which are connected by control nodes and adequate set of intermediate or final actions.

4.6.3.2 Abstract Syntax for Statements

Statements are used to evaluate particular information of a model, which serves as an artifact of an activity, and provide other nodes with their result. Basically, such statements can be formulated using state-of-the-art languages, such as OCL or EVL. However, since the definition of statements is mostly a complex and confusing task using the standard textual notation, we introduce a more intuitive graphical notation, which abstracts from the formal characteristic of SOTA languages and enables the consideration of MMV data to design more method-oriented statements.
Therefore, we introduce the meta model for statement modeling, which is part of the general guideline meta model and presented in Figure 4.13. In parallel, we detail the contextual statement modeling, which considers artifact information (cf. Section 4.4) during statement design.

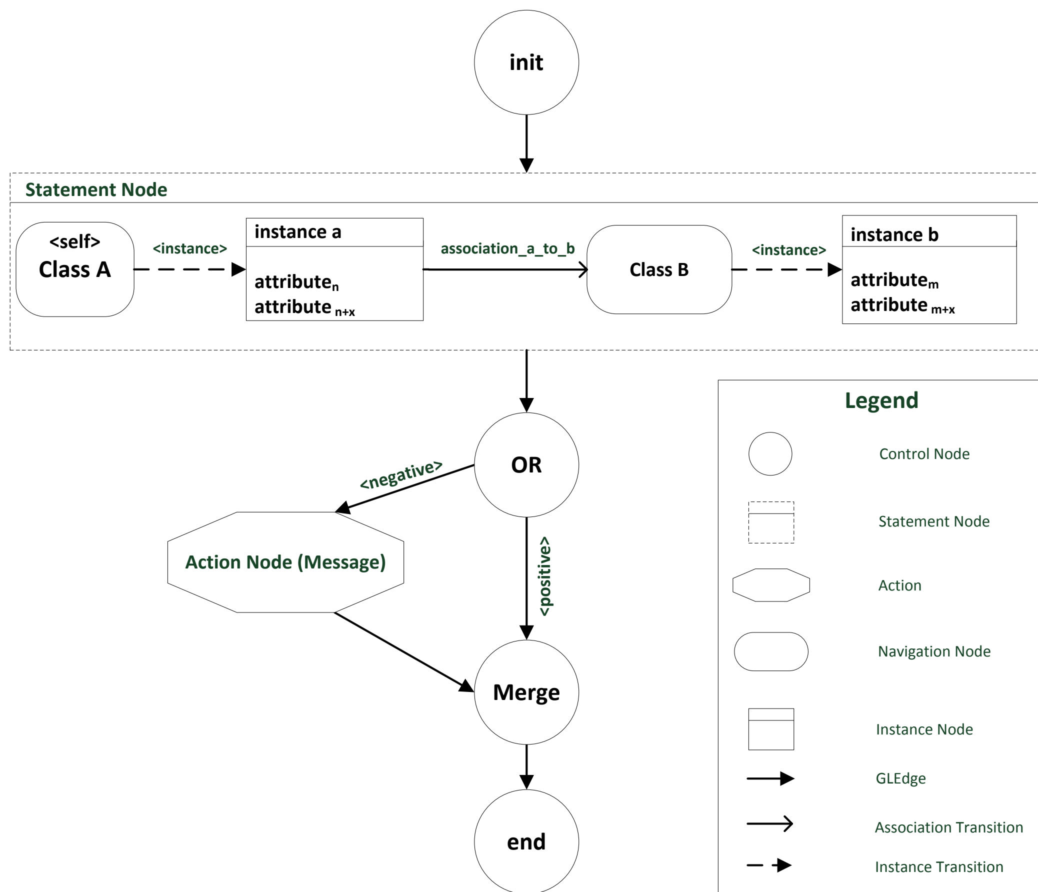


Figure 4.12: Guideline Elements

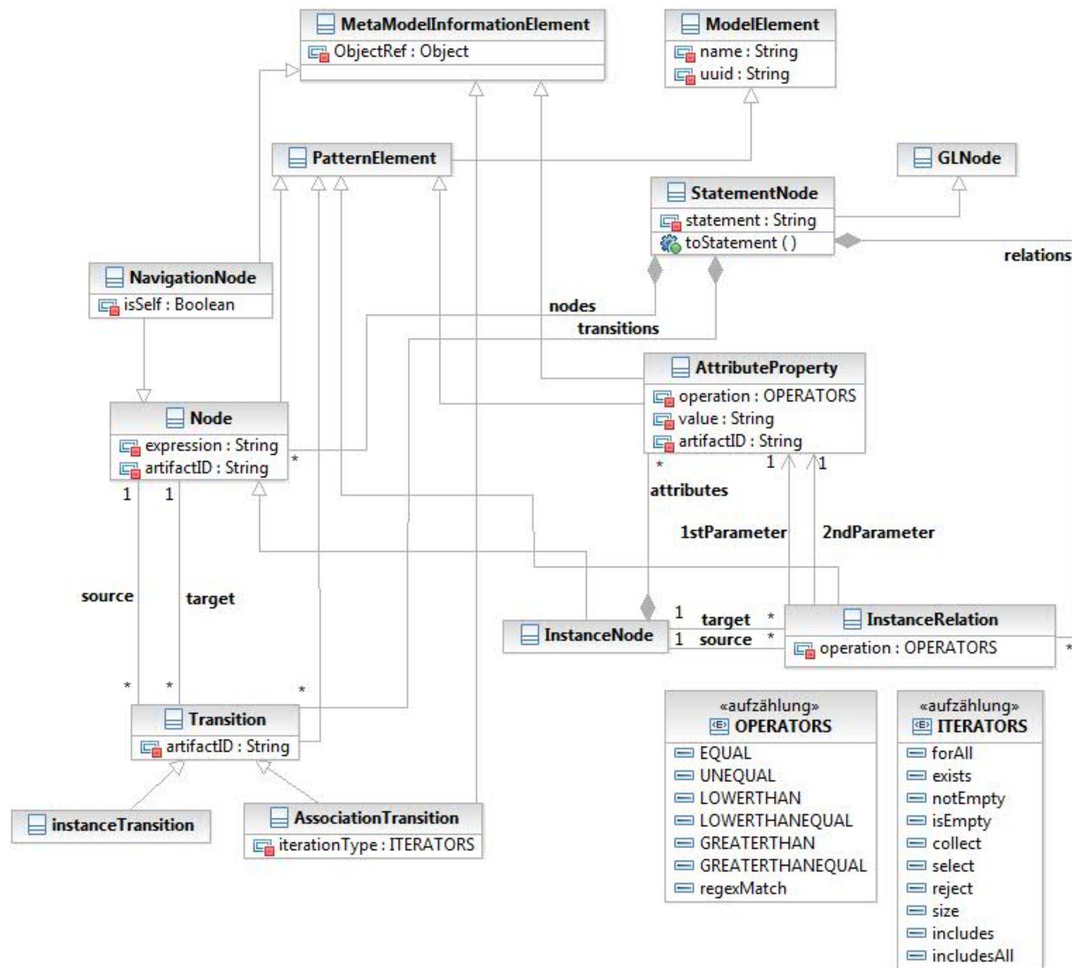


Figure 4.13: Statement Meta Model

Basically, we define a statement as a directed acyclic graph leading from a start model element to other reachable model elements via nodes and edges. (Reachable means, that the navigation path from one element to another follows the abstract syntax, as defined by some meta model) This provides us with a concept to navigate in models, similar to OCLs' dot-Notation. A graph defines multiple paths, which are composed of two types of nodes (*Node*) and two types of edges (*Transition*) to navigate between meta model classes (*NavigationNode*) and modeled instances (*InstanceNode*). *InstanceNode*s enable the evaluation of attribute properties (*AttributeProperty*) of instances and the definition of conditional relationships (*Relation*), which must hold for attributes of different instances. That way, multiple simple statements can successively be combined into one complex statement. In the following, we detail the abstract syntax for the definition of statements and introduce a concrete syntax, which supports the visual design of guidelines.

Statement Node

A *StatementNode* is a particular *GLNode*. It is a composite node for *Node*s and *Transition*s, which both are necessary for statement definition. To simplify the definition and understanding of guidelines, a modeled statement serves as an abstraction of a rule on technical level, which must normally be modeled by using a more complex validation language, such as OCL. To be open to a broad range of platforms or languages, a *StatementNode* provides the *toStatement()* method, which can be realized differently, to derive particular platform-specific validation code from the statement model. In Section 5.5, we will demonstrate such a transformation of a statement model using the example of OCL.

Meta Model Information Element

To face the product-centric facet of a guideline, as discussed in Section 4.6.1.1, i.e., to address relevant entities of an MUD, statement design must be provided with a vocabulary, which is basically provided by the elements of the MUD's meta model. Therefore, to realize the *product-centric facet* of a guideline, the *MetaModelInformationElement* is an abstract class, whose *objectRef* attribute enables a linkage between a statement element and its counterpart element in some meta model. To reduce the general meta model vocabulary to a method-relevant subset, contained statements exclusively must address the MMV elements of an artifact, which is associated with the respective MC, for which the statement must hold. Depending on the concrete realization of the *MetaModelInformationElement*, it references a classifier, an association, or an attribute, as defined by some meta model and referenced by an associated MMV.

Node

A Node is an abstract *MetaModelInformationElement* and base class for the two node types of a statement: a *NavigationNode* to address a class of a meta model and an *InstanceNode* to address instances of a class.

Navigation Node

A *NavigationNode* is a *Node*, which represents a meta model class. It provides the *isSelf* property, which is a logical value to indicate whether the node provides the context of

the statement, i.e., the starting point for evaluation. One statement must define exactly one node, whose *isSelf* attribute is **true**.

Instance Node

An *InstanceNode* is a *Node*, which represents a typed object, by which instances of a meta model class can be addressed. It aggregates a set of *attributes* to enable the validation of particular instance properties. Beside simple expressions, by which an attribute can be compared with a fix value (e.g., *name == "WatchdogManager"* or *delay* <> *5*), attributes also can be source (*1stParameter*) or target (2ndParameter) of a *Relation* to interrelate two *AttributePropertys* of the same or different *InstanceNodes* (e.g., *instanceA.delay* <= *instanceB.delay*). That way, conditions or constraints are defined for individual typed model instances and their attributes. Again, the associated MMVs restrict the vocabulary of referenceable classes and attributes.

Transition

A *Transition* is an abstract base class for directed edges between a source *Node* with a target *Node*. There are two types of *Transitions*: a *NavigationTransition* to navigate between meta model classes and a *instanceTransition* to link *InstanceNodes* with their respective type, i.e., a *NavigationNode*.

Association Transition

An *AssociationTransition* is a *Transition* between either a *NavigationNode* (source) or an *InstanceNode* (source) and a *NavigationNode* (target), to follow an association property of a particular meta model class or one of its instances. Basically, the *AssociationTransition* reproduces a meta model association and enables the meta model conform navigation from one typed statement *Node* to another. Therefore, an *AssociationTransition* likewise provides the *objectRef* property to reference its meta model counterpart and to document the transition type. As meta model elements can only be referenced, if they are defined by an associated MMV of the respective MC, two benefits can be achieved: First, only method-relevant associations can be used for statement modeling, which makes statement definition less confusing. Second, statement nodes, i.e., classes and instances, are only navigable via a transition according to the underlying meta model.

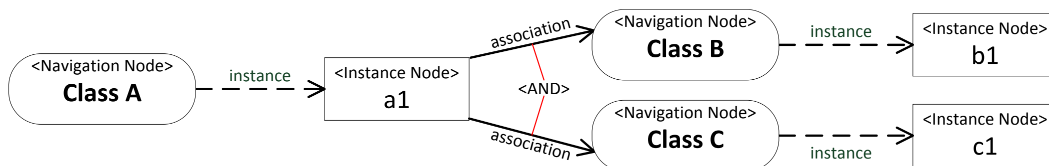


Figure 4.14: Statement Modeling I: association Transition

*NavigationNode*s, unless its *isSelf* property is set to true, must have exactly one ingoing *AssociationTransition*, whose source node defines the meta model type of the referenced node. In contrast, any *Node* may have multiple outgoing *AssociationTransition*s, whereof each transition spans one sub-statement in the context of the source node's type. The resulting set of sub-statements is combined by default using the logical operator *AND*. This is sketched in Figure 4.14, where an *InstanceNode* is starting point for two statements. Both statements are defined in the context of an instance of type A, i.e., a1, for which a meta model defines two associations: the first between class *A* and class *B*, and the second between class *A* and class *C*. That way, the first statement is defined for referenced objects of type *B* in the context of *a1* and the second statement is defined for referenced objects of type C in the context of a1.

Finally, referenced associations with multiplicity 0 or 1 and associations, which reference a collection of objects, must be distinguished. Associations with a maximum multiplicity of 1 at most reference one object and can be processed immediately. On the other side, collection associations must define how to process contained objects. For that reason, an *AssociationTransition* provides the *iterationType* property to indicate how the collection must be processed. According to the collection operations of OCL [OMG06a], the meta model in Figure 4.13 presents a selection of potential *ITERATORS* to define the *iterationType* of an *AssociationTransition*.

Instance Transition

An *instanceTransition* is a *Transition* between a *NavigationNode* (source) and an *InstanceNode* (target) to address and to distinguish named instances of a class, which can be used for further processing. *InstanceNode*s must exactly have one ingoing *instanceTransition*, whose source, the *NavigationNode*, defines the type of instance, via the inherited *objectRef* attribute. In contrast, a *NavigationNode* may have multiple outgoing *instanceTransition*s, whereof each transition spans one sub-statement about a named instance of the same type. The resulting set of sub-statements is combined by default using the logical operator *OR*.
This is sketched in Figure 4.15, where a *NavigationNode* typed with *B* is starting point for two statements. Both statements are defined in the context of an instance of type *A*, i.e., *a1*, for which a meta model defines an association between class *A* and class *B* (cf. *Associ-*

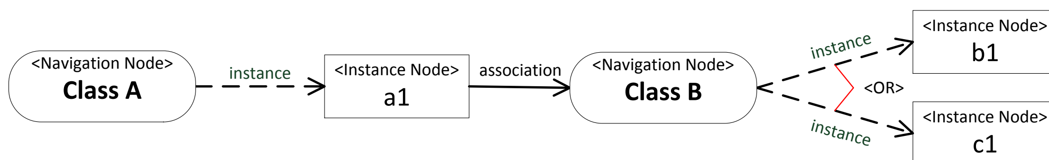


Figure 4.15: Statement Modeling II: instance Transition

ationTransition). That way, the first statement is defined for *b1* in the context of *a1* and the second statement is defined for *b2* in the context of *a1*.

Attribute Property

An *AttributeProperty* is a *MetaModelInformationElement* to address an attribute of an instance, for which a boolean expression can be defined and validated. An expression relates an attribute value of an instance with an expression *value* via a predefined *OPERATOR*. For example, a delay (property) is "lower than"(operation) 5 (value), i.e., $delay < 5$. Multiple expressions of the same instance are connected by the logical operator *AND* by default.

Instance Relation

The *InstanceRelation* relates two *AttributeProperty*s to enable boolean expressions with two variables. It is a directed edge between two *InstanceNode*s, where the source references an *AttributeProperty* of an *InstanceNode* as *1stParameter* and the target references an *AttributeProperty* of an other *InstanceNode* as *2ndParameter*. A selectable binary *operation* relates the two parameters.

The *objectRef* attribute of *MetaModelInformationElement*s enables referencing any meta model element, which is provided by an associated MMV. However, to design statements, whose navigation paths and attribute constraints are in line with a meta model's abstract syntax, the element's application context must be considered, as well. Therefore, the following three restrictions for a *MetaModelInformationElement*'s *objectRef* attribute were defined, to clarify the allowed context in which classifiers, attributes, and associations are referenceable. The guarantee of these restrictions is part of the statement editor's functionality, and saves statement designers from misuse and information explosion.

First, the allowed set of referenceable classes must be restricted. As guidelines exclusively are defined for an MC, this set is defined through concepts, which are provided by the MMVs of an MC's input an output artifacts. This is ensured by the following restriction.

Restriction 1 (*Referenceable statement classes*)

Let g be a guideline model *GLModel*, which is referenced by a guideline MF of an MC (mc). Let $InputPart(mc)$ be the union set of all MMVs, which are associated with input artifacts of mc, and let $OutputPart(mc)$ be the union set of all MMVs, which are associated with output artifacts of mc.
Then for each *Node*, i.e., *InstanceNode* or *NavigationNode*, of g, the set of referenceable meta model classes is restricted to the following set of concepts, which are contained

in an associated MMV:

$$\{c \in CONCEPTS | c \in ViewConcepts(v), v \in InputPart(mc) \cup OutputPart(mc)\} \tag{4.58}$$

Second, an *InstanceNode* must not state expressions about attributes, which are not defined by a referenced classes or one of its super classes. This is ensured by restriction 2.

Restriction 2 (*Referenceable attribute properties*)

Let g be a guideline model *GLModel*, which is referenced by a guideline MF of an MC (mc). Let I_c be a class, which is set as *objectRef* of an *InstanceNode*, for which an *AttributeProperty* is defined.
Then the set of the set of referenceable meta model attributes is restricted to the following set of attributes, which are contained in an associated MMV:

$$\begin{aligned}&\{a \in ATTRIBUTES | a \in viewAttributesOfConcept(I_c, v),\\&v \in InputPart(mc) \cup OutputPart(mc)\}\end{aligned} \tag{4.59}$$

Third, a *NavigationTransition* must not connect two *Node*s, which reference classes, for which no association was defined in the underlying meta model. This is ensured by proposition 3.

Restriction 3 (*Referenceable association properties*)

Let g be a guideline model *GLModel*, which is referenced by a guideline MF of an MC (mc). Let s_1 be a *Node*, whose *objectRef* attribute references class S, and let t_1 be *Node*, whose *objectRef* attribute references class T. Then the set of referenceable MMV associations, which are applicable to the *objectRef* attribute of a *NavigationTransition*, which connects s_1 (source) with t_1 (target), is restricted to the following set:

$$\begin{aligned}&\{r \in ASSOCIATIONS | \exists m \in CARDINALITY, \exists b \in BOOLEAN :\\&(r, m, b, y) \in AssocProp, r \in viewAssociationsOfConcept(x, v),\\&v \in InputPart(mc) \cup OutputPart(mc), x \in superConceptsOf(s_1),\\&y \in superConceptsOf(t_1)\}\end{aligned} \tag{4.60}$$

4.6.3.3 Modeling Statements

The last section has introduced the abstract syntax for statement design. We now define the concrete graphical syntax. Thereby, our approach implicitly makes use of MMV

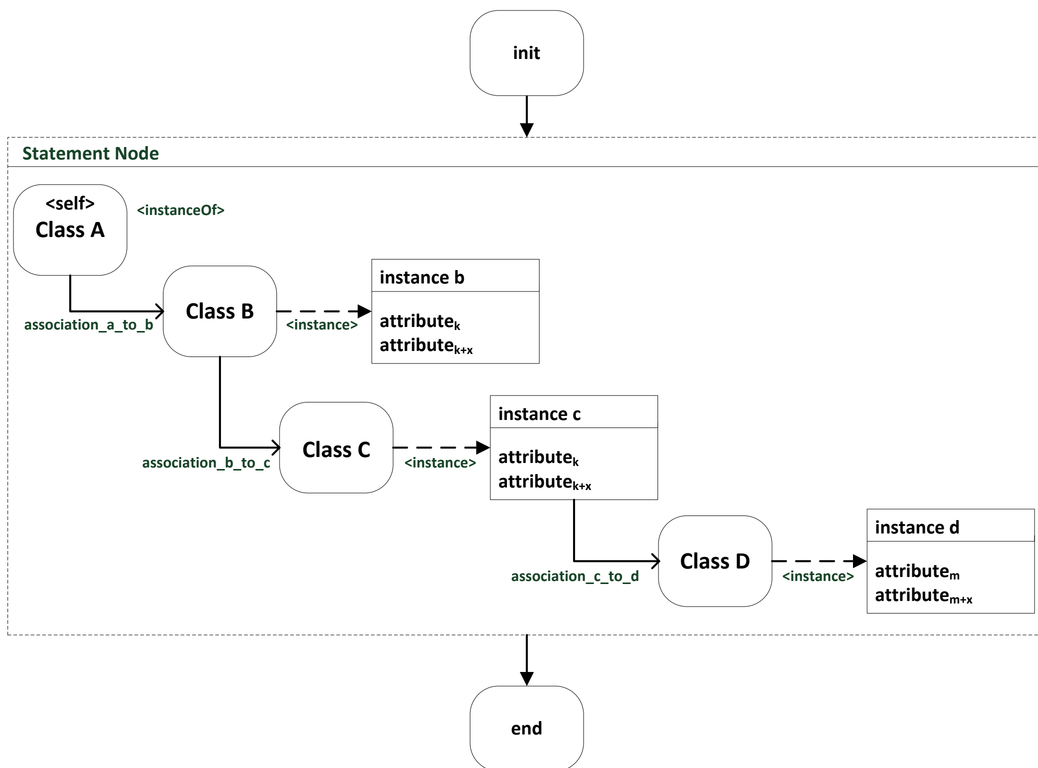


Figure 4.16: Guideline Navigation

information, as specified for the artifacts of an MC. The MMVs are used to restrict the guideline vocabulary, as follows: A statement is defined for a specific meta model. The meta model's vocabulary, though, consists of all features, i.e., attributes and associations, and classes. However, as guidelines are defined for a specific MC, which normally refers to individual meta model parts, only a subset of that vocabulary is required. This meta model subset can be derived from the associated MMVs of an MC's artifacts, to restrict a guidelines vocabulary.
In the following, we inductively define the construction of statements starting from simple basic forms to identify instances right up to complex statements to validate particular characteristics of a model.

Instance Identification

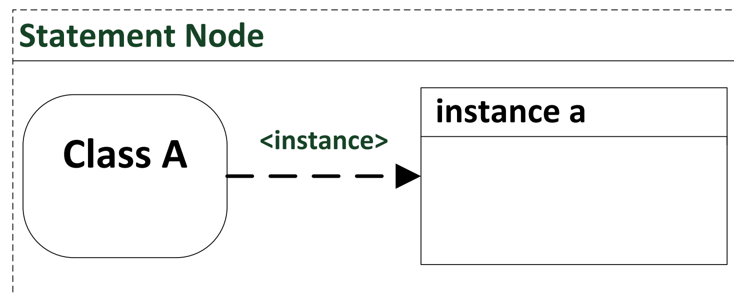


Figure 4.17: Instance Identification

The access to instances of a model class, looks as follows: First, a *NavigationNode*, as indicated as rounded rectangle in Figure 4.17, is required to identify the meta model class and to define the type of the instance of interest. The instance of a model class itself is made addressable by the means of a separate *InstanceNode*, as realized as cornered rectangle in Figure 4.17. This separation is required to distinguish different instances of the same model class, as needed, for example, for comparing various instances of the same set of instances (cf. x1.name != X2.name)). To indicate, that a *NavigationNode* determines the type of an instance node, the *instance* transition relates a class with an instance, as illustrated as directed dashed line. Thus, we can address multiple instances of a model class and use that instances for further processing, as demonstrated in the following.

Attribute Property

Figure 4.18 demonstrates the identification of an attribute of an instance for evaluation purposes. An *AttributeProperty* is realized as an additional entry in the context of an *InstanceNode*. Thereby, valid attributes are predefined by the type of the instance and respective MMV restrictions. For defined *InstanceNode* attributes, additional fields define

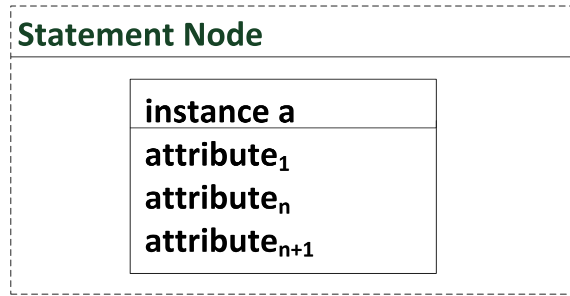


Figure 4.18: Attribute Property

the operation and value (e.g., $delay < 5$), as defined in the statement meta model (cf. Figure 4.13).

Association Expressions I

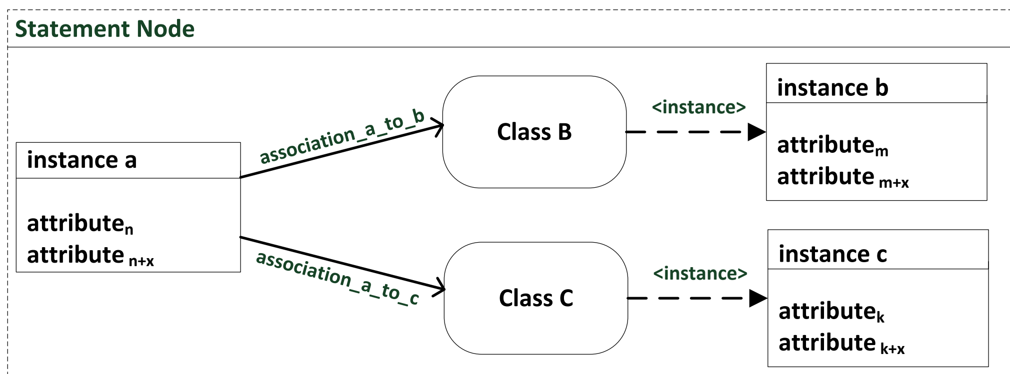


Figure 4.19: Statement navigation via associations

After the identification of instances and attributes is enabled, we discuss the navigation between instance objects of a model, as illustrated in Figure 4.19. To navigate from one instance to another one, the *AssociationTransition* is required first, as illustrated in the figure by a non-dashed line connecting an *InstanceNode* with a *NavigationNode*. The *AssociationTransition* is required to identify the associated model class, from which particular instances are addressed secondly using the *instance* transition, as discussed above. Due to the *objectRef* attribute of the abstract meta class *MetaModelInformationElement*, as defined in the statement meta model (cf. Figure 4.13), we are provided with type information of respective nodes, which enables to associate the two nodes according to the meta model's rules, i.e., setting the *AssociationTransition*'s *objectRef* attribute accordingly. As shown in the figure, the *AssociationTransition* ends in a *NavigationNode* and does not relate two in-

stances directly. This is required to address various instances of the associated type, as briefly discussed earlier and detailed in the following.

Association Expressions II

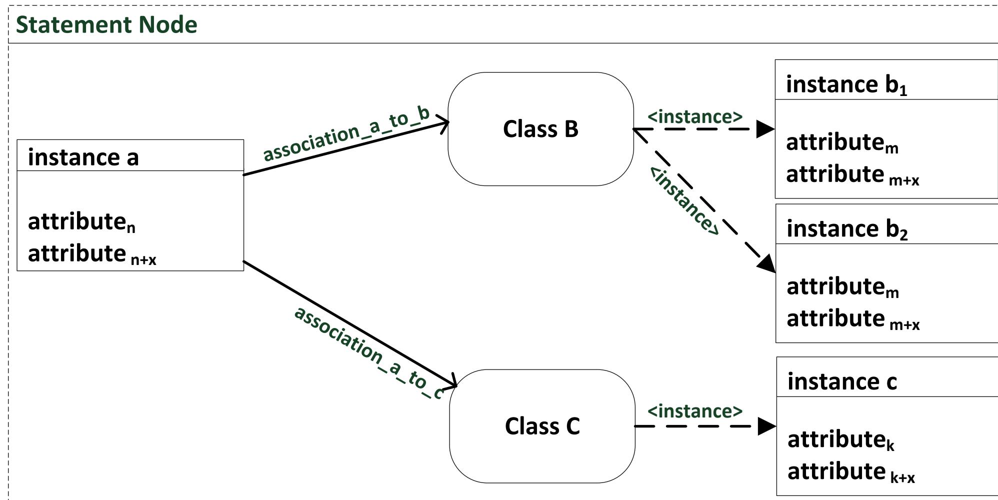


Figure 4.20: Multiple instance variables and iterators

Basically, Figure 4.20 looks very similar to Figure 4.19. The difference is, that the *NavigationNode* for $ClassB$ associates multiple instances. The figure demonstrates the design of iterator expressions in order to iterate a set of instances. For meta model associations with a cardinality of more than one, the association results in a set of referenced instances. To iterate over all instances, an *AssociationTransition* (the non-dashed line), defines the *iterationType* property, as defined in the statement meta model illustrated in Figure 4.13. The property defines a way for iterating a set of instances, similar to languages, such as OCL, EVL, or, in general, predicate logic. Since iterator expressions allow for pair-wise comparison of contained elements, a navigation node, such as $ClassB$ allows the association of several instance nodes to address a required set of iterators.

Crossing Attribute Relations

Finally, in addition to boolean expressions to evaluate attributes of one instance, also attributes of different instances can be related into one expression, as depicted in Figure 4.21. To relate the values of two different attribute of different instances in a boolean expression, the *InstanceRelation* is used. The relation is indicated as non-dashed blue line between $instance\ b$ and $instance\ c$ and references the two attribute properties of interest. The two related attribute values are compared via a logical operation (e.g., $==, != , <, >$, etc.).

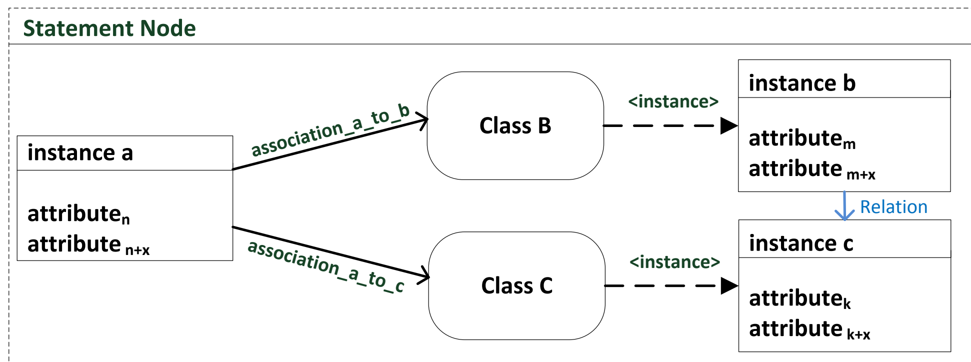


Figure 4.21: Instance attribute relations

4.6.3.4 Modeling Complex Guidelines

The presented meta model for statements misses some of the expressiveness, which is known from other languages, such as OCL [OMG06a], EVL [KRPP10], or Visual OCL [KTW02]. Due to the graphical syntax, which was developed to facilitate the design and readability of statements, the full set of features cannot be integrated without loosing the advantage of an easy design. From our point of view, a concrete graphical syntax, which provides all features of OCL's abstract syntax, would be even more confusing, than its textual counterpart. Especially, the multitude of relationships and interleaved nodes influences readability, usability, and evolution negatively.

Instead, most restrictions can be qualified by arranging multiple statements into a guideline. As multiple statements with different context elements can be combined into more complex guidelines, the power of guideline modeling compensates most restrictions, while introducing additional features. Furthermore, as a modeled statement is transformed into an textual counterpart automatically, the generation of constraint skeletons enables the manual adaption of constraints in a respective language, which is much easier for most of the cases, than development from scratch. This is detailed in Section 5.5.

4.7 Role-centric Workflow Management

Although, role or user management is out of scope for this thesis, this section sketches some basic ideas on how to consider user and role management within process models. Basically, as developers differ in their skills and rights across the project and organization structure, process models should integrate this information with process models, i.e., resource related MFs, in order to allocate tasks and tooling functionality more adequately.

4.7.1 User Skills

Role-specific MFs normally are identified by their name and provide information about necessary skills using natural language text. This complicates the automated matching of roles with human capabilities, which are needed to perform a particular task.
As many developers are subsumed by abstract role descriptions, this approach neglects capabilities and skills of individual developers. The situation can be enhanced by introducing a skill database for storing particular data on employee's expertises. Required skills are linked with role-specific MFs and respective developers. Thereby, roles can still be identified by abstract textual descriptions, but also by matching required with provided skills.
A workflow management system can use this information to allocate tasks to available users with sufficient skills according to the task's priority. Thereby, as appropriate developers can be found via skills instead of role descriptions, unnecessary delays of time-critical tasks can be avoided.

4.7.2 Data Access

Not only user skills, but also rights might be relevant for process execution and contained methods. To protect intellectual property or sensitive data, fine-granular or task-specific access privileges complement exclusively annotated user skill information. User- and activity-specific privileges could be applied to restrict unauthorized developers to view or modify individual data.
User management capabilities, cf. [ZCO07, DPS03, SCFY96, SS94], can be an useful add-on, which can be integrated into process models. However, this would go beyond the scope of this thesis.

4.8 Case Study

To demonstrate the application of Computational Method Engineering (CME), as discussed above, a case study is presented, which is inspired by the automotive sector. Therefore, we developed a meta model, which provides a required subset of data types to support our automotive software development process. The meta model is inspired by the MAENAD project and serves us as a simplified version of the herein developed EAST-ADL meta model. As illustrated in Figure 4.22, it provides a *System* element, which aggregates various models, which belong to the different phases (or abstraction levels) of system development, i.e., analysis, design and implementation. On the analysis level, Requirements Engineering activities are conducted. Therefore, the meta model provides elements to create functional (Functional_RQ) and non-functional (NonFunctional_RQ) requirements. On the design level, a functional design is created using functions and connectors. Additionally, the meta model provides elements to define function-specific timing constraints. Finally, on the implementation level, the functional design of the design level is refined into an implementation-specific software architecture, which we

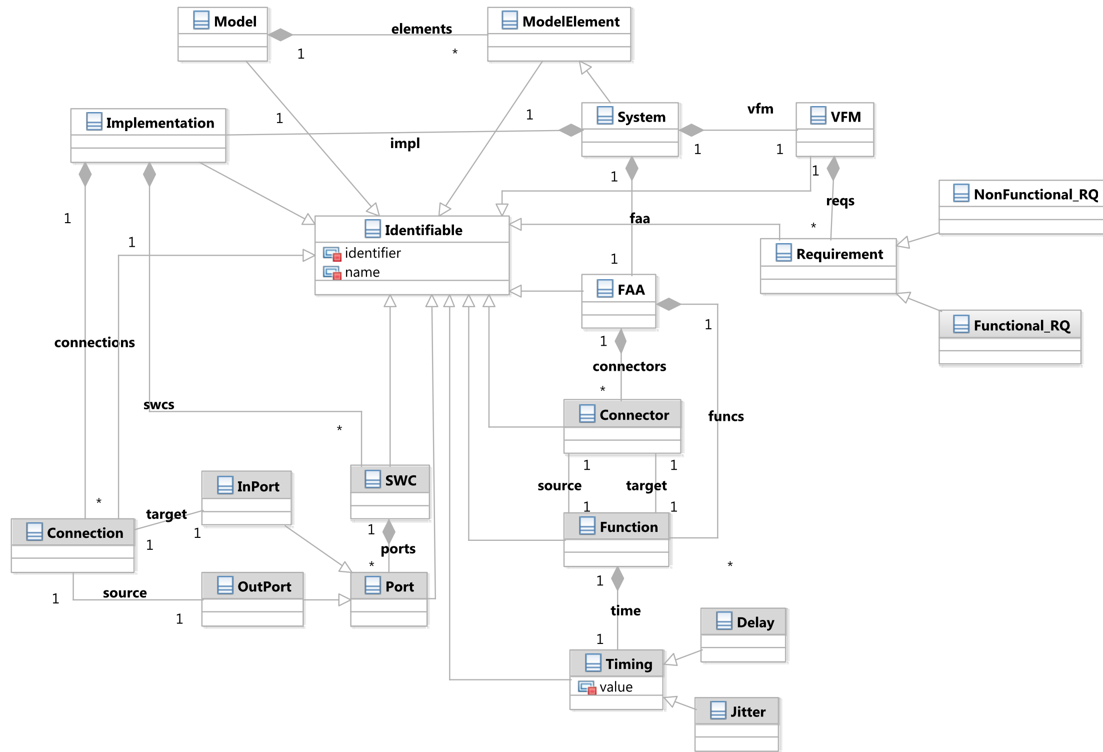


Figure 4.22: Case Study: Meta model

require to be composed of software_components, ports, and connectors.

In our development process, which is illustrated in Figure 4.23, individual parts of the meta model are applied to different development activities. The development process consists of three phases (Analysis, Design, and Implementation), composed of eight MCs to support the following tasks: On the analysis level, it starts with *CreateFunctionalRequirements* to derive functional requirements from external stakeholder requirements, followed by *CreateNonfunctionalRequirements* to specify non-functional requirements for each of the identified functional requirements. The analysis phase ends with the *CompleteRequirements* activity, to validate the complete set of functional and non-functional requirements. Subsequently, the design phase starts using the combined set of requirements in order to derive a functional design from the results of the analysis phase. Therefore, during the MC *DefineFunctions* functions are derived from identified requirements. To specify the interaction between functions, *ConnectFunctions* produces a *Functional_Architecture*. The *Functional_Architecture*'s functions are refined by timing information, during the subsequent MC *DefineFunctionTiming*. During the final development phase (Implementation), the functional architecture has to be refined into a software architecture by using the results of previous phases. Therefore, the artifact *FA_refined_by_timing* is input to *DefineSWCs*, which derives software components from functions of the functional design. The final step, *DefineSWArchitecture*, has to connect software components using port and connector elements of our meta model.

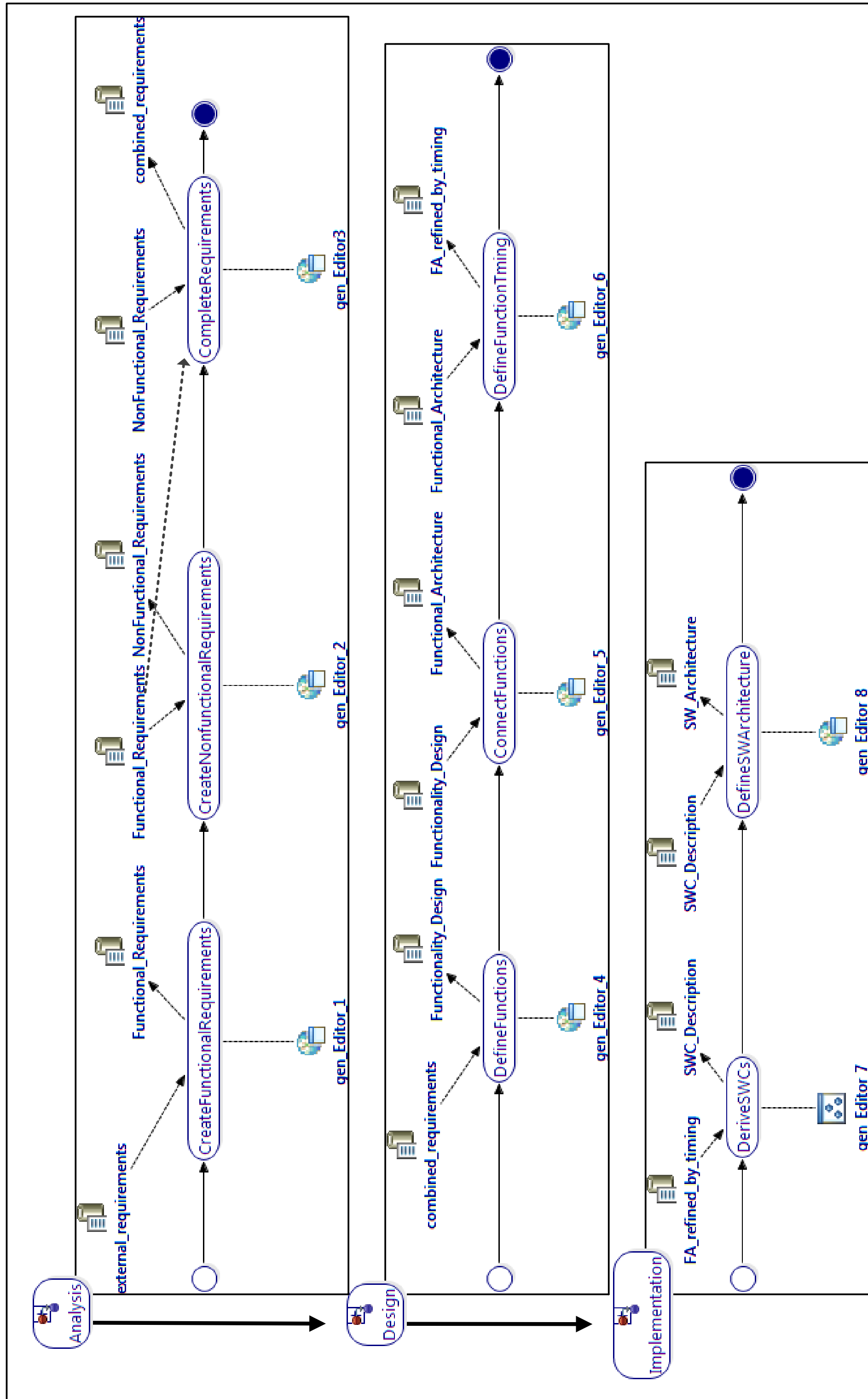


Figure 4.23: CME Case Study Process

MCs are realized using the SPC semantics, i.e., the sequence of MCs within one phase is executed in a strict order, as defined by the process model. In contrast, phases are realized using the FPCs semantics in order to enable the flexible control of individual development phases using validation results and introduced traceability strategies. In the following, we demonstrate the technical refinement of the constituent parts of our development process, i.e., the MCs and their MFs. We will discuss, the definition of MMVs to detail the semantics of artifacts, the annotation of EUSAs to enable the editor generation, the definition of From-Artifact-To-Artifact relations, and the definition of method-specific guidelines. Thereby, we aim at the design of an executable process, which subsequently provides developers with situational guidance, validation and customized editors.

4.8.1 Artifact Design

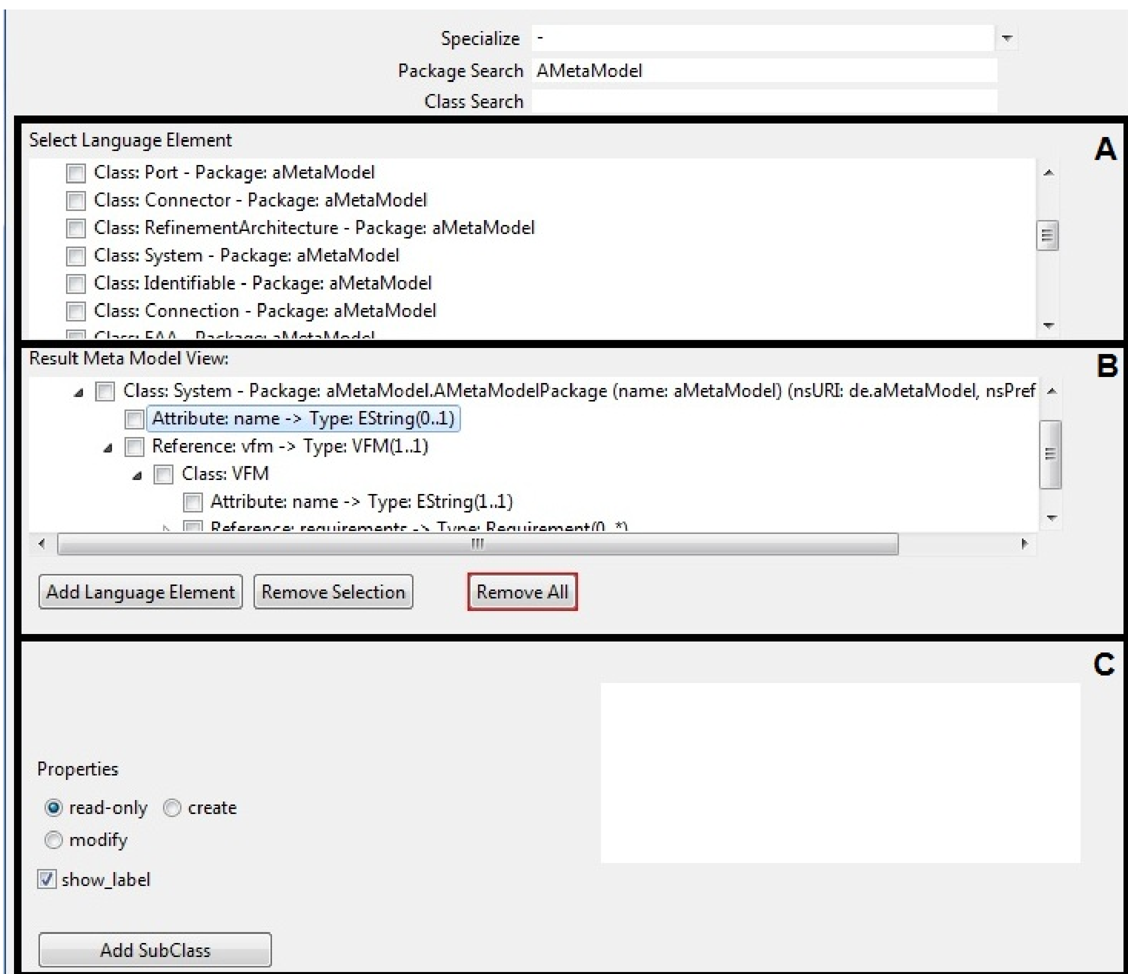


Figure 4.24: Meta Model View Designer

We start with an excerpt of annotated MMVs, which explicitly specify the data structure of an individual artifact based on the meta model illustrated in Figure 4.22. Due

to the large number of artifacts, we focus on a subset of artifacts, while for the real case-study all artifacts were annotated with a respective MMV. For the definition of an artifact-specific meta model, we developed an editor, which is depicted in Figure 4.24. This editor, which is based on the eclipse platform, is connected with the eclipse-specific meta model registry to access any relevant meta model and to select relevant language elements to add them to a MMV. While section *A* of the editor, provides the set of available language elements as originally defined in some meta model, section *B* shows the restricted set of elements, i.e., the MMV implicitly organized in the form of an MMVT, as described in Section 4.5. Additionally, the section *C* of the editor illustrates the possibility to define the EUSA for a specific element. The figures, which we are discussing in the following, represent the result MMVs, which were created using our developed editor.

Figure 4.25 depicts the MMV as specified for the output artifact of the MC *CreateFunctionalRequirements*. The artifact *Functional_Requirements* only focuses on functional requirements associated with the *VFM* container element of the overall *System*. Other elements of the base meta model are masked out. It can be seen, that inherited attributes are incorporated with the MMV elements, likewise, to integrate relevant characteristics, such as an element's name name or an required association, with the view. However, while the *identifier* attribute of elements typed with *Functional_RQ* is relevant for the view, the *VFM* element does mask this attribute out, since this type of information is not relevant for the respective artifact.

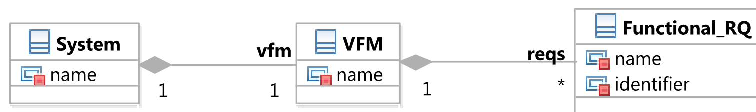


Figure 4.25: MMV of the Artifact Functional_Requirements

The second MMV is illustrated in Figure 4.26 and shows the output MMV associated with the artifact *combined_requirements*. In contrast to the MMV discussed before, it additionally considers both types of requirements defined in the base meta model. However, this task aims at the validation of requirements. Therefore, the MMV particularly neglects the identifier attribute of requirements, since the information is not relevant in this context.

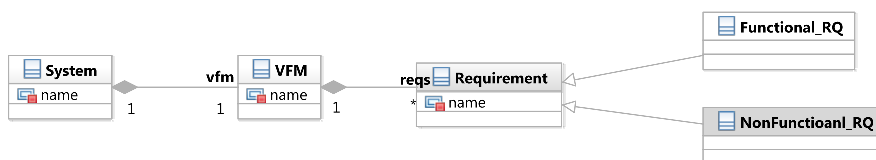


Figure 4.26: MMV of the Artifact Combined_Requirements

In Figure 4.27, the output MMV of the task *DefineFunctionTime* is depicted. It masks out any language element, except the *Functions* contained in a *System*'s *FAA*. Additionally, it considers the *Timing* information of a *Function*, such as its *Delay* or *Jitter*. Some attributes and associations are masked out, while others are used, as predefined in the meta model. For example, while each element requires a name for identification, the only element, for which an identifier attribute is relevant during the task, is the *Timing* element.
Figure 4.28 depicts another MMV specified for the output of MC *DefineSWArchitecture*.

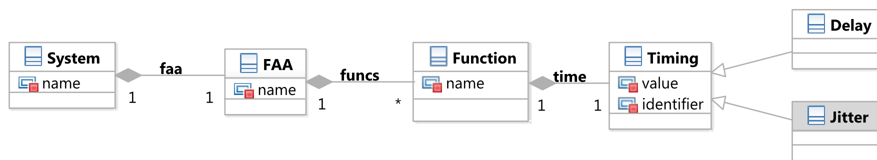


Figure 4.27: MMV of the Artifact FA_refined_by_timing

For this artifact the software components (*SWC*) and connections between associated ports are relevant. Likewise, the MMV neglects irrelevant attributes and associations, which are not required for defining a software architecture.

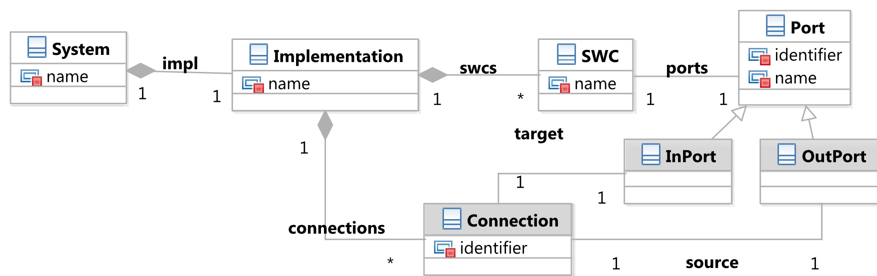


Figure 4.28: MMV of the Artifact SW_Architecture

After defining the MMVs for all artifacts, the artifact content propagation, as discussed in Section 4.4.3, is considered. For example, the output artifact of the MC *CompleteRequirements* (combined_requirements) aggregates a complete set of artifacts, i.e., all requirements, which were defined in the context of previously defined artifacts, are propagated to the artifact. Therefore, we define a *FromTo* relationship between a *NonFunctional_RQ* concept of the output MMV element of *CompleteRequirements* and the input artifact *NonFunctional_Requirements*. Moreover, a *FromTo* relationship is defined between a *Functional_RQ* concept of the output artifact and the *Functional_Requirements* input artifact. That means, at process execution time the framework is enabled to associate defined elements of the input artifacts with the output artifact automatically.

4.8.2 Editor Design

Based on the specification of relevant artifact data structures, the required editor behavior is defined, i.e., for each element defined within the MMV of an artifact, the EUSA annotation is defined to specify, if the element can be created, is modifiable, or if it is read-only. This is realized in parallel with the MMV definition. As illustrated in section *C* of Figure 4.24, our prototypical MMV designer enables the EUSA annotation in parallel with the selection of elements of the MMV.

For the above defined MMVs, Table 4.2 to Table 4.5 exemplify the EUSA annotation, as it was realized for each MMV, i.e., artifact, in our case-study process. For example, Table 4.2 shows the EUSA annotation of the artifact *Functional_Requirements*. While the first column references a class of the MMV, the second column addresses one or more features of the respective class. The type of the feature, i.e., attribute or association, is represented in the third column. In the subsequent column, for each element the respective EUSA annotation is given. The last column indicates whether or not the feature is used as label provider.

4.8.3 Guideline Design

After defining all MMVs and EUSAs, guidelines can be defined. In the following, we describe the design of individual guidelines, as specified for the above process (cf. Figure 4.23).

Figure 4.29 illustrated a guideline, which was specified for the MC *CreateFunctionalRequirements*. This guideline consists of two checks to ensure, that each functional requirement, which is created in the context of that task, has a name, which starts with the simple indicator characters *"FRQ_"* and ends with a random number of characters and digits.

Class Name	Feature Name	Feature Type	EUSA	use as label
System			read-only	
	name	attribute	read-only	x
	vfm	association	read-only	
VFM			read-only	
	name	attribute	read-only	x
	reqs	association	read-only	
Functional_RQ			create	
	name	attribute	modify	x
	identifier	attribute	read-only	

Table 4.2: EUSA Annotation of the artifact Functional_Requirements

Class Name	Feature Name	Feature Type	EUSA	use as label
System			read-only	
	name	attribute	read-only	x
	vfm	association	read-only	
VFM			read-only	
	name	attribute	read-only	x
	reqs	association	read-only	
Functional_RQ			create	
	name	attribute	modify	x
NonFunctional_RQ			create	
	name	attribute	modify	x

Table 4.3: EUSA Annotation of the artifact combined_requirements

Class Name	Feature Name	Feature Type	EUSA	use as label
System			read-only	
	name	attribute	read-only	x
	faa	association	read-only	
FAA			read-only	
	name	attribute	read-only	x
	funcs	association	read-only	
Function			create	
	name	attribute	modify	x
	time	association	read-only	
Jitter			create	
	id	attribute	read-only	x
	value	attribute	modify	
Delay				
	id	attribute	read-only	x
	value	attribute	modify	

Table 4.4: EUSA Annotation of the artifact FA_refined_by_timing

Class Name	Feature Name	Feature Type	EUSA	use as label
System			read-only	
	name	attribute	read-only	x
	impl	association	read-only	
Implementation			read-only	
	name	attribute	modify	x
	swcs	association	read-only	
	connections			
SWC			read-only	
	name	attribute	read-only	x
	ports	association	read-only	
InPort			create	
	identifier	attribute	read-only	x
	name	attribute	modify	
OutPort				
	identifier	attribute	read-only	x
	value	attribute	modify	
Connection			create	
	identifier	attribute	read-only	x
	source	association	modify	
	target	association	modify	

Table 4.5: EUSA Annotation of the artifact SWArchitecture

The regular expression, which realizes this naming conventions is $(FRQ_)[a-zA-Z0-9]+$. The guideline starts at the *init* node. Following the outgoing edge, a statement node is called to check first, whether or not each requirement has a name, at all. The statement is modeled using the techniques introduced in Section 4.6.3.3 and addresses the instances of a functional requirements (*FunctionalRQ*) to define, that their name attribute must not be null. Based on the result of this check a conditional branch either provides developers with the message, that names must not be *null* (FAILURE) or the next statement node is executed (SUCCESS). The second statement addresses functional requirements to define the above regular expression $(^{(}FRQ_)[a-zA-Z0-9]+)$ for their name attribute. Subsequently, the result of this check is evaluated and either results in a warning message, if the naming convention is not hold, or the guideline is finalized.
This simple example demonstrates the basic control flow of a guideline using the alternative control nodes *OR*/*ORMerge* and statement nodes. In Figure 4.30 another statement node is illustrated to demonstrate a guideline, which we specified to guide the *Define-Functions* MC. The statement expresses, that the number of functional requirements, which are defined on VFM level must be equal to the number of functions, which are defined on the level of the Function Analysis Architecture (FAA). Note, that this example statement demonstrates a guideline, which is specific to a particular method, so that it would not make sense to integrate it as general meta model constraint via, e.g., OCL. Due to simplicity, we mask out the control flow of the guideline and focus the interesting statement node. The statement addresses a *System* instance and the associated VFM and

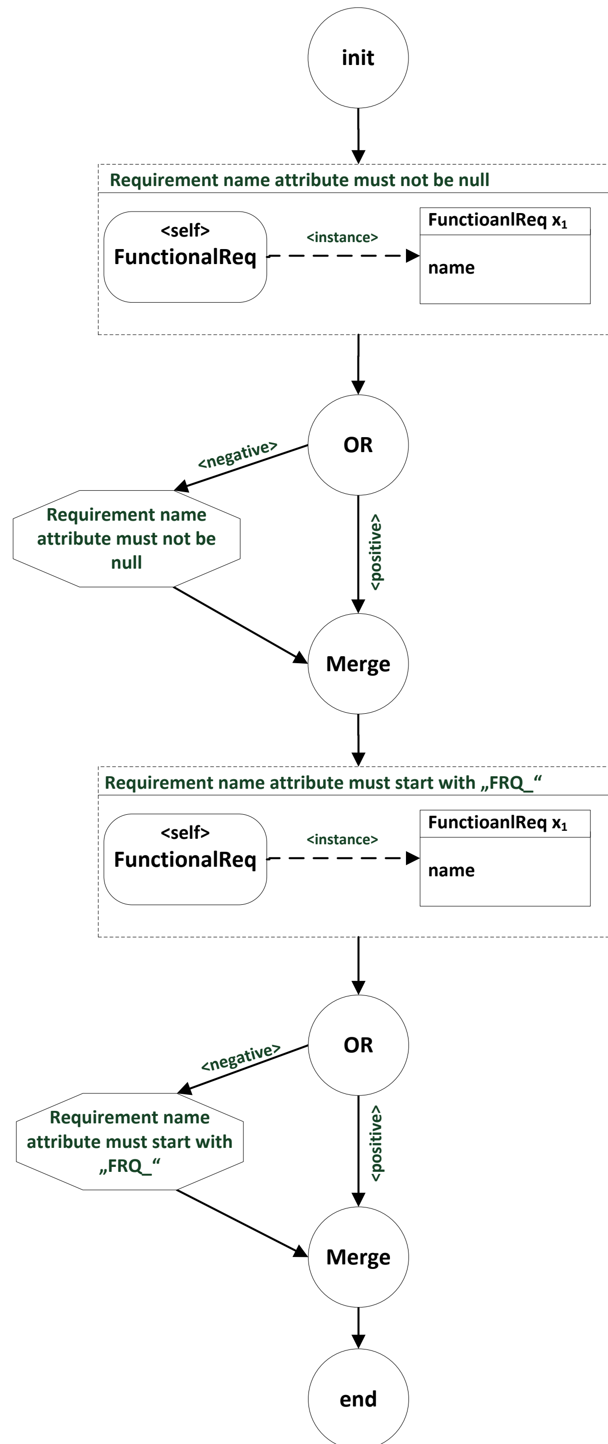


Figure 4.29: Case Study Guideline: Functional requirements must have a name, which corresponds to a particular naming convention

FAA container elements, to identify the two distinct sets of contained functional requirements of the VFM and the *Function* elements, which are contained in the FAA. The size of these two sets is compared using the *InstanceRelation* language element of the statement meta model. This is indicated in Figure 4.30 by the blue directed edge (Relation) between the two instance nodes.

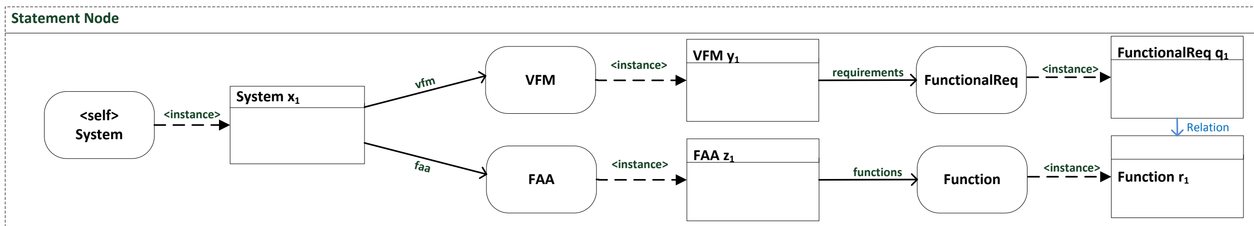


Figure 4.30: Case Study Pattern: The number functions must be equal to the number of functional requirements

A last example of a guideline, which we defined for the case-study process is depicted in Figure 4.31. The guideline validates the range of a particular value attribute. To demonstrate it, we use the example of the *Delay* class associated with each *Function*. Therefore, the statement of Figure 4.31 addresses instances typed with *Function* and navigates to the associated *Delay* instance. For each identified *Delay* the statement checks the value attribute to be lower than 100, i.e., the delay of the function must be lower than 100 milliseconds. While a successful check finalizes the guideline, the failure case results in a warning message provided to a responsible developer to correct the value immediately.

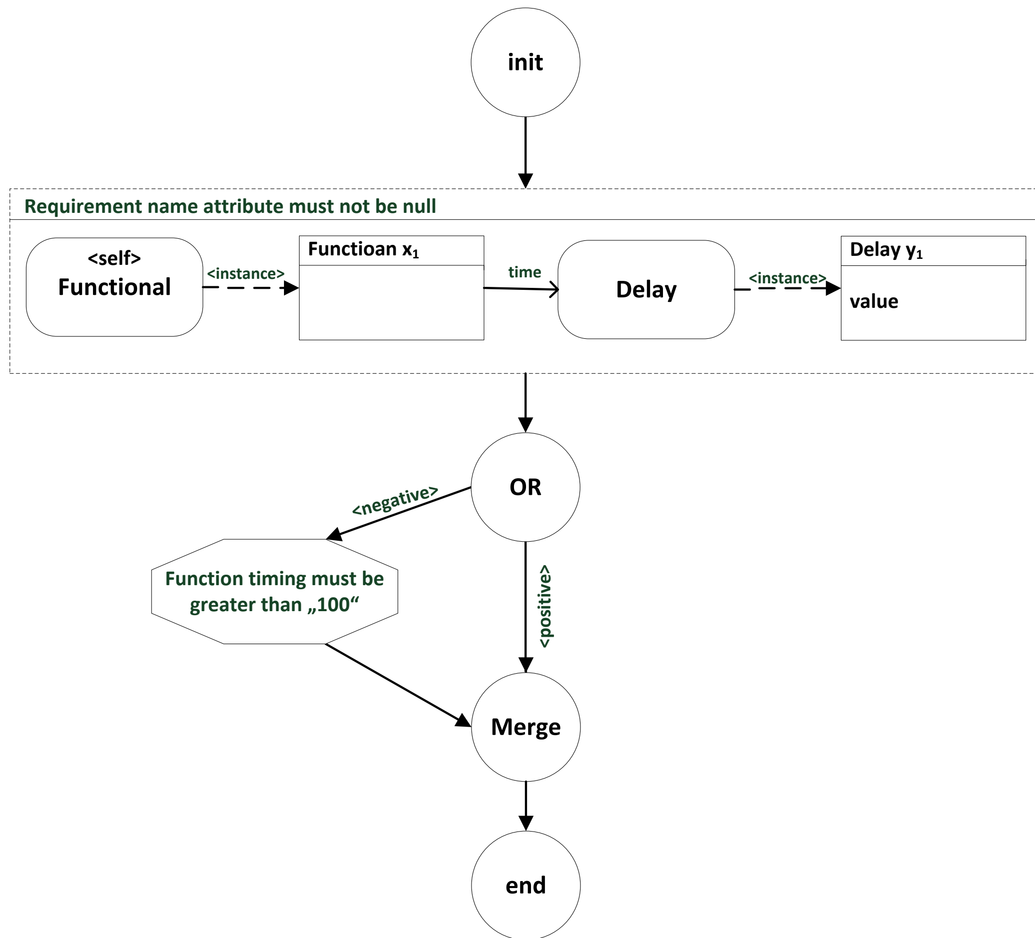


Figure 4.31: Case Study Guideline: a function's timing value must not be lower than 100

5 Method-driven Guidance of Development Processes

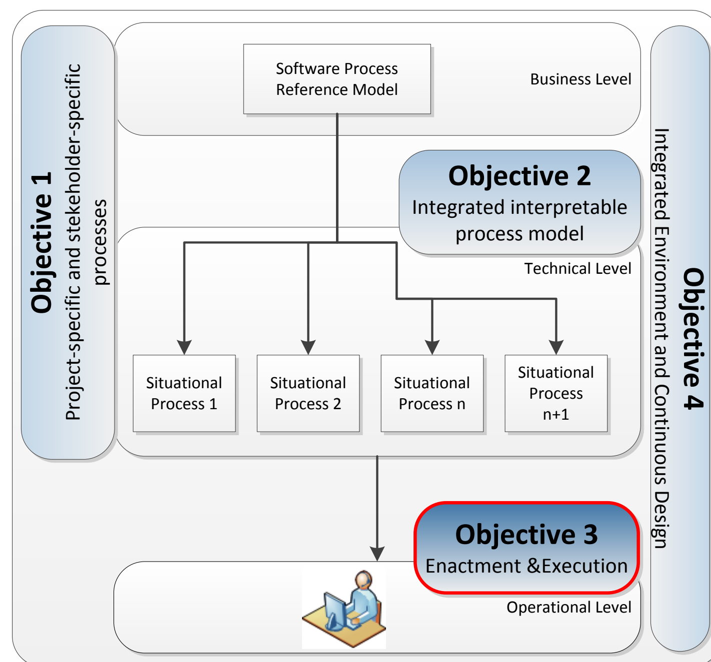


Figure 5.1: Objectives overview

The previous chapters have described the possibilities for complementing software development process models with computer-interpretable information and managing that information in a process line. To face *Objective 3*, as illustrated in Figure 5.1, this chapter describes how the specified development process can be refined from technical design level to operational level.

5.1 Motivation

As today's software development process models are mainly used for process documentation purposes, there is a gap between prescriptive process modeling and subsequent descriptive process evaluation. Between the process design and a final project review,

the multitude of documents and large-scale processes complicate developers's life, since they are lost in the middle of a myriad of information. As a result, it is nearly impossible for individual developers to always consider all rules, standards, and guidelines, which are specific to a software project. However, since the design of processes, templates, checklists and other documents is a time-consuming task, defined process models and associated information must affect developers work actively. Contrasting a passive database of information, where developers must search for relevant information, processes actively have to support developers during their work. Therefore, processes must provide developers with a vehicle, which enables them to be creative in a coordinated and assisted way. A framework, which realizes the required support for development processes, must be aware of the following challenges.

5.1.1 Task-centric Challenges

MCs and composing MFs are information sources, which are defined to support developers in doing a specific job. The challenge is to interpret the information automatically and to provide interested parties with relevant information, which is required to accomplish a job successfully. Therefore, the computer-interpretable formalization of MF information, as introduced in chapter 4, plays a major role to face the following challenges:

- Provide developers with task-specific editors and design capabilities, as required from an MC's artifact information.
- Monitor the progress of products, as specified in an MC's artifact model, and identify potential conflicts between artifacts and development activities.
- Provide supportive information and recognize problems, e.g., the violation of guidelines, during the development process, as specified in an MC's guideline model.
- Manage authorization for particular artifacts and products, as specified in a MC's role model. As stated already before, this is out of scope of this thesis.

5.1.2 Workflow-centric Challenges

The support of individual tasks, requires the managed assignment of tasks to developers. Therefore, another challenge is to enable an automated process coordination. A framework, which manages the activities of developers, monitors active tasks, and provides context information, must face the following challenges:

- Efforts for workflow management and guidance must be in proportion to the efforts of actual product development, i.e., the framework must not increase efforts for doing a job.
- Process mining and subsequent reviews can improve future projects sustainably. Therefore, workflow management must enhance gathering of information throughout the process.

- Synchronize the development process with actual needs of the product under development, i.e., enable situational assignment of development activities.
- The framework must interpret the process to guide the activities of developers and to ensure compliance with standard processes, while not restricting their creativity.

The described challenges are faced in the following by describing the realization of technical process design information, as introduced in chapter 4, on operational level.

5.2 Overview: Method-driven Guidance of Development Processes

In chapter 4, we introduced an approach to detail the information of a workflow and the particular fragments of an MC, such as artifacts, editors, and guidelines. In combination with our SPLE approach (chapter 3), we are now enabled to configure computer-interpretable process models for the situation at hand. That way, the chapters faced *objective* 1 and *objective* 2, as introduced in Section 1.2. In this chapter, we focus on *objective 3* and discuss the application of the designed process information on operational level.

We start with MF information, where we use model transformations to transform computer-interpretable MF information into a platform-specific/operational format. In Section 5.3, we describe an M2T transformation to generate method-specific editors, which are capable to provide developers with relevant information and method-specific capabilities, using MMVs and associated EUSA information. In Section 5.4, we introduce an observer mechanism, which monitors performed design activities to establish a history of an artifact's content, using MMVs and other method-specific context information. The information about an artifact's content is reused in Section 5.5, where we describe an M2M transformation and the interpretation of guidelines to enable a purposeful validation of artifacts during the development and to provide developers with adequate information. Finally, in Section 5.6, we detail a flexible workflow management, which considers the consistency between artifacts. Thereby, monitored artifact information and particular consistency rules are used to automatically control the process in a flexible way. We conclude the section with a case study in Section 5.7.

5.3 Method-driven Editor Generation

From the process model extended with technical information, method-specific user interfaces with editing capabilities can be generated automatically. Although, it is possible to use Commercial off-the-shelf (COTS) editors and to adapt their functionality using, e.g., MMV information, we describe a generic transformation to achieve completely customized editors based on the defined process, i.e., contained MC information. Therefore, we use the platform-independent information of an MMVT, as annotated with an

MC's input and output artifacts (cf. Section 4.5), to generate editors for different platform-specific target languages. An MMVT restricts a meta model to relevant elements and defines rules for a correct application of that elements. As a result, a generation is enabled to provide a resulting editor with the following capabilities:

1. The visualized part of the meta model is restricted according to the information provided with the MMVT.
2. An appropriate action set is provided to realize the annotated EUSAs.
3. The Editor considers the MMVT-specific navigation of model elements.

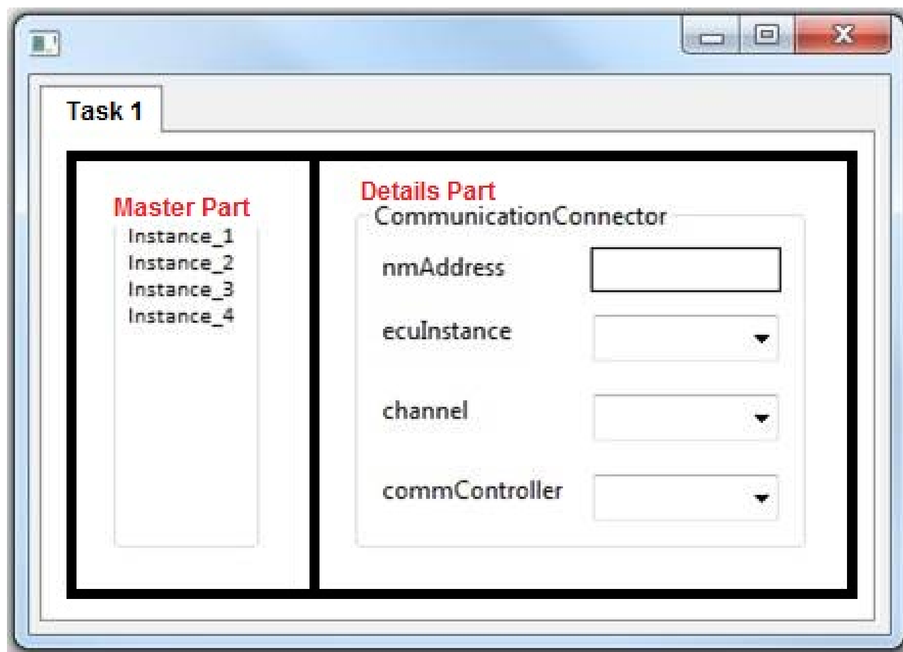


Figure 5.2: Example of a Generated Master-Detail Editor

The following section deals with the automated generation of such an editor using the MMVT in combination with platform-specific code templates. We describe the general generation using the example of XML Widget Toolkit (XWT), by which we generate editors realizing the master-detail pattern, as discussed in Section 4.5. A resulting editor is exemplified in Figure 5.2. After, we discussed the generation of *structural code* to visualize relevant elements in a master-detail structural relationship, we describe how functionality is added to the editor by generating so-called *behavioral code*. Structural code generation and behavioral code generation result in the editor for one MC. As all MCs, which are contained in a process, are provided with an individual editor, the generation runs repeatedly to provide editors for the whole process. Finally, we discuss some of the arisen challenges and identified solutions.

5.3.1 Generation of Structural Editor Code

A generated editor's structural code consists of various so-called Basic Structural Element to organize the different model elements according to underlying meta model relationships. Basic Structural Elements (BSEs) visualize individually typed model elements, which either are embedded into superior (master) or inferior (detail) Basic Structural Elements. The BSEs are grouped into two categories: Visual Structural Elements (VSEs), which enable the visualization of typed model information, such as a *list viewer*, and Functional Structural Elementss (FSEs), which enable the realization of an action, such as a *Button*. For the generation of BSEs following an MMVT's master-detail design, the MMVT is traversed top-down, while a declarative M2T transformation generates nested master and detail parts to aggregate the BSEs correspondingly, as exemplified in listing 5.1.

```
1 createMasterDetailHierachy(MMVT tree){
2 //A variable to store nodes to be visited
3 //and associated BSE information
4 Dictionary<MMVT_Node,BSEInformation> currentMaster  :=
5 new Dictionary<MMVT_Node,BSEInformation>();
6 //Create Basic Structural Element for the root node
7 BSE masterBSE =createBSE(tree.root);
8 //Add the root node and associated BSE information to visitable node
9 currentMaster.add(tree.root,masterBSE);
10 while(currentMaster.notEmpty()){
11         for(MMVT_Node master : currentMaster.next()){
12         //get all children node of a visited node of the MMVT
13                 for(MMVT_Node child : master.getChildren){
14                 //Create BSE information for the children node,
15                 //as detail for the visited master node
16                         BSEInformation detail := createBSE(child);
17                         //Related details with a master part
18                         master.addDetail(detail);
19                         //Next, children nodes must be visited
20                         currentMaster.add(child,detail);
21                 }
22                 currentMaster.remove(master);
23     }
24   }
25 }
```

Listing 5.1: Master Detail Definition

Due to the tree-structure of the MMVT the traversal can start from a root node to generate BSEs in a master part, which is refined into an inferior details part to realize the BSEs of its children nodes in a second step. Depending on the depth of the MMVT this is repeated several times until a leaf node is reached. Each time an MMVT node is visited, BSEs to be generated depend on a node's EUSA: while read-only-annotated nodes force the generation of a VSE exclusively, other EUSA-annotations force the generation of an additional FSE.
By traversing the MMVT, a visited MMVT-Node provides an BSE with relevant meta

model information, such as type information, attributes, associations, or cardinalities. This information is required to provide a so-called *Databinding*, which enables an BSE to process a respective model element. Therefore, we provide each BSE type with a generic template, which is completed using meta model information provided during the traversal.

```
1 <ListViewer Name=<ListViewerName>
2  DataContext=<ContextObject>
3  ItemsSource={Binding Path=<ItemsPath>}
4  IsEnabled=<!IsReadOnly>
5  ...
6  />
```

Listing 5.2: ListViewer Template

For example, while listing 5.2 illustrates an XWT-specific template, which is associated with an VSE to represent a list of objects, i.e., to visualize a set of associated objects, listing 5.3 exemplifies the completed template. The completed template realizes a list viewer to visualize objects typed with the meta model class *Function*. These objects are aggregated by an *Functional Analysis Architecture (FAA)* object, following the meta model introduced in Figure 4.22.

Depending on the current node in the MMVT the variable parts of the template (<ListViewerName>, <ContextObject>, <ItemsPath>, and <!IsReadOnly>) are replaced by values provided by the MMVT-Node. The *ListViewer Name* variable is used to assign the generated behavioral code to the VSEs it is responsible for. The *DataContext* variable refers to the object, which aggregates the data, that are visualized by the means of the list viewer. The *ItemsSource* specifies an optional path within the *DataContext* to the displayed data. The *isEnabled* variable indicates, whether or not visualized data can be modified or not.

```
1 <ListViewer Name=FAAListViewer
2  DataContext=FAA
3  ItemsSource={Binding Path=Function}
4  IsEnabled=True
5  ...
6  />
```

Listing 5.3: Example: Completed ListViewer Template

5.3.2 Generation of Behavioral Editor Code

By now, editors only visualize artifacts according to MMV information. To provide the visualizing parts of the editors,i.e., BSEs, with relevant functionality (or modeling capabilities) they must be complemented with that functionality. Therefore, handler parts add capabilities to FSEs, which are associated with an individual BSE to process contained model elements. The capabilities can be categorized into three different types of actions:

- *Attribute Modify* is the modification of an object's attribute typed with a primitive type, such as String or Integer. Every time, a change is made in the attribute's VSE, the action changes its associated object's attribute in the same way. For example, the deletion of text in a text box leads to the deletion of its associated object's string attribute.
- *Reference Modify* adds/removes appropriately typed objects to/from a list of associated objects of another object.
- *Containment Create/Delete* initializes a new object, or deletes an existing one within the context of a parent object. If the action creates a new object, then it is added to the parent object's containment association, vice versa, if the action deletes an object, it is removed from the parent object's association.

Similar to VSEs, every action, i.e., an FSE, needs to be associated with affected meta model information. Therefore, individual FSEs realizing a specific EUSA are provided with generic templates, likewise. The action templates are completed using information from a respectively visited MMVT-Node. For example, to customize an *Attribute Modify*-Template, the relationship to the parent object of the attribute it is modifying has to be identified. In contrast, *Reference Modify*- and *Containment Create/Delete*-Templates need more efforts to customize. Therefore, the customization of a *Reference Modify*-Template is exemplary depicted in listing 5.4 and listing 5.5.

```
1  public void addReferenceTemplate(){
2  <ReferencedElementType> toAdd = <ReferenceVSE>.getSelectedElement();
3  <CurrentElementType> addedTo =
4    <CurrentElementVSE>.getSelectedElement();
5  if(<ReferenceCardinality> != 1){
6          addedTo.<ReferenceName>.add(toAdd);
7  }
8  else{
9                  addedTo.<ReferenceName> = toAdd;
10         }
11 }
```

Listing 5.4: ReferenceAdd Handler Template

A *Reference Modify* action needs information about different VSEs and associated meta model information. Therefore, the respective template has variable parts to bind relevant information. This is exemplified in listing 5.4. While the <ReferencedElementType> is a type information to identify the object type, which has to be referenced, the <CurrentElementType> provides type information to identify the referencing object. Additionally, the <ReferenceName> refers to the association name, which relates the <CurrentElementType> with the <ReferencedElementType>. Finally, <ReferenceCardinality> represents the cardinality of the reference, commonly INFINITY if the reference is a list or ONE, otherwise. As illustrated in listing 5.5, the variable parts are replaced by the respective information of the MMVT-Node, such as class names, e.g., *Function* or *FAA*, or VSE names, e.g., *FunctionListViewer* or *FAAListViewer*. Depending on an association's cardinality the template manages a single associated object or a list of associated objects.

```
1 public void addReference(){
2         Function toAdd = FunctionListViewer.getSelectedElement();
3         FAA addedTo = FAAListViewer.getSelectedElement();
4         if(INFINITY != 1){
5                 addedTo.functionList.add(toAdd);
6         }
7         else{
8                 addedTo.functionList = toAdd;
9         }
10 }
```

Listing 5.5: Example: Completed ReferenceAdd Handler

Customizing a *Containment Create/Delete*-template is similar to the customization of a *Reference Modify* template. The *Containment Create/Delete*-template also needs type information about referenced objects and about the referencing object. Adding and removing of objects is handled the same way as in reference templates, depending on the containment cardinality.

5.3.3 Challenges and their Solutions

When realizing our editor generation framework, we faced different challenges, which we discuss in the following to reason the way of the above approach.

First, we started without having the concept of the MMVT and a model transformation was directly applied to the MMV. As the algorithm strictly followed the paths given by associations between meta model elements to generate BSEs, it sometimes ended up in an infinite loop, when it came to cyclic associations between meta model elements. The MMVT resolves this cyclic associations with its hierarchical structure. A former cyclic association is now resolved to a linear succession of meta model elements. As the transformation now works on MMVT, the algorithm only loops as long as it does not hit a leaf node within the MMVT. Therefore, the execution time of the algorithm is determined by the depth of the MMVT.

Second, before introducing the MMVT, it was impossible to independently annotate different meta model classes, that derive from one common super meta model class, with different EUSAs. Therefore, if a super-type class was annotated with *read-only*, all derived sub-classes were *read-only*, as well. On the one hand, by introducing the additional indirection of the RD, the MMVT enables the explicit annotation of inheriting meta model classes. On the other hand, the MMVT made it possible to restrict the inheriting meta model classes to a particular subset.

With all the benefits that come with a MMVT, there is also drawback. The declarative model transformation approach proved to be more complicated, when it comes to the root of the MMVT. As the MMVT only describes a cutout of the original meta model, an

artificial root node is created. Defined actions, such as *Reference Modify* or *Containment Create/Delete* can not be implemented for the root node as realized for children or leaf nodes. As root elements do not have a defined parent node, to which new objects can be added to or removed from, we restricted root elements of the MMVT to be annotated with the EUSA annotation $read\text{-}only$ exclusively.

5.4 Artifact-specific Information Management

As discussed earlier in Section 4.4.3, having information about an artifact's content, would enable a more detailed validation, change impact analysis and traceability on data element level, as well as, a more flexible workflow management.

The challenge is, that an elements's data type, as annotated with artifacts using MMVs, is not sufficient to decide about an element's unambiguous membership to an artifact. For example, given two artifacts, called *middleware design architecture* and *application design architecture*, which were both created using the same data type, called *SoftwareComponent*. In that case, it is not possible to query the physical model for *SoftwareComponent*s, which exclusively are assigned to either the *middleware architecture* or the *application architecture*. Instead, querying the model would result in a general set of *SoftwareComponent*s, unless we would have additional context information about the *SoftwareComponent*'s membership. This can be avoided by providing explicit design guidelines, which prescribe the structure of the overall models considering the artifacts' needs. However, the definition of these guidelines is a time consuming task and the manual association of model elements to affected artifacts proves to be difficult, likewise.

Therefore, the following deals with automated assignment strategies to associate elements of physical models with virtual artifacts. However, since model elements are used in a specific context of the development process, this kind of membership relationship can only be derived at process runtime, i.e., a process model can not be provided with this kind of information at design time. Therefore, we identified the following two alternative strategies to derive necessary information at process runtime:

1. Modeled elements manage the knowledge about their artifact membership
2. Artifacts manage the knowledge about associated modeled elements

The first alternative has two drawbacks: To identify an artifact's content, all elements must be queried for their membership, which is very inefficient. Furthermore, the identification of artifact content would require, that meta model elements are extended with an additional property to reference a respective artifact. As this proves difficult for existing or standardized meta models, such as UML or AUTOSAR, we rejected the first alternative.

Instead, the second alternative is a more promising approach. Using this strategy, an artifact must be provided with information about member model elements. Therefore, at process runtime, a workflow management system allocates development tasks

in the form of MCs, which in turn provide detailed artifact and editor information via associated MFs, and so determines the actual process context. That way, the monitoring of editor commands in combination with available context information, in particular, associated MMVs of the artifacts, provides sufficient information to observe the instantiation or modification of model elements and to assign this information with responsible artifacts. The membership indications available at process runtime, are:

- The data type of the model element instance to be assigned
- The unique identifier of the model element instance to be assigned
- The editor's action command performed to read, modify, or create the respective model element instance
- The restricted set of artifacts and associated MMVs influenced by a touched model element instance.

The monitored information enables the assignment of elements to artifacts, as detailed in the subsequent section, which is the prerequisite, that enables the extraction of an artifact's content to validate contained information, as discussed in Section 5.5, and to trace dependent artifacts and tasks of a development process, as discussed in Section 5.6.

We start with the introduction of the *Artifact-Observer-Pattern* developed to provide a lightweight communication mechanism between artifacts and respective generated or third-party editors. Afterwards, we detail generic strategies to assign model elements to responsible artifacts.

5.4.1 Artifact Observer Mechanism

The artifact observer mechanism is based on the conventional observer pattern, as introduced by Gamma et al. in [GHJV95]. Its general structure is depicted in Figure 5.3, where a subject and a set of observers are defined. Inspired by the approach applied in [CHCC03], we adapted the pattern as follows:

The subject, which represents the object of interest, is the command, which is performed in the context of an editor to process a model element instance. Because of the

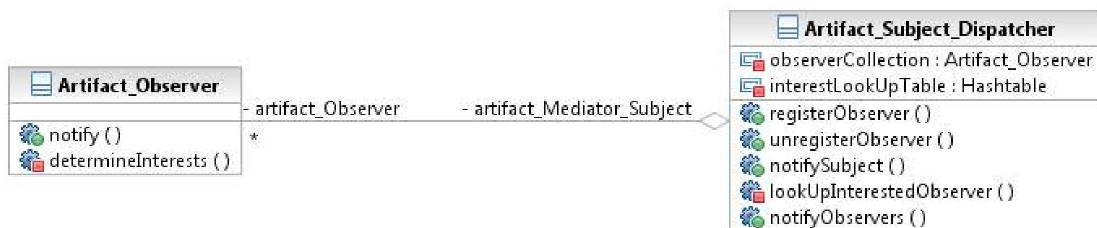


Figure 5.3: Artifact Observer Mechanism

overhead, to provide editors with the information of interested artifacts (in particular for third-party editors), we introduce an intermediate *Dispatcher* component, which serves us as subject in the context of our *Artifact Observer Pattern*. The *Dispatcher* provides editors with a light-weight central communication interface to notify interested observers, i.e., artifacts, about performed modeling events.
An artifact used during the process, is considered as an observer object, that indicates interests in particular modeling events. These interests, basically, are defined by the associated MMVs and associated EUSA information. Therefore, an artifact object implements the *Artifact_Observer* interface, by which it can be notified about a particular modeling event using the *notify* method. The *Notification Event* must provide a six-tuple composed of the following information entries:

- Command: The type of the action accomplished by using an editor. Therefore, we extend the EUSAs, as defined in Section 4.5.2, and differentiate four types of modeling actions, namely the element $read\text{-}only$, $modify$, $create$, and $delete$ (a derivate of modification).
- DataType: The name of a model element's data type, which is affected by a modeling event (i.e., the command).
- ElementID: An unique identifier of the model element instance, which was processed by the current modeling event.
- MethodID: An identifier, which allows to relate modeling events with a responsible MC. As generated editors exclusively support a specific MC, as discussed above, an editor's identifier allows to identify an MC unambiguously, as well.
- GroupingElementID: Since methods can be applied in the context of different scenarios, a unique identifier must be transmitted to assign a method to its corresponding GE (cf. Section 3.3) uniquely.
- TimeStamp: Each notification must provide a time stamp. The time stamp is used to bring various events into an order, which is convenient for traceability and change impact analyses, as discussed in Section 5.6.

A *Notification Event* containing above information, must be assigned with an artifact following the strategies as discussed next. To realize this assignment, we introduce a relation, which assigns an event to an artifact, as follows:

Definition 30 (*Assignment of Notifications to Artifacts*)

Let mc be an MC, the current method in which a notification was created, and let $ArtifactID$ be an identifier of the artifact, which is interested in a notification. Let n be the *Notification Event* to be assigned. Then n is associated with the corresponding artifact of an MC via the relation $NotificationAssignment$:

$$NotificationAssignment \subseteq NotificationEvent \times MC \times ArtifactID \tag{5.1}$$

Based on this relation, the function

$$
\begin{aligned}
&artifactHistory := ArtifactID \to \mathcal{P}(NotificationEvent)\\
&artifactHistory(artifact) := \{n | (n, mc, artifact) \in NotificationAssignment\}
\end{aligned}
\tag{5.2}
$$

results in the history of artifact-specific notifications, which subsequently can be used to trace an artifact's content during the development.

While editors provide the *Dispatcher* with *Notification Events*, artifacts register their interests for particular modeling events at the *Dispatcher*. Based on this, the dispatcher analyzes received messages with regard to registered observers and their interests, before it distributes the editor events to interested observers, i.e., artifacts. This interaction between observer, dispatcher, and editors is illustrated in Figure 5.4 and works as follows:

1. If a workflow continues and switches from a $Task_n$ to a subsequent $Task_{n+1}$, a former subscription for interests of $artifact\ observer_n$ becomes obsolete. Therefore, $artifact\ observer_n$ unregisters all of its interests by calling the respective *Dispatcher* method. The figure outlines it in 1. (Note, that this first step is not required, if no preceding task is defined, e.g., at the beginning of the development process.)

2. Before $Task_{n+1}$ is performed by an user using an associated editor, all context information of the corresponding MC must be made available to the dispatcher. Therefore, the output artifacts ($artifact\ observer_{n+1}$) of $Task_{n+1}$ are queried for associated MMVs. To realize this, the function $MMVofArtifact(artifact\ observer_{n+1})$ results in $S\ |_\chi$, i.e., the associated MMV of $artifact\ observer_{n+1}$.
   $ViewConcepts(S\ |_\chi)$ is subsequently used to extract a list of contained view concepts $c_1 \cdots c_n$, which are assigned with particular EUSA values, as discussed in Section 4.5. For the resulting concepts the artifact registers an EUSA-specific interest by calling the dispatcher's function
   $registerObserver(c_x, getEUSAFromConcept(c_x, mc, S\ |_\chi), artifact\ observer_{n+1})$
   (with $x \in 1..n$), as outlined in the figure at 2. After all concepts ($c_1 \cdots c_n$) are registered, the dispatcher is provided with all interests of $artifact\ observer_{n+1}$, which enables the dispatcher to assign performed editing events with the corresponding artifact. Therefore, for each relevant artifact observer, the dispatcher stores the necessary interests in a look-up-table.

3. The third step in the scenario, is the subject's notification from the editor. Therefore, an editor uses the *notifySubject* method of the dispatcher and passes information about an affected model element (i.e., its unique instance identifier and its instance type name) and the executed action (i.e., read, modify, delete, or create). (Note, that for generated editors (cf. Section 5.3), we can simply integrate this notification functionality during the generation phase. For other COTS editors, we must extend the editor's functionality manually).

4. The dispatcher is aware of the currently performed MC and associated context information. By querying the look-up-table of interests for a notified element type, the dispatcher is enabled to decide, which observer gets informed about a notified

modeling event. As a result, in step 4, the observer notifies all artifacts according to their interests, as registered in step 2. However, there are additional challenges influencing the association of modeling events to an artifact's history, as discussed in the following section.

5.4.2 Artifact Element Assignment Strategies

Basically, an MMV's concept, for which an artifact indicates an interest at the dispatcher, corresponds to the data type of an element processed by an editor. Therefore, the assignment of notifications is based on matching the *DataType* of an editor's notification message with registered interests. Based on the matching, the assigned notifications set up an artifact-specific history of modeling events. However, to realize this, the following assumptions must be ensured:

- All artifacts must be uniquely distinguished from other artifacts. Otherwise, it is not possible to assign an element to an artifact unambiguously and to retrieve artifact-specific information for further analyses. This property can be fulfilled by unique artifact names.
- Each model element must provide an unique identifier, by which it is related to an artifact's identifier.
- All MMVs connected with output artifacts of the same MC, must be disjoint regarding their contained meta model elements. (Input artifacts do not influence this property.) Otherwise, if there is more than one output artifact referencing the same meta model element, we are not able to assign an instance of that element unambiguously.

However, since some model elements are relevant to different artifacts, elements can not be assigned to an artifact exclusively. To face that challenge and the obstacles, which were discussed in Section 4.4.3, we define strategies for a clear assignment of elements to artifacts. In addition to the assumptions made above, the strategies depend on three criteria:

- the MMV concept, i.e., the meta model class, which defines the type of model elements, which have to be assigned. This is the element type, for which the artifact registers an interest. By default, as input information are *read-only*, the output artifacts of a currently performed MC register their interests exclusively. However, the interests can be adapted to elements of an individual input artifact using the *FromTo* relationship.
- the *FromTo* information, as described in Section 4.4.3. Using the *FromTo* relationship allows to relate a concept of an MC's output artifact with an equally typed concept of an input artifact (equally means, that the two MMV concepts refer to the same meta model class). As a result, an output artifact indicates an exclusive interest for notifications, which concern the data elements of the related input artifact. Relating

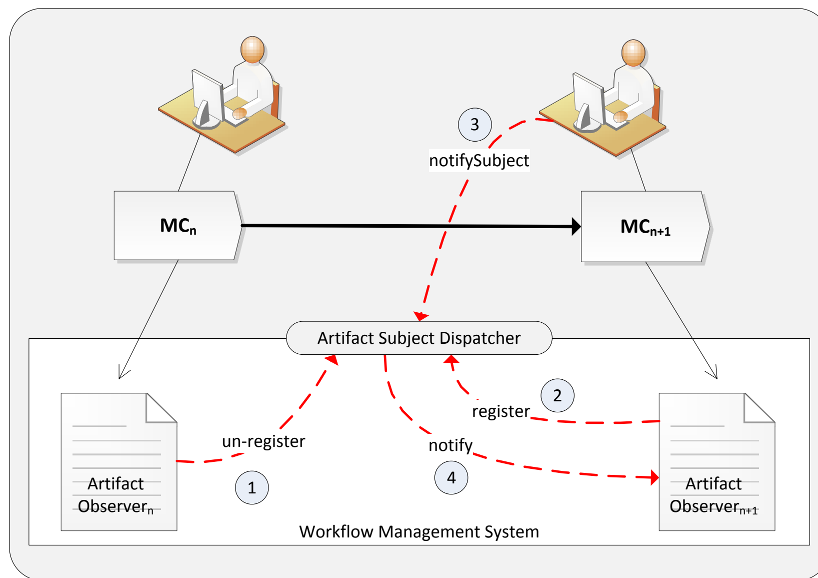


Figure 5.4: Artifact Observation

artifacts using a *FromTo* relationship means, that (in addition to newly created elements) only those notifications are relevant for the output artifact, which concern the elements already assigned with the input artifact. In parallel, this causes a synchronization need between the output artifact and the related input artifact, i.e., the output artifact and the input artifact register an equal interest.

- the *EUSA*, which is annotated with the artifact's concept to specify the editor command an artifact observer is interested in. However, there is one exception, if a concept's EUSA is *read-only*, while one of its features is modifiable. The feature's EUSA applies to the superior concept, since the modification of a single feature affects the overall object.

We distinguish strategies for the initialization phase of an MC, i.e., when a context switches from a $task_n$ to $task_{n+1}$, from strategies, which are applied during the performance of an MC. For both phases, we further distinguish two scenarios: While the first scenario refers to a defined *FromTo* relationship between an element of an MC's input and an output artifact, the second scenario addresses unrelated output artifacts. Both phases are detailed and respective strategies are explained in the following. We start with the initialization phase, whose strategies are summarized in Table 5.1.

Some model elements belong to an artifact regardless whether or not they are modified or created during an MC. They are part of an artifact, as they provide the artifact with contextual or structural information, which can not be derived from performed editor commands only. For example, if the scope of an MC is to refine elements, which are provided by an input artifact, not only the elements, which are modified during the MC, may belong to the output artifact, but the complete set of potentially modifiable el-

	Output Type	FromTo	EUSA	Assignment @ Initialization
1	mmv:concept::x	$from :: input_x$	read	only the elements from $input_x$ typed with mmv:concept::x
2	mmv:concept::x	$from :: input_x$	create	only the elements from $input_x$ typed with mmv:concept::x
3	mmv:concept::x	$from :: input_x$	modify	only the elements from $input_x$ typed with mmv:concept::x
4	mmv:concept::x	-	read	all available elements typed with mmv:concept::x
5	mmv:concept::x	-	create	all available elements typed with mmv:concept::x
6	mmv:concept::x	-	modify	all available elements typed with mmv:concept::x

Table 5.1: Artifact Element Assignment Strategies @ MC's Initialization

ements. Thus, if a workflow continues and the context switches from one MC to the next one, some preparing steps are required in order to assign individual elements of an input artifact to the respective output artifacts. This is realized before any notification is sent from an editor to the dispatcher.

The required assignments depend on the information specified for respective MCs during the design phase (cf. chapter 4). In particular, it depends on the optional *FromTo* relationship between an input and an output artifact, and the EUSA attribute of MMV concepts. To demonstrate the strategy, we discuss the two scenarios with regard to a specific EUSA:

1. For the first scenario, an output artifact, i.e., one of its associated concepts typed with, e.g., *mmv:concept::x*, is related to an input artifact $input_x$ via the *FromTo* relationship. As a result, since the output refers to elements of the related input artifact exclusively, all elements of $input_x$ typed with *mmv:concept::x* belong to the referencing output artifact regardless of the annotated EUSA. This is summarized in Table 5.1 (row 1-3).
2. In the second scenario, the output artifact, i.e., one of its associated concepts typed with *mmv:concept::x*, does not define a relationship to a particular input artifact. This implies, that all available model elements typed with *mmv:concept::x* are assigned with the output artifact. This is summarized in Table 5.1 (row 4-6).

After existing objects are assigned correspondingly, we detail the strategies for the assignment of notifications caused by performed modeling events, i.e., when an MC is performed and the dispatcher gets notifications from an editor. This is summarized in Table 5.2:

If a developer performs an MC to produce an intended output, data or model elements are changed using a method-specific editor. Existing elements are modified or deleted, while new ones are created and modified. This changes the content of an individual artifact and has to be assigned accordingly. Therefore, an artifact observer has to

	Output Type	FromTo	EUSA	Interested Assignment Events
1	mmv:concept::x	$from :: input_x$	read	cf. initialization
2	mmv:concept::x	$from :: input_x$	create	the input artifact $input_x$ and the output artifact indicate interests for the creation & modification of elements typed with mmv:concept::x, which belong to the output artifact
3	mmv:concept::x	$from :: input_x$	modify	the input artifact $input_x$ and the output artifact indicate interests for the modification of elements typed with mmv:concept::x, which belong to the output artifact
4	mmv:concept::x	-	read	cf. initialization
5	mmv:concept::x	-	create	creation & modification of elements typed with mmv:concept::x
6	mmv:concept::x	-	modify	modification of elements typed with mmv:concept::x

Table 5.2: Artifact Element Assignment Strategies @ Runtime

register additional interest, as discussed in the follwoing.

1. For the first scenario, the assignment depends on the *FromTo* relationship and the EUSA of an output artifact's concept typed with, e.g., *mmv:concept::x*. *Read-only* elements by default must not be modified or created during the performance of an MC. Therefore, no editor command is expected to be assigned at runtime, and the only relevant elements are assigned during the initialization phase of the MC.
   Elements, whose data type or concept is annotated with EUSA *modify*, are modifiable. During the initialization phase, relevant elements of an input artifact ($input_x$) were assigned with the output artifact according to the *FromTo* relationship. At runtime, no new elements must be assigned, but modifying notifications concerning elements of typed with *mmv:concept::x* are assigned exclusively, i.e., only input artifact elements referenced by the *FromTo* relationship are relevant for the assignment. As modified elements belong to both artifacts, i.e., the input artifact and the output artifact, they are notified about modification events by registering a respective interest, in parallel.
   In contrast, elements, whose concept is annotated with the EUSA *create*, can be created and modified. Due to the *FromTo* relationship, the notifications concern both elements, i.e., existing elements of the input artifact and newly created elements, and must be assigned with the input and the output artifact by registering a respective interest. This is summarized in Table 5.2 (row 1-3).

2. In the second scenario, no *FromTo* relationship is defined between the concepts of an output and an input artifact. That means, elements of an artifact are not restricted to the elements of an input artifact. Instead, depending on a concept's EUSA, a respective output artifact indicates interests for all elements of a particular data type

of the corresponding MMV concept, in general.
For *read-only* elements, only the initialization phase is significant, i.e., no interests are registered, since no editor commands are expected. If an element is annotated with *create*, the creation of such elements and the modification of accordingly typed elements are relevant for the assignment. Therefore, an output artifact registers two interests. One for modification events on elements typed with the concept data type, and another one for creation of respectively typed elements.
For concepts annotated with the EUSA *modify*, the output artifact registers an interest concerning the modification respectively typed output elements. This is summarized in Table 5.2 (row 4-6).

We discussed, for which modeling event an artifact may indicate an interest depending on particular MC information. If a notification is assigned with an artifact, the affected element belongs to that artifact implicitly. This information subsequently can be used to identify the concrete content of an artifact, e.g., for artifact-specific evaluation of a guideline. Additionally, the above strategies enable us to monitor method-specific artifact changes and the usage of contained elements. Thus, we monitor the context of an element's reading, modification, or creation. These information can subsequently be analyzed to derive the impact of particular events on distinct artifacts and workflows, and to possibly take proper actions.

5.5 Situational Model Validation - Guideline Application

Common CAx Tools support developers in providing mechanisms to define customized constraints to validate their models or other development artifacts. However, all these tools manage one global set of constraints to validate particular data formats without considering any relationship between method-specific guidelines on associated artifacts.

In contrast to a global set of constraints, local validation applies an appropriate set of guidelines/constraints, i.e., individual constraints are only applied in a context of relevant methods and their associated data (artifacts). This results in miscellaneous advantages:

Enhanced guideline management: While general constraints are managed within a global set, they are normally difficult to adapt. Therefore, we propose to connect guidelines directly with the affected method in the process model. Thus, adoptions can be made local for specific methods or sections of the overall process. As guidelines are specific to some process part, the number of guidelines, which are required to a specific process period, can be reduced to a manageable and minimum set.

Advanced compliance & traceability capabilities: By applying particular guidelines at specific points of a managed development process, the accomplishment of required tasks and associated rules can be ensured more effectively. For example, if mandatory constraints are ensured before continuing the process, negative or inconsistent states of a

product can demonstrably be prevented. Additionally, if constraints are associated with the particular steps of a development process, the traceability of individual product states and sources of errors can be identified.

Automated constraint (re-)generation: Although, developers are capable to define constraints using languages, such as Java, OCL or EVL, the produced code is often difficult to read or hard to manage. Thus, adapting constraints afterwards can be a time-consuming task. Based on the more comprehensible constraint design, which we described in Section 4.6, code can be generated for a specific target language. As the graphical constraint abstraction is more understandable, also the model can be adapted more simple to re-generate constraint code and to save time.

Contextual validation: Current practices using global constraints do not consider, that in certain situations particular constraints are more reasonable than in other ones. The result of a constraint can provide information, which is not necessary for the moment when it is evaluated. For example, if a function designer aims at evaluating a design architecture composed of different functions he is working on, he would be interested in a validation of functions he is responsible for. However, if there is a context-free constraint, which implicitly requires the role of a timing analyst to associate functions with timing information, this constraint is also validated and results in an error provided to the role of the function designer. Thus, if constraints are context-specific, it not only reduces the number of constraints to be checked, i.e., it saves processing power and memory, but it enhances the result of the respective validation and makes it more usable for developers' situation.

Such local constraints perfectly match with guidelines and statements of MCs, which were introduced in Section 4.6. Therefore, in Section 5.5.1, we define a translational semantics using the semantics of OCL to evaluate modeled graphical statements. Based on this, the interpretation (Section 5.5.2) and the effects (Section 5.5.3) of a guideline evaluation in the context of an MC is described.

5.5.1 Guideline Realization

In Section 4.6, we introduced a graphical notation for the definition of statements and complex guidelines. A guideline is a directed graph or control-flow, which consists of statement, action and control nodes. Statement nodes represent a constraint, which is evaluated to ensure individual properties of an MUD. To interpret and to evaluate modeled statements, a concrete statements semantics is required. To formally define the semantics, we adopt a translational definition of the statement. In a nutshell, since the statement syntax corresponds with a subset of the syntax of well-known constraint languages, such as OCL or EVL, we are enabled to follow a statement's navigation paths and transform edges and visited nodes into the respective semantics of the target language.
In the following, we first detail required basic concepts of OCL's abstract syntax. Afterwards, we define the translational semantics of statements using the example of OCL.

5.5.1.1 The Object Constraint Language

As basis for the translational semantics for statements, we first introduce relevant concepts of the abstract syntax of OCL. OCL specifies different expression types, whereof various complex constraints are combined to evaluate individual model characteristics. Basically, OCL defines a set of primitive types as depicted in Figure 5.5 and more specific OCL expressions to represent types <TypeExp>, variables <VariableExp>, Literals <LiteralExp>, and expressions, such as <FeatureCallExp> and <LoopExp>, to evaluate model characteristics or to address individual model elements. The basic structure of OCL expressions is illustrated in Figure 5.6, that also depicts expression types, such as <IfExp>, <MessageExp>, and <StateExp>, which can be neglected in the following.

By the means of OCL's call expression <CallExp>, the particular features of a model

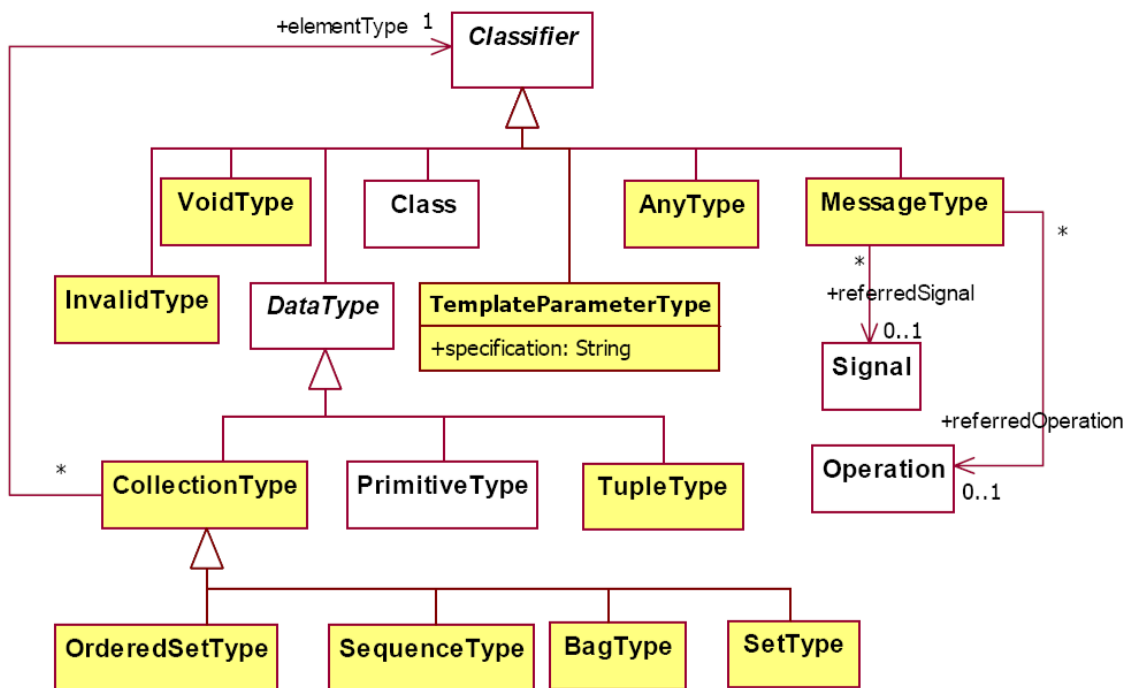


Figure 5.5: Abstract Syntax Kernel Meta Model for OCL Types from [OMG06a]

element can be addressed. Therefore, a source property is an <OCLExpression>, that represents the context element on which the <CallExp> is evaluated. On the other other side, a target <OCLExpression> specifies a constraint, which has to be evaluated for the source element, or a feature, i.e., attribute or association addressed by the source expression. To relate source and target, the <CallExp> expression provides some of operation, which depends on the concrete realization:

While a <LoopExp> iterates on a set of objects to evaluate multiple objects of composite relations inside of meta models, a <FeatureCallExp> refers to features of a meta model element and enables to call a respective property to provide its value for further

processing. <FeatureCallExp> either are <PropertyCall> expressions to extract a specific feature of a meta model class, i.e., association or attribute, or <OperationCall> expressions to relate the result of a source expression with another expressions' result using one of the predefined operations, such as =, <>, first, +, -. To call attribute or association features of a meta model class, a <PropertyCall> expression is used in the context of a meta model class, that targets the respective feature. For <LoopExp> expressions, the source is defined as a <PropertyCall> expression, that calls the association property of a meta model element to provide the set of relevant objects.

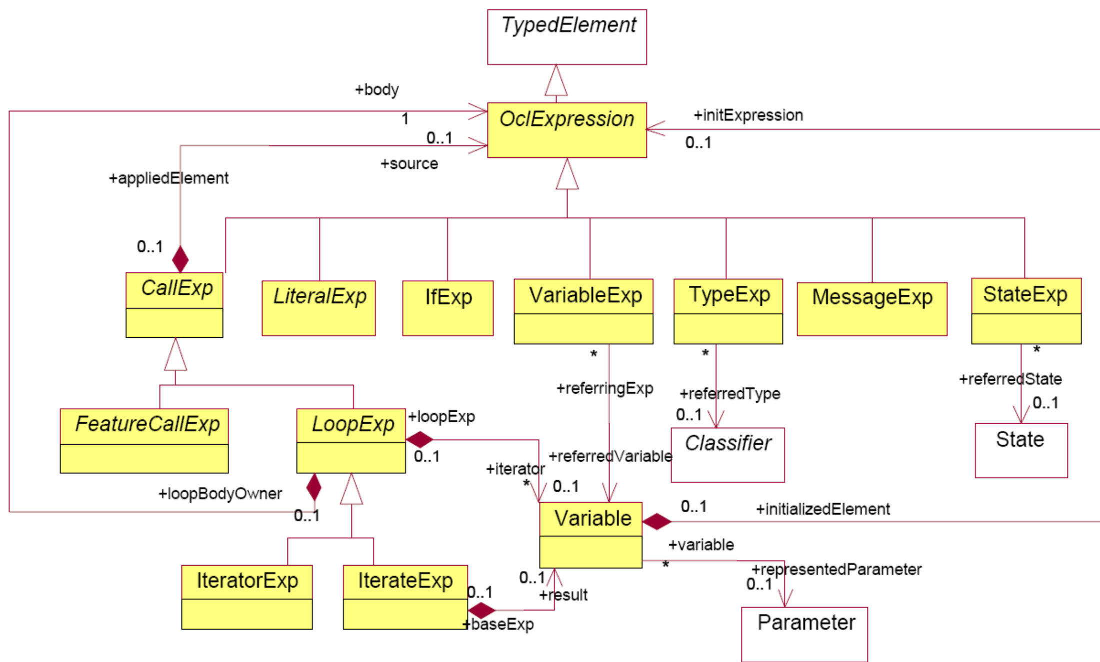


Figure 5.6: The basic structure of the abstract syntax kernel metamodel for Expressions from [OMG06a]

An <LoopExp> iterates over a set of objects and consist of four parts: The *source*, which is starting point to identify the relevant object set, some *iterator variables* to address objects from the set, an *iteration type*, such as forAll, exists, select, or collect, to determine the way objects should be processed and a body expression. The body (*target*) uses iterator variables to express conditions or to define more complex expressions using the iterator variable as context element or source for a nested OCL expression.

To demonstrate OCLs' abstract syntax, we use the example meta model, which is depicted in Figure 5.7: A *system* model element aggregates an *architecture* element, which aggregates some characterized *components*. To express, that all components inside the *architecture* of a *system* must provide a *characteristic* property, whose value is greater than 0,

the following OCL statement is defined:
$self.arch.components{-}{>}forAll(v1|v1.charateristics.value > 0)$. For this statement, Figure 5.8 depicts an abstract syntax tree to exemplify the composition of OCL expressions and the correlation of a source expression with its target expression.

The main expression, i.e., root node, is an iterator expression to iterate all the *component*s within the *architecture* of a *system*. To address a specific *component*, the expression defines an iterator variable, which is not depicted. The iterator is split into a source part and a target part, which is called body here. While the source part <self.arch.components> determines the actual context with the meta model as start for the encompassing iterator expression, i.e., it determines the set of *component*s over which the expression has to iterate, the body expression $< v1.characteristics.value > 0 >$ represents the condition part, which hast to be applied for each *component* during the iteration. The same principle is applied for a *PropertyCall* expressions, as shown in the upper left part, i.e., the source of the iterator expression. The expression defines a source, i.e., the context of the called property, which is a *system*'s *architecture*. To get all *component*s of that *architecture* the target expression, i.e., the referred property, calls the association property *components* of the *architecture* class.

A similar principle is applied to graphical constraint design, where we use concept nodes and navigation edges to navigate within a meta model to identify instances of a particular object by using the *instance* transition and *instance* nodes. For each instance, constraints can be defined which represent the target of the encompassing OCL expression. Thus, as the structures of the OCL expressions and graphical constraints are very similar, we can use it to define a translational semantics, as detailed in the following.

5.5.1.2 Guideline Semantics

To formalize a translational semantics for graphical constraints in the form of statements, as discussed in Section 4.6, we use an inductive way. We start with some simple basic structures, whereof more complex structures can be composed of. The complete definition of the translational semantics is given in appendix B, where we used QVT to transform a statement model into OCL's abstract syntax.

Basically, there are two types of nodes: concept nodes and instance nodes. The first node type serves as context element to identify an object of a specific type to reference its features. The second node type is needed to address and also distinguish individual instances of an respective class. Which instance refers to which class is determined by the $instance$-transition, that relates a concept with various instances, for which constraints can be defined.

Individual constraints either refer to an attribute or an association feature. Attribute features are contained in an instance node to constraint the attribute value of a particular

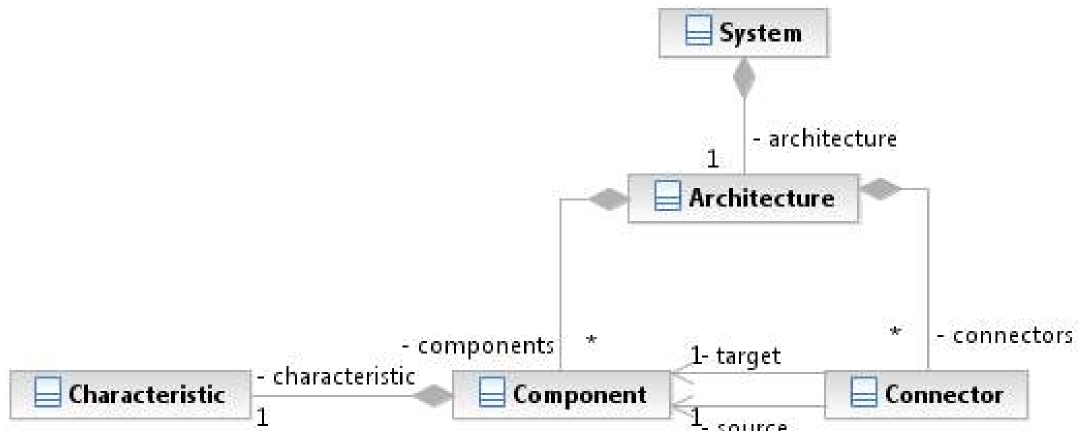


Figure 5.7: Example Meta Model

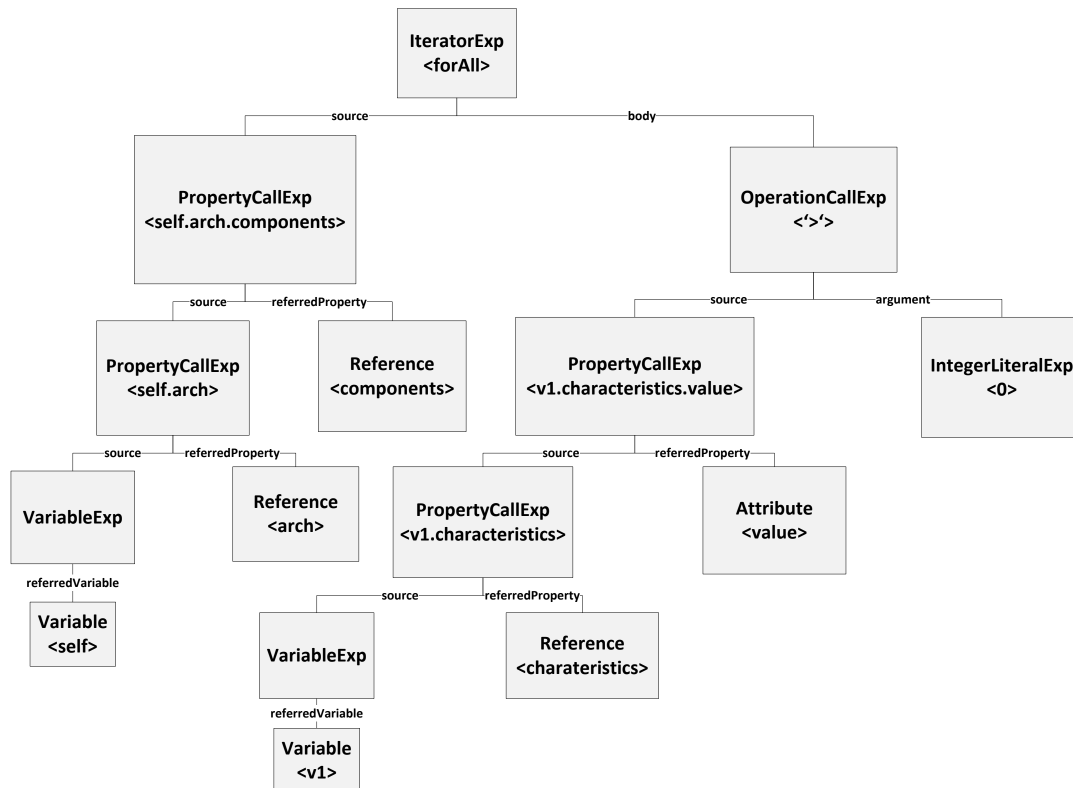


Figure 5.8: Example OCL Abstract Syntax Tree

instance. Association features are represented by a navigation edge between an instance node and a navigation node to address an associated class.
The set of features which is allowed for a particular instance node is determined by the meta model, which forms the basis of a statement. To establish the link between statement elements and particular meta model elements, all nodes, attributes, and edges, except the $instance$ relation, are annotated with a respective meta model element. The information is used during design phase to avoid constraints, which are not conform with the meta model. During the transformation phase, it provides meta model-specific type information about classes and their features.

Instance Identification

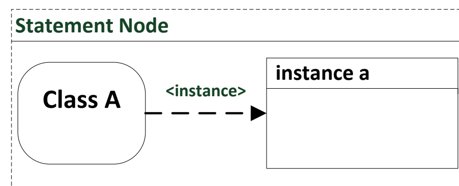


Figure 5.9: Instance Identification

The simplest case is shown in figure 5.9. To address an instance of a class, we use the *NavigationNode* (rounded rectangle) to identify the context of the statement and the *InstanceNode* (cornered rectangle) to identify the instance. In Figure 5.9, the variable $instance\ a$ is identified as instance of $Class\ A$, which is referenced by the *NavigationNode*. The figure also depicts the root or start node, which is characterized by no ingoing edges. The root node is the entry point for each transformation.

Attribute Expressions

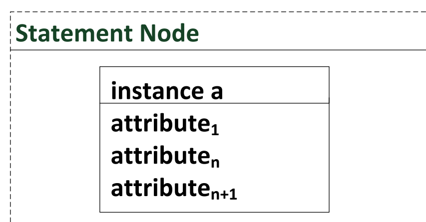


Figure 5.10: Attribute Property Calls

After an instances was identified, its attribute features can be called as prescribed by the underlying meta model. This is depicted in figure 5.10, where an instance vari-

able $instance\ a$ refers to $attribute_1..attribute_{n+1}$. For each feature, the *PropertyCallExp* is used in the context of the source variable $instance\ a$ to address the attribute target, i.e., $a.attribute_1..\ a.attribute_{n+1}$. The context, which is provided by that new expression, is a specific attribute feature, which can be used as source variable for another expression. For example, knowing that context, we can further process the condition, which is associated with attributes of an instance node.

Conditions either can be relational operators for *String* or *Integer* attributes, such as $<, <=, >, >=, ==, <>$, or an operation, which matches *String* attributes with a regular expression. In either case, an *OperationCall* expression is applied. The source of that expression is the attribute context, e.g., $a.attribute_n$, and the argument of that expression is the value, which was specified as reference for the respective attribute. Finally, the operator is set as $referredOperation$ of the *OperationCall* expression, as specified by the *AttributeProperty*. All modeled attribute properties result in multiple expressions $a.attribute_1$ <operation> $value....a.attribute_{n+1}$ <operation> $value$, which afterwards are combined by using an *OperationCall* expression and the Boolean operator <and>. This results in an expression
$A := (a.attribute_1$ <operation> $value_1)$ <and>....<and> $(a.attr_{n+1}$ <operation> $value_{n+1})$

Association Expressions I

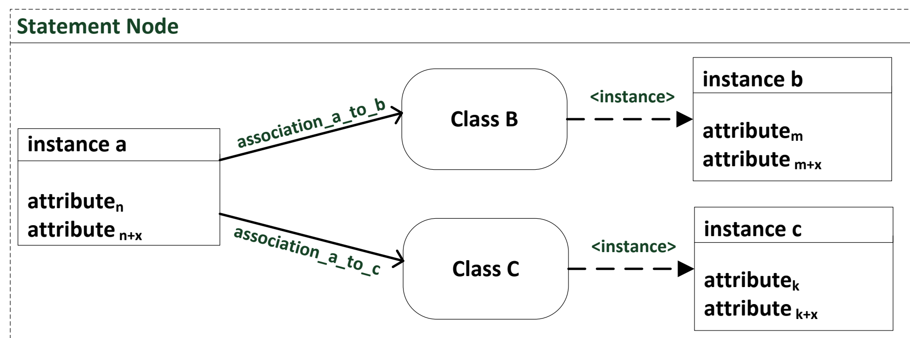


Figure 5.11: Association Property Calls

Beside the definition of attribute features, also association features are defined using the *AssociationTransition* of the guideline meta model, as depicted in figure 5.11. The handling of association features is different from handling attribute properties, as the resulting expression depends on the multiplicity of the associated objects. In particular, a multiplicity greater than 1 requires to set an appropriate iterator function, such as $forAll$, $exists$, or $collect$, which defines the mechanism by which the resulting set of associated objects must be processed.
For each outgoing association feature, i.e., $association_a_to_b$ and $association_a_to_c$, which relates an instance with an associated class (*NavigationNode*), we apply the *PropertyCall* expression using $instance\ a$ as source to call its association feature as target. Similar to attributes, we get the following *PropertyCall* expressions:

$a.association_a_to_b....a.association_a_to_c$. As a result, we get a new context, which is represented by the respective *NavigationNode*s, i.e., $ClassB$ and $ClassC$. To express a constraint for a specific instance of that class, the $instance$ transition allows us to address a particular instance and to make statements for the features of that instance. For example, if the class, which is referred by $association_a_to_b$ in the guideline defines attribute features, e.g., $attribute_m....attribute_{m+x}$, a *PropertyCall* expression can be generated to call that property. This results in multiple expressions, which are combined via the Boolean operator <and> in an *OperationCall* expression
$B := (a.association_a_to_b.attribute_m < operation > value_m)$ <and>
$(a.association_a_to_b.attribute_{m+x} < operation > value_{m+x})$

Association Expressions II

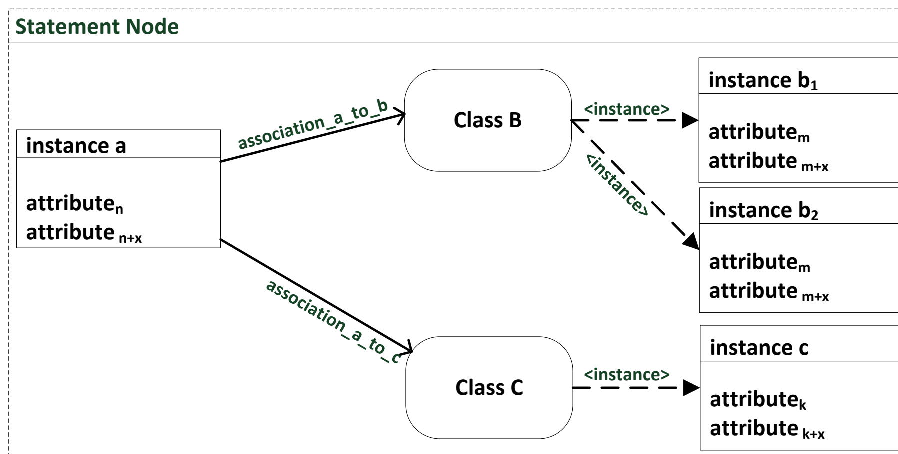


Figure 5.12: Iterator expressions

The more complicated case refers to association properties with a multiplicity greater than 1, as they are implemented as Collection type in contrast to the above demonstrated single object semantics, which is implemented as field. This is illustrated in figure 5.12. The collection or set semantics of such associations implies, that all contained elements must be processed. To iterate over a set of objects, OCL provides an iterator expression, which allows different iteration modes <iterMode>, as defined with the respective *AssociationTransition* of a statement. Basically, all OCL-specific iteration modes, such as $forAll$, $exists$, $select$, $collect$, or $include$, can be supported for collection-based associations, likewise.

As said earlier in this section, collection-based navigation associations must be handled differently from the case of single referenced objects. After a *PropertyCall* expression switches the context from an $instance\ a$ to an associated collection of objects, i.e., $association_a_to_b$, we cannot refer to features of one of its instances directly. Instead, we create an iterator expression for the navigation edge, whose $source$ will be the actual context, e.g., $a.association_a_to_b$, and the iteration mode is set as specified by the navi-

gating association. By now, we have the following statement to identify the collection of objects and the iteration mode: $a.association_a_to_b < iterMode >$
As navigating associations reference the object type of the collection, concrete instances of the collection are addressed via the $instanceTransition$. Each instance, which is related with the same navigation node becomes a free variable in the context of an iterator expression. One iterator expression is created for each outgoing *AssociationTransition* $association_a_to_b$....$association_a_to_c$, whose upper bound is greater than 1. After processing all instance variables of one *NavigationNode*, e.g., $ClassB$, we get the following expressions, which still miss the body expression to detail the statements for a specific iterator variable: $a.association_a_to_b < iterMode > (b_1, b_2|)$.

The end of an *AssociationTransition* represents the new context, wherein instances can be addressed to define the body of the iterator expression, i.e., to define statements for the selected instance. Instance variables provide attribute and association features, which are handled as described above. In contrast, the context from which features are called has switched from the initial context a to the respective iterator variables b_1 and b_2 to create the body expressions $body_expression_b_1$ and $body_expression_b_2$ for each instance. The resulting body expressions are combined in the respective navigation node $a.association_a_to_b$ using the *OperationCall* expression and the Boolean operator <OR> to create the final set of body expressions
$C := a.association_a_to_b < iterMode > (b_1, b2|$
$body_expression_b_1$ <OR> $body_expression_b_2)$

This kind of OCL expression directly corresponds with navigation associations, which are typed with an iteration mode, such as $exists$, $forAll$, $collect$, $reject$, and $select$. For other modes, which refer to the semantics of navigation edges, such as $includes$, $inlucdesAll$, $isEmpty$, $notEmpty$, or $size$, slightly different strategies were applied.

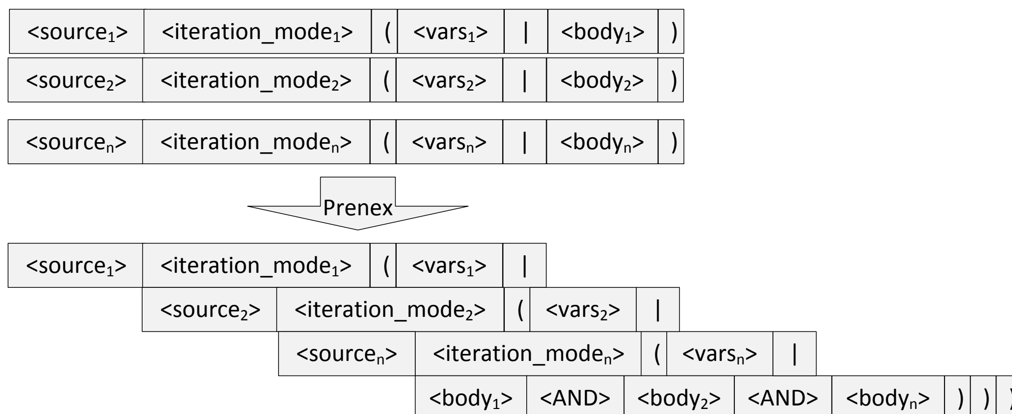


Figure 5.13: Conversion to prenex normal form

Multiple navigation transitions starting from the same instance node result in different iterator expressions. The resulting iterator expressions are combined by converting all iterator expression to prenex form, as exemplified in Figure 5.13. If we have n iterator expressions, all source- and variable-related parts are nested in one single hierarchic iterator expression, where we recursively define the body of an iterator expression as the subsequent iterator expression. In parallel, we gather all bodies of each single iterator expression and combine them with the Boolean operator <and>. The resulting *OperationCall* expression is used as body expression for the innermost nested iterator expression. The result is an expression D:=
$a.association_a_to_b < iterMode > (b_1, b_2|(\cdots)a.association_a_to_c < iterMode > (c|body_expression_b_1$ <and> $body_expression_b_c < and > body_expression_c))$

That way, all variables are bound before one of the body expressions is evaluated, so that each body expression refers to all relevant variables, which are needed for crossing attribute relations, as described in the following.

After we processed all properties, we combine all properties of the same instance node by using the Boolean operator <and> and get an *OperationCall* expression $E := A$ <and> B <and> D.

Crossing Attribute Relations

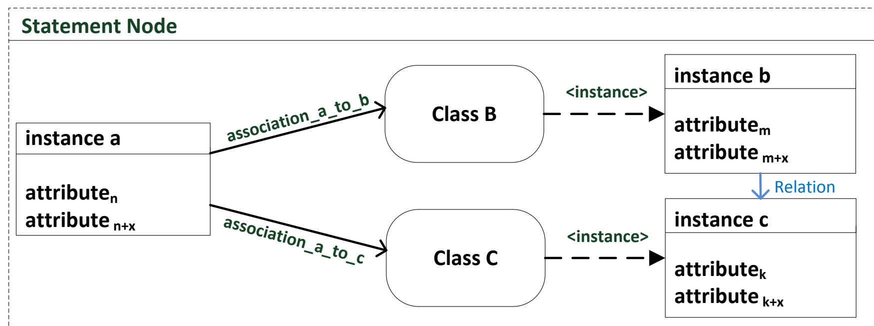


Figure 5.14: Crossing attribute relations

A crossing attribute relation, as depicted in Figure 5.14, is special to the handling of attribute properties. The blue arrow (Relation), which connects the instance of instance b with the instance c relates the two variables $attribute_m$ and $attribute_k$ by some comparing operation rel_{mk}, i.e., $attribute_m$ <rel_{mk}> $attribute_k$. It expresses a statement, whose first part, the source, depends on the context, which is determined by $association_a_to_b$ and a second part, the target, which depends on another navigation path $association_a_to_c$. During the transformation, either $association_a_to_b$ or $association_a_to_c$ are processed at first. In both cases, the context of the source instance node is different from the context of the referred instance node. However, as only the context of the source node is known, we cannot create an expression from the information provided with the target

node directly. Instead, after the source node and its known context is used to create the source expression to call $attribute_m$, the referred property $attribute_k$ must be called, as described in the following and related by an *OperationCall* expression using the respective operator.

As stated before, the prenex form, which is created to combine various iterator expressions, allows us to evaluate one encompassing body expression, for which all free variables are bound. As a result, we may assume that all instance nodes, i.e., variables, can be used for the *OperationCall* expression without modification.

In contrast, if the related target instance is not part of an iterator expression, we must derive its context by backtracking the guideline graph to a node, that represents the context for this node. We do this by iteratively tracing backward all ingoing navigation and instance transitions, until we reach the root node. Thereby, we use the instance variable as starting node to derive the context from all visited nodes and association properties in reversed order. This backtracking can be done deterministically, since all nodes must have at most one ingoing navigation or instance edge.

5.5.2 Guideline Application

Guideline application faces the process-oriented facet, as discussed in Section 4.6.1.2, to control the procedural and temporal interpretation of a guideline. The application must clarify the point in time when a guideline is evaluated and, in particular, how guidelines, which are composed of several statements, are interpreted to achieve an expected result. There are two strategies to initiate the evaluation of guidelines: During the execution of a method to validate each design step, or before/after the method starts/ends.
The first strategy has several drawbacks, as it provides an overhead of information all the time and hinders creative work through too much intervention from the framework. As the second strategy is less invasive and as the finalization of one method implies the start of a subsequent method, our approach is based on the guideline evaluation after a method is finalized, i.e., validation can be initiated by developers actively, after completing the task of a method.

After a developer has finalized a method, the set of guidelines, which is associated with the actual method, is used to ensure defined statements and the correct state of associated development artifacts. Therefore, all guidelines are interpreted, as discussed in the following.

In a nutshell, nodes and edges of the guideline, cf. Figure 4.11, are interpreted by using tokens similar to petri nets [Pet81]. The directed edges of a guideline transport tokens and evaluation results from a source node to a target node. Whether a node sends tokens to outgoing edges depends on the individual nodes' semantics and available tokens received from ingoing edges.
There are three types of nodes: statement nodes, action nodes, and control nodes. Each node type waits to receive one token from each of the ingoing edges. If a node has received all tokens, the node executes an associated task. After a node has completed the

task, the node sends exactly one token to each of its outgoing edge to trigger a subsequent node and to proceed the guideline workflow. As described in Section 4.6, only control nodes may have more than one ingoing or outgoing edge, i.e., statement and action nodes are activated as soon as a token arrives.

5.5.2.1 Statement Nodes

If a statement node receives a token, the contained statement is evaluated in the context of the actual method. First, a query is created to select relevant model elements from the underlying artifact model. As the statement defines a root element, all elements with the respective type of the root element are queried. For our purpose, we use EMF Query as framework to query the model for specific elements. Afterwards, the received set of objects may be further restricted to evaluate only elements, which are relevant to output artifacts of the actual processed method. This step is necessary, as individual model objects may belong to different artifacts even if they have the same type. In Section 5.4, we discussed, that this restriction is enabled through an observer mechanisms, which monitors modeling actions and assigns them with a respective artifact.

Afterwards, the restricted set of artifact-specific model elements is evaluated by the statement using an interpreter, which corresponds to the defined translational statement semantics, e.g., OCL. The result of the evaluation is a Boolean value or a set of objects, which fulfill the respective statement. For further processing and to combine the results with other results of other statements of the guideline, this result together with a token is passed to the subsequent node, to activate the next node whose execution depends on the previous result.

5.5.2.2 Action Nodes

If an action node receives a token, the associated behavior is started. As said earlier, possible actions are to provide messages to the developers, to provide situational access to supporting documentation, or to call program code to modify or rectify the artifact/model.

All actions depend on the received input. For Boolean values, a message informs developers if an artifact is in a particular state or not. For a set of objects, the message action can provide more detailed information about affected objects within the artifact.

A Boolean value reaching the action node may initiate some code fragments to modify the underlying artifact or not. For an ingoing set of objects, the snippet which is deposited for the action node may modify all contained objects.

In either case, i.e., Boolean or Collection-based inputs, an action which provides additional information in form of documents is called in exact the same manner.

After the action was executed a token together with the initial input parameter is send via the outgoing edge.

5.5.2.3 Control Nodes

As control nodes can have more than one ingoing or outgoing edge, their main task is to aggregate or combine parameters and to synchronize and control the workflow of a guideline. An initial node has no ingoing edges, thus, it provides an empty parameter set and starts the workflow interpretation by sending a start token to its predecessor node. In contrast, a final node has no outgoing edges and only waits for a token to indicate the end of interpretation.

The AND node has one ingoing edge and multiple outgoing edges. The node passes an ingoing token and the associated parameter through to all of its outgoing edges to activate multiple parallel paths. These parallel paths must be synchronized by using the JOIN node, which allows multiple ingoing edges, but only one outgoing edge. Thereby, the JOIN node aggregates ingoing parameters with the Boolean operator AND for Boolean values and with a union operation for multiple collections. The guideline designer has to ensure, that all ingoing results are compatible. Afterwards, the JOIN node sends one token together with the aggregated result.

OR and MERGE nodes are inverse to AND and JOIN nodes. An OR node allows one ingoing edge and multiple conditional outgoing edges. Therefore, the conditional attribute property of a *GLEdge* is relevant. For each *GLEdge* one of the binary values $SUCCESS$ or $FAILURE$ can be set. An OR node propagates an input parameter, which either is an ingoing non-empty collection or a $true$-valued parameter to all outgoing edges whose condition attribute is set to $SUCCESS$. For a parameter whose value is $false$ and for an empty collection, the OR node propagates all outgoing edges whose condition is $Failure$.

The MERGE node is a counterpart of the OR Node to synchronize the parallel sequences, which were initiated by an OR node. Therefore, a merge node has multiple ingoing edges and one outgoing edge to propagate a result, which is aggregated from input parameters. To aggregate ingoing parameters, a merge node provides an additional attribute. The min attribute defines the minimal number of ingoing edges, which must provide a token to activate the task of the MERGE node. By default its value is 1. However, if there is more than one ingoing parameter the OR node combines them by using the Boolean operation OR for Boolean values and the intersection operation for collections. Again, the guideline designer has to ensure that ingoing types are compatible.

5.5.3 Guideline Effects

During the guideline interpretation, all executed action nodes produce an entry for a validation report to inform developers about critical and questionable states and to provide an additional means to consider the effect-oriented facet, which we introduced in Section 4.6.1.3. The validation report is based on the meta model, which is depicted in figure 5.15. One general report is used during the development process to gather information, warnings, and critical errors, which were identified during the evaluation of a guideline. Whether a specific action node provides an information, a warning, or an error, depends

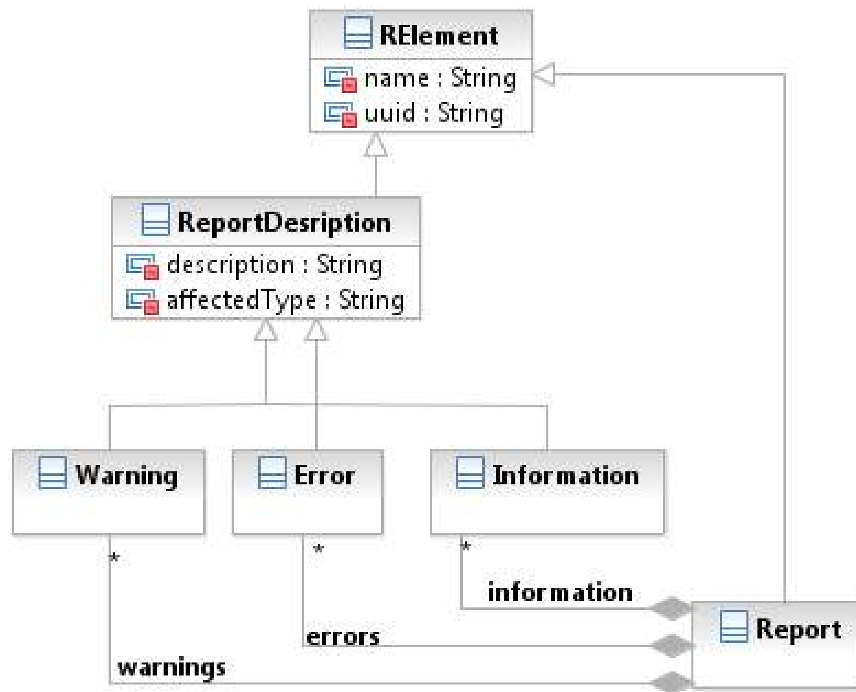


Figure 5.15: Validation Report

on the priority attribute, which is associated with each action node.

As a result, a detailed report is generated for developers at work to provide recommendations to enhance or even rectify a produced artifact. Furthermore, as the overall report is created and detailed continuously, information is evaluated to improve the process, after the project has been finished. That way, e.g., repeated design errors or other problems can be identified and avoided through selective trainings.

5.6 Engineering Process Coordination

Former sections detailed the technical design of process models and constituent MCs, as well as, the interpretation of MCs on operational level. However, to coordinate the process and to guide developers efficiently, an execution semantics must be defined in order to organize the runtime behavior of modeled processes. In [PGR99], they say, that "process coordination is not process control. Rather than a process being tightly or centrally controlled, process coordination allows individual agents to make good decisions and be notified of changes. Process coordination is more than simple workflow management, that determines to whom the document should be next routed. Process coordination is the runtime determination of who should be notified of what effect of the last change in the project." Therefore, general business processes are realized by a workflow management system, which determines the assignment of process activities or documents to

individuals or a group in the predefined order. The system uses, e.g., a directed graph of a process models, to follow the edges of that graph to get from one node representing an activity to another one. In either case, the system starts at an initial state and follows restricted set of alternative paths to reach a final state.
While this behavior perfectly meets the needs of general business processes, such as a flight reservation or a purchase order, this semantics is too restrictive for creative development processes. As the control-flow of development processes is influenced by strict dependencies between activities and the outcome of individual activities likewise, general execution semantics does not fit development processes. In particular, an execution semantics for a more flexible process management faces challenges, as discussed in the following.

Basically, we can not assume, that when software is developed all activities are performed at 100% in a first attempt. Such a material order, indeed, corresponds with most of today's process documentations, but it does not describe real chronological order of a performed process and its constituent parts. We name this the *deviation of material and chronlogical orders*. Generally, this results from the fact, that based on the current project situation, i.e., the status of individual artifacts, decisions, which influence the control-flow, must be made. The deviation mainly is influenced by the following development paradigms:

- *Iterative Development:* For development processes, it is difficult to decide automatically, whether or not a sub-workflow has to be repeated. At process design time, there are no criteria to express such conditions in general. Therefore, it must be possible to simply influence the workflow behavior at runtime, which induces a control-flow deviation.
- *Incremental Development:* Especially, software is developed incrementally to refine software and intermediate work products stepwise, i.e., work products are modified, which induces control-flow deviation.
- *Interactive Outcomes:* The produced output of a method influences other artifacts and methods. Especially, the modification of information, which is consumed at various phases of a development process, induces the interactivity of artifacts, i.e., the modification of an artifact influences the information content of related artifacts and induces control-flow deviation.

In summary, development processes must consider data, which change during the process, more than general business processes. Such information can not be modeled at process design time. Instead, in order to ensure the quality and consistency of artifacts as well, we must provide possibilities to monitor and to analyze the data processing at runtime.

To enable the flexible control of a development process, which overcomes the above challenges, we combine general control-flow semantics of a procedural PDL with a rule-based coordination mechanism. Therefore, we provide a mechanism, which analyzes monitored design information to cause a relevant set of predefined control-flow strategies. In our approach, different strategies are used to identify inconsistencies between

artifacts. Based on an identified inconsistency, a strategy influences the control-flow to fix inconsistencies, as discussed in the following. As a result, a strategy serves as a reaction rule, which influences the control-flow behavior to ensure consistency between artifacts. Additionally, other strategies or rules can be developed to face other coordination challenges.

5.6.1 Overview: Flexible Workflow Management

For enabling flexible workflow management, we developed the consistency production line, as illustrated in Figure 5.16. Therefore, we identified four sequential phases, which must be processed to extract relevant information required for controlling process behavior more flexibly. The four phases are called setup, monitor, evaluation, and control. During the setup phase, static data structures, which specify relevant relationships between process components, are prepared using static process model information. During the monitor phase, relevant runtime data are gathered and analyzed during the evaluation phase. Finally, analysis results and prepared dependency information are used to manage the process properly during the control phase. Before detailing the four activities, we explain the basic concepts, which enable the flexible management of process activities. We explain the difference between FPC and SPC and we explain the notion of a Rule-based Coordination.

5.6.1.1 Strict and Flexible Managed Components

In Section 4.3.3, the two generic concepts of SPCs and FPCs were introduced. While an SPC is restricted by its predefined control-flow, a control-flow connecting various FPCs has a recommending character. The recommending control-flow is a best case scenario, which can be controlled based on an additional set of rules and perceived process knowledge. The following example will demonstrate the difference between that two paradigms: In general software engineering, *RE*, *Analysis*, *Design* and *Implementation* define a sequence of four abstract process periods. Since we defined an FPC as a long-lasting

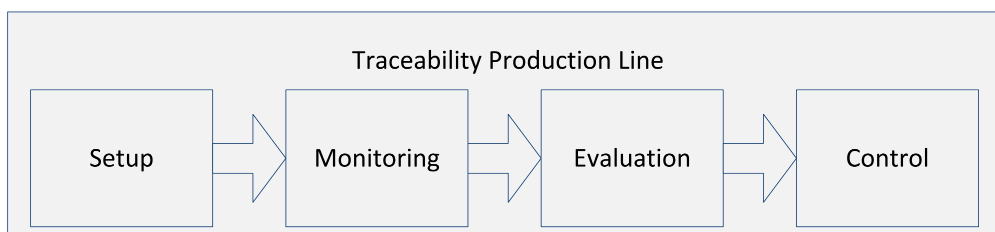


Figure 5.16: Consistency Production Line

process period, whose execution depends on e.g., actual project situations, iterative/incremental rework, unplanned and planned change requests, concurrent development or other external and random events, the four periods perfectly match with the semantics of an FPC. Now, we assume, that the *Analysis* period consists of an activity, which consists of a distinct set of MCs to construct an analysis architecture. We further assume, that the *Design* period consists of another activity to refine the existing analysis architecture with additional information in order to construct a logical architecture. Additionally, both activities, which are only parts of the respective *Analysis* and *Design* periods, are defined as an SPC. The SPC semantics effects, that both activities, i.e., contained MCs, are performed and finalized following the defined control-flow in strict order and it can not be interrupted through potential events. However, after an activity is finalized, flexible workflow management takes control to decide about further proceeding depending on the current project situation. For example, if no indicators can be identified after the logical architecture was created, the workflow management will continue the process following the defined default control flow. In contrast, if conflicts would be identified between logical architecture developed during the *Design* period and the analysis architecture developed during the *Analysis* period, the workflow management is allowed to execute an FPC on demand. In our example, the workflow management would re-start the *Analysis* period, if elements of analysis architecture are in conflict with the logical architecture, i.e., the validity of the analysis architecture can be ensured on demand. This simple example demonstrates, that FPCs enable us to start other FPCs as induced by the current process situation and underlying rules.

5.6.1.2 Rule-based Coordination Mechanism

While common control-flows follow a predefined sequence of activities in a strict order, FPC nodes and contained SPCs can be started independently from a predefined order and based on situational needs. To decide about the necessary deviation from a default control-flow between FPCs as defined by a process model, each time an FPC is finalized, we check specific criteria and start affected FPCs. The following will introduce the process of managing the FPC control-flow and criteria, which can be used to analyze the impact of activities, which were performed during an FPC, on artifacts of other FPCs. However, to analyze the impact between artifacts of different or the same FPCs, we first require a specialized notion of traceability.

In general, the IEEE Standard Glossary of Software Engineering Terminology [{IEEE Computer Society}90] defines traceability as “the degree to which each element in a software development product establishes its reason for existing.” A very similar definition can be found in [GF94], where traceability is defined as “the ability to describe and follow the life of a requirement, in both a forward and backward direction; i.e., from its origins, through its development and specification, to its subsequent deployment and use, and through periods of ongoing refinement and iteration in any of these phases.”

Although, mentioned definitions require the traceability over all phases of the development life-cycle, traceability still lacks support for development activities, which are

different from requirements engineering, i.e., traceability of requirements [SR09]. In this field, well known traceability relationships with a particular semantics, such as *Refine*, *Satisfy*, *Verify*, *Realization*, *DerivedRequirement*, and *Derived* are set between design elements manually. However, in [TZD09], they identified the following factors, that complicate the establishing and maintenance of trace dependency links, in general:

- There are no explicit links between process design and implementation languages. This lack of dependency links is caused by not only syntactic and semantic differences, but also the difference of granularity as these languages describe a process at various levels of abstraction.
- A substantial complexity is caused by tangled process concerns. Either the process design or implementation comprises numerous tangled concerns, such as the control, data processing, service invocations, transactions, fault or event handling. As the number of services or processes involved in a business process grows, the complexity of developing and maintaining the business processes increases along with the number of invocations, data exchanges, and cross-concern references, and therefore, multiplies the difficulty of analyzing and understanding the trace dependencies.
- There is a lack of adequate tool support to create and maintain trace dependencies between process designs and implementations.

To overcome these challenges, and in contrast to other approaches, which focus on artifacts and manually defined links between individual elements, we focus on the establishment of an automated traceability between elements, which are contained in artifacts. Therefore, for our purposes, we define traceability as follows:

Definition 31 (*Traceability*)

> Traceability is a capability to link and follow artifacts based on contained elements. With respect to artifacts of a development process, traceability must establish a backward and a forward link between artifact elements and their changes during the development.

Based on this definition of traceability, we developed a sophisticated traceability mechanism, which enables a detailed change impact analysis on artifacts across the overall process. Traceability is based on the set of artifact histories, which were introduced in Section 5.4. We follow the history of model type instances, which are spread over various artifacts, and identify potential conflicts. Based on the characteristic of a specific conflict, we enable the automated derivation of actions to manage the control flow in order to solve or validate the potential conflict. This is, what we call Rule-based Coordination.

For example, when an element x is produced in the context of an artifact A and x is modified in the context of an artifact B, the element x belongs to two artifacts A and B. We can derive, that the modification of B influences A and dependent activities. Based on such information and the identification of such impacts, general rules are defined to decide about the automated or controlled initiation of corresponding FPCs, which are influenced by B and A. The case of a controlled initiation means, that impact analyses

must not be seen as obligatory for performing an action. Instead, they can provide some hints for potential conflicts and the final decision about what to do can be dedicated to the developer.

The following describes the four phases of the consistency production line, as illustrated in Figure 5.16.

5.6.2 Consistency Check: Setup Phase

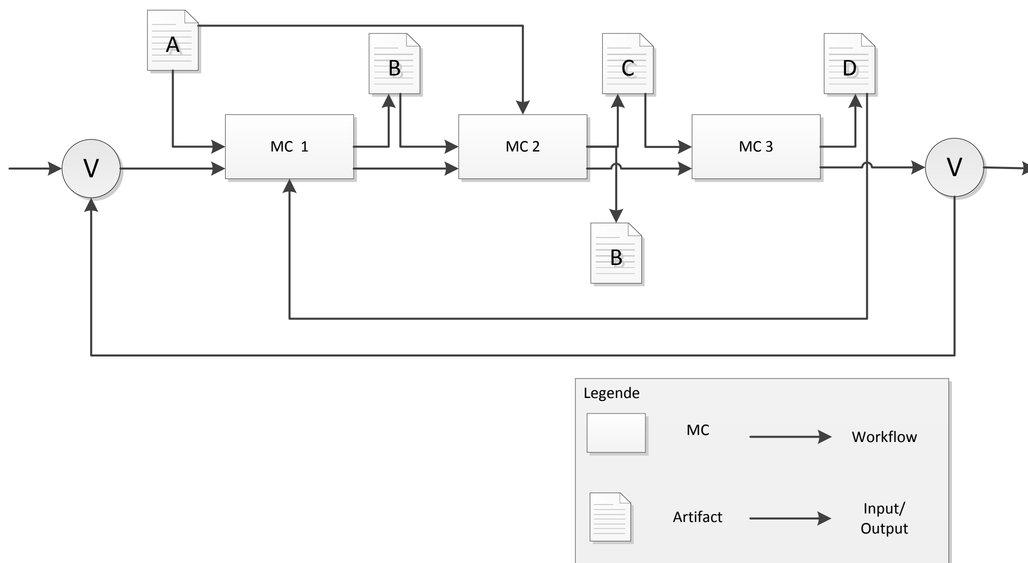


Figure 5.17: Simple Workflow Example

To avoid long-lasting operations at runtime, we prepare a distinct data structure from the process model. The data structure serves as a traceability look-up table and provides us with information about the influence of individual artifacts on MCs spread across the process.

The data structure, which is prepared before runtime, is called *Artifact Influence Table*. For each artifact, this table assigns a list of MCs, whose execution is influenced by the respective artifact. The influence table allows to identify the tasks, which are influenced (or must be validated), when a specific artifact has changed. Listing 5.6 shows the algorithm to set up the *Artifact Influence Table*. To identify the dependencies between MCs and associated artifacts, we use backwards traversal to reach all nodes of a process model. A node is handled, as illustrated in listing 5.7. An initial node of the process provides us with a termination criteria and control nodes can be ignored by handling only its predecessor nodes. For MCs, which specify input artifacts, an input artifact influences the execution of the MC and we put the MC in an artifact-specific list of dependent MCs. For example, based on the process model, which is depicted in Figure 5.17, Table 5.3 exemplifies a corresponding influence table. For each artifact, the table shows a list of influenced MCs.

```
Hashtable<Artifact, List<MC>> influenceTable :=
  new Hashtable<Artifact, List<MC>>();

void createInfluenceTable(ProcessModel process){
      //backwards traversal
      List<FinalNode> finalNodes = process.getFinalNodes();
      if(!finalNodes.isEmpty()){
        return null;
      }
      for(FinalNode finalNode : finalNode){
        traverseNode(finalNode);
      }
}
```

Listing 5.6: Artifact Influence Table

```
private void traverseNode(Node node){
      //handle all predecessor nodes of the actual parameter node
      for(Edge edge : node.getInEdges()){
        Node preNode = edge.getSource();
        //end condition
        if(preNode instanceof InitialNode){
          break;
        }
        //normal case - relate outputs with inputs of an MC
        if(preNode instanceof MC){
          MC actualMethodChunk = (MC) preNode;
          //Insert the influenced MCs in the hashtable
          for(Data artifact : actualMethodChunk.getinputs()){
            List<MC> mcs = influenceTable.get(artifact);
            influenceTable.put(artifact, mcs.add(actualMethodChunk));
          }
          //handle next node
                for(Edge preNodeEdge : preNode.getIn()){
                  traverseNode(preNodeEdge.getSource());
                }
        }
      }
      //Control node case - handle predecessor nodes
      if(preNode instanceof ControlNode){
        for(Edge preNodeEdge : preNode.getIn()){
          traverseNode(preNodeEdge.getSource());
        }
    }
  }
}
```

Listing 5.7: Handle node for the influence table creation

After the artifact influence table is prepared, a corresponding workflow can be started and modeling events are monitored, as discussed in the following.

Influence	*Task List*
Artifact A	MC1; MC2
Artifact B	MC2
Artifact C	MC3
Artifact D	MC1

Table 5.3: Example: Artifact Influence Table

5.6.3 Consistency Check: Monitoring Phase

After all static information are set up and prepared to be available at runtime, the monitoring and logging of modeling events can be started. Since this was already discussed in detail in Section 5.4, we here revisit the basic idea shortly: We log modeling actions for each data element in the context of an MC and associate the monitored event with a time stamp. This is required to order the set of events afterwards. Subsequently, we assign virtually each event with an affected artifact to establish an artifact history. The monitored artifact history is analyzed in the next phase to trace the artifact evolution and its influence on other artifacts or individual MCs.

5.6.4 Consistency Check: Evaluation Phase

After modeling events were monitored, an FPC serves us as checkpoint for the evaluation of monitored information and a subsequent decision making to influence the control-flow of a process by starting relevant FPCs. Therefore, we developed a set of strategies to identify potential consistency conflicts between artifacts. A strategy analyzes a particular set of modeling events, which describe a time-based constellation of modeling events spread across various artifacts and/or MCs. A modeling event is composed of the information, as introduced in Section 5.4.1. Therefore, we extract relevant information from the notifications and define an event as an 5-tuple, which is composed of a specific MC MC_n, an output artifact $artifact_a$ of MC_n, a time-stamp t of the corresponding modeling action, the model element instance $inst$, which was influenced by the modeling action, and the type of the modeling action. The following discusses the identified strategies in detail:

Conflict Identification Strategy 1

$$\left.\begin{array}{l} Event(MC_n, artifact_a, t, inst, create \vee modified) \wedge \\ Event(MC_n, artifact_a, t+x, inst, modified \vee delete) \end{array}\right\} \implies \texttt{no conflicts}$$

In strategy 1, one element ($inst$) of an $artifact_a$ is created, modified, and/or deleted during one working step, i.e., MC_n several times. Since multiple modification of one element has no more influence, than an unique modification, this scenario causes no conflicts.

Conflict Identification Strategy 2

$$\left.\begin{array}{l} Event(MC_n, artifact_a, t, inst, create \vee modified) \wedge \\ Event(MC_{n+y}, artifact_a, t+x, inst, modified \vee delete) \end{array}\right\} \implies \texttt{potential conflicts depending on } artifact_a$$

In strategy 2, a model element ($inst$) is created or modified in the context of an $artifact_a$ during MC_n, and the same element of the $artifact_a$ is modified or deleted, in a second different working step (MC_{n+y}). Due to the modification of $artifact_a$ in the context of two different MCs, other MCs could use the modified content of $artifact_a$ between t and $t+x$ inconsistently, i.e., after modifying the artifact in MC_{n+y}, other MCs could base their design decisions on inconsistent information from MC_n. This is a hint for a potential conflict, which causes the repetition of MC_n.

Conflict Identification Strategy 3

$$\left.\begin{array}{l} Event(MC_n, artifact_a, t, inst, create \vee modified) \wedge \\ Event(MC_{n+y}, artifact_b, t+x, inst, modified \vee delete) \end{array}\right\} \implies \texttt{potential conflicts depending on } artifact_a$$

In strategy 3, one model element is created or modified in the context of an $artifact_a$, and the same model element is modified or deleted in another artifact $artifact_b$ during a second different working steps, i.e., both artifacts are influenced. Due to the modification of model element $inst$ in the context of two different MCs, other MCs could use the modified content of $artifact_a$ between t and $t+x$. After modifying the $artifact_b$ in MC_{n+y}, other MCs based their design decisions on inconsistent information of $artifact_a$. This is a hint for a potential conflict, which causes the repetition of MC_n.

Conflict Identification Strategy 4

$$\left.\begin{array}{l} Event(MC, artifact, t, inst, create \vee modified) \wedge \\ Event(MC, artifact, t+x, inst, read) \end{array}\right\} \implies \texttt{no conflicts}$$

In strategy 4, a model element is created or modified, before it is read in the context of another MC. Since a simple reading of available information does not influence other artifacts or the execution of MCs, no conflicts can arise.

Conflict Identification Strategy 5

$$\left.\begin{array}{l} Event(MC_n, artifact_a, t, inst, read) \wedge \\ Event(MC_n, artifact_a, t+x, inst, modified \vee delete) \end{array}\right\} \implies \texttt{no conflicts}$$

In strategy 5, a model element of an artifact is read and subsequently modified or deleted in the same task and artifact. Since all modifications of an element in the context of the same MC do not influence itself, no conflicts are caused.

Conflict Identification Strategy 6

$$\left.\begin{array}{l} Event(MC_n, artifact_a, t, inst, read)\wedge \\ Event(MC_{n+y}, artifact_a, t+x, inst, modified \vee delete) \end{array}\right\} \implies \texttt{potential conflicts depending on } artifact_a$$

In scenario 6, a model element of an artifact is read and subsequently modified or deleted in the context of another MC. Since the first MC's outcome is based on modified data, this situation causes a potential conflict in $artifact_a$.

Conflict Identification Strategy 7

$$\left.\begin{array}{l} Event(MC_n, artifact_a, t, inst, read)\wedge \\ Event(MC_{n+y}, artifact_b, t+x, inst, modified \vee delete) \end{array}\right\} \implies \texttt{potential conflicts depending on } artifact_a$$

In strategy 7, a model element of an artifact is read and subsequently modified or deleted in the context of another MC and another artifact. Since the first MC's outcome is based on modified data, this situation causes a potential conflict concerning $artifact_a$.

Conflict Identification Strategy 8

$$\left.\begin{array}{l} Event(MC_n, artifact_a, t, inst, create)\wedge \\ Event(MC_n, artifact_b, t, inst, create)\wedge \\ artifact_a \veebar artifact_b \in inputsMC_c \end{array}\right\} \implies \texttt{potential conflicts depending on } artifact_a$$

The last strategy 8, faces a situation, when a set of similar typed model elements, which are already associated with an artifact, is extended in the context of another artifact. For example, if an $artifact_a$ inputs model elements of type X to an MC, e.g., MC_n, to refine $artifact_a$ by additional elements of type X in a new $artifact_b$. Hereby, the adding of an element of type X to the output $artifact_b$ also influences $artifact_a$, since elements added to $artifact_b$ belong to both artifacts, at the same time. This is indicated by a corresponding *FromTo* relationship between the respective input and output artifacts, which causes the relevant interest registration for artifacts in the context of our observer mechanism. Both artifacts, i.e., $artifact_a$ and $artifact_b$, would be notified about the creation of the same element. However, while the produced output only does influence subsequent MCs, the change in the consumed $artifact_a$ may influence MCs, which use $artifact_a$ as input in parallel with MC_n. This indicates a potential conflict, caused by $artifact_a$.

Contrasting an analysis of static artifact dependencies, we make use of predefined MMVs and monitored modeling actions. During the change impact analysis, the above rules consider the specific elements of the artifacts. Therefore, dependent artifacts, which are not affected by a modification of specific element are neglected for change impact analysis. Through the knowing of influenced MCs, only relevant MCs can be triggered purposefully, to enable the validation of artifacts, which potentially are in conflict or inconsistent with another artifact.

To reason about monitored information in order to coordinate development tasks is a critical task. Based on the reasoning, the evaluation result identifies possible conflicting artifacts associated with particular MCs of a process. As a result, process entry points are identified to resolve the conflict, and to guide developers in doing the required tasks to validate or rectify the potential inconsistent outcomes. This is described in the following.

5.6.5 Consistency Check: Control Phase

The control phase uses the prepared artifact influence table and analyzed runtime information to control FPCs as required for the current project situation. Therefore, all conflicting artifacts, which were identified during the evaluation phase are looked up in the artifact influence table. Querying the table results in a set of MCs, whose execution is influenced by these artifacts, i.e., the artifacts are the input artifacts of respective MCs.

The resulting set of MCs is the set of activities, which must be performed in order to ensure, that a conflicting artifact does not have negative side effects on other outcomes. However, since our approach of flexible process management is based on FPCs, i.e., a MC can not be executed beside the predefined control flow, for each influenced MC, the encompassing FPC has to be identified. Since FPCs are container nodes to structure MC and other FPCs hierarchically, this can be achieved by using the transitive parent relationship between MC, SPCs and respective FPCs.

After all FPCs were identified, they are put into an execution queue, which manages the normal flow of FPCs and identified conflict resolving FPCs in parallel. The management of FPCs, can be realized by a priority based queue, which initiates relevant FPCs and contained SPC workflows using a general workflow engine.

5.7 Case Study

Using the case-study from Section 4.8, we show how the process model and technical MF information enable the application of CME on operational level. Therefore, the MMVs, which we introduced in Section 4.4 in combination with the artifact observer mechanism (cf. Section 5.4.1) enables the *identification of an artifact* and its contained information. Furthermore, the MMVs, which are associated with the input and output artifacts of an MC, in combination with the annotated EUSA information enable the generation of *method-specific CASE support*. Likewise, the guidelines, which were specified for a particular MC, enable the application and evaluation of *method-specific constraints*, while the workflow, appropriate control nodes, and the semantics of strict and flexible managed process components enable the *automated assignment of activities* to individual roles in the correct order. Additionally, monitored information and known relationships, which are given by the process model, are used for further enhancements, such as *consistency management*, *traceability* between artifacts, and the application of *process mining* techniques. All this is demonstrated realizing the design example of our case-study, as described in Section 4.8.

5.7.1 Guidance Preparation

To set up our execution environment, i.e., the step before a process is executed, four general tasks are performed to generate operational artifacts from the process model:

1. method-specific editors are generated using the MMV information in combination with annotated EUSA values.
2. the interests of artifacts are registered at the observer automatically. Similar to the editor generation, this task uses MMV and EUSA information.
3. the graphical guideline models are translated into an operational format using the translational semantics.
4. the artifact consistency management is prepared by generating the artifact influence table.

5.7.1.1 Editor Generation

The input and output artifacts of an MC on the technical level were extended by MMVs to define their internal data structures. As these data structures define the relevant vocabulary, which has to be provided with an editor to support the execution of a development activity, we use this information to generate the editor part automatically, as discussed in Section 5.3. Some of the generated editors, which were derived from the process model automatically to support individual task, are illustrated in the following.

For example, in Figure 5.18 one can see the generated editor, which supports the analysis phase of our case-study process and, in particular, the MC *CreateNonFunctionalRequirements*. The editor was generated based on the MMV, which is associated with the input artifact *Functional_Requirements* (cf. Figure 4.25), and the MMV, which is associated with the output artifact *NonFunctional_Requirements*. As one can see on the left side of Figure 5.18, the partition between input and output information is realized in the generated editor. In the section *A* of the illustrated editor, the root element of the input's MMV is depicted. Since MMVs, in general, are represented hierarchically, this perfectly matches the master-detail pattern, which we applied for the generation of editors (cf. Section 5.3). In section *B* of the editor, the output artifact is represented. The root element of the artifact is of type *System*, whose instance element *TheSystem* is selected as actual master element. For this element, its details are depicted in section *C* of the editor. This section is split into two sections: the attribute section to represent relevant attribute features, and the containments section *D* to represent relevant objects, which are part of a containment association of the master object, i.e., *TheSystem*. While the original meta model (cf. Figure 4.22) requires a *System* element to have an identifier attribute and a name attribute, the generated editor neglects the identifier attribute, as specified in the MMV of the MC's output artifact. Additionally, since the name attribute was associated with the EUSA read-only, the generated editor also does not permit to change the value of that name attribute during this process step. Instead, the editor provides an exclusive context menu in section *F*, which enables to add (and remove) elements to (from) the list

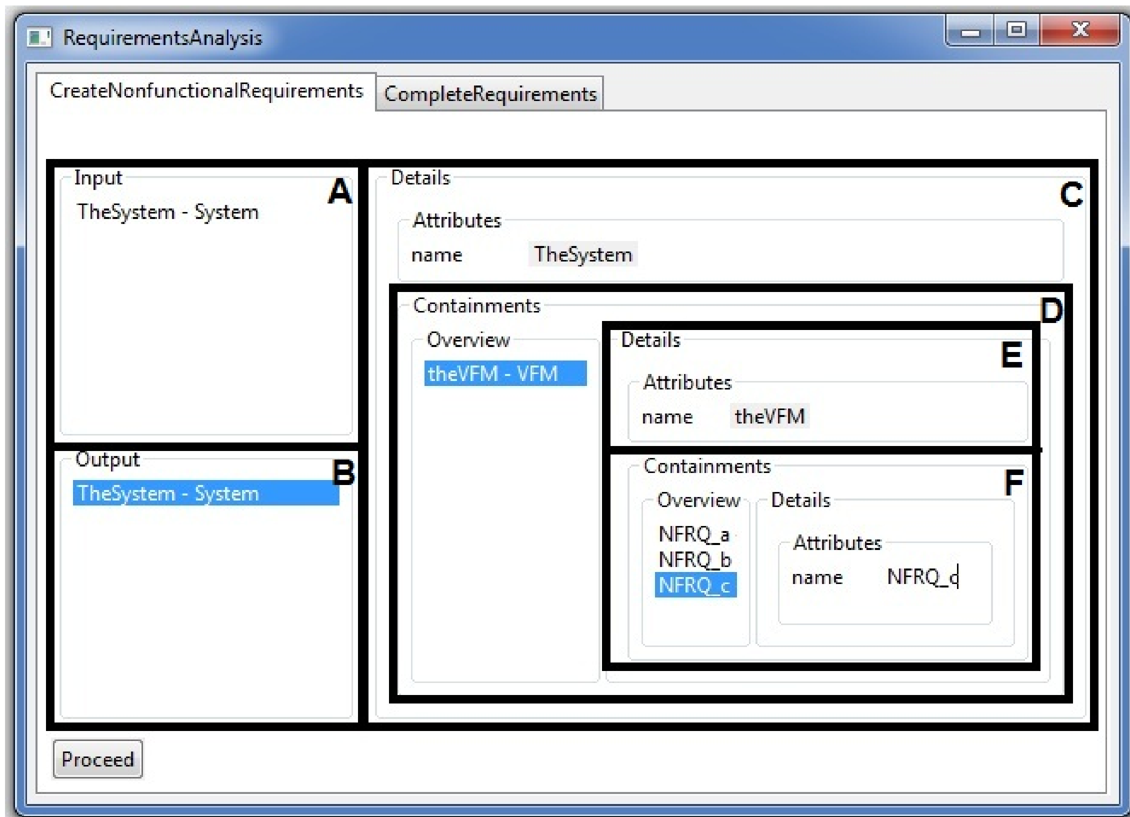


Figure 5.18: Generated Editor to support the MC: CreateNonFunctionalRequirements

of non-functional requirements, which are contained in *theVFM*.
Next, while the original meta model specifies a *System* element to have the three containments *vfm*, *faa*, and *impl*, the editor neglects elements, which are not specified as part of the respective MMV. Therefore, the only element, which is represented as contained by the *System* is *theVFM*, which is typed by VFM. Due to the selection of the element *theVFM*, it becomes a master for a subsequent details section (*E*/*F*). *E* is a Basic Structural Element, that represents the *VFM*'s name attribute feature as read-only. Within the respective containment section *F*, only elements typed by *NonFunctional_RQ* are visualized to enable the modification of their *name* attribute value, as specified in the MMV.

The same principles are realized in the editor, which is depicted in Figure 5.19. The shown editor supports the MC *CompleteRequirements* and produces a combined and validated view on functional and non-functional requirements as its output. Therefore, in the details view of the *VFM* (section *G*), functional (FRQ_) and non-functional (NFRQ_) requirements are represented.

Another example for an editor, which results from the case-study process, is illustrated in Figure 5.20. It demonstrates the treatment of references. On the one side, one can see, that the shown label of a *Connection* element is the identifier attribute in contrast to its name attribute (section *H*). On the other side, as referenced *InPort*s and *OutPort*s are contained by a software component (*SWC*), the *source* and the *target* reference must be added from the set of already available *Port* elements. Therefore, the editor extracts referenceable elements from the model and allows to set/unset the reference feature value via the add/remove button (section *I*).

5.7.1.2 Artifact Interest Indication

The generated editors use the dispatcher interface, as introduced in Section 5.4.1, to notify interested artifacts about performed modeling activities. To register their interests, artifacts use strategies, as introduced in Section 5.4.2. For example, the output artifact of the MC *CreateFunctionalRequirements* registers its interest according to the EUSA annotations, as illustrated in Table 4.2: As the concept of a *Functional_RQ* is annotated with the EUSA attribute *create*, the artifact registers an interest for the creation and modification of such elements in the context of that task.
A more complex example for the registration of interests is given in the context of the *CompleteRequirements* MC. The output of that MC, i.e., its MMV and associated EUSA values, is summarized in Table 4.3. Here, the interests of the artifact *combined_requirements* are registered according to the EUSA attribution, as described above. However, since we defined a *FromTo* relationship between the MMV concepts functional and non-functional requirements of that artifact and individual input artifacts (cf. Section 4.8.1), related input artifacts are required to register their interests correspondingly. Therefore, before the MC *CompleteRequirements* is performed, the input artifact *Functional_Requirements* registers an interest for creation events concerning instances typed with *Functional_RQ*, and the input artifact *NonFunctional_Requirements* registers an interest for creation events concerning instances typed with *NonFunctional_RQ*. That way, not only the output artifact is

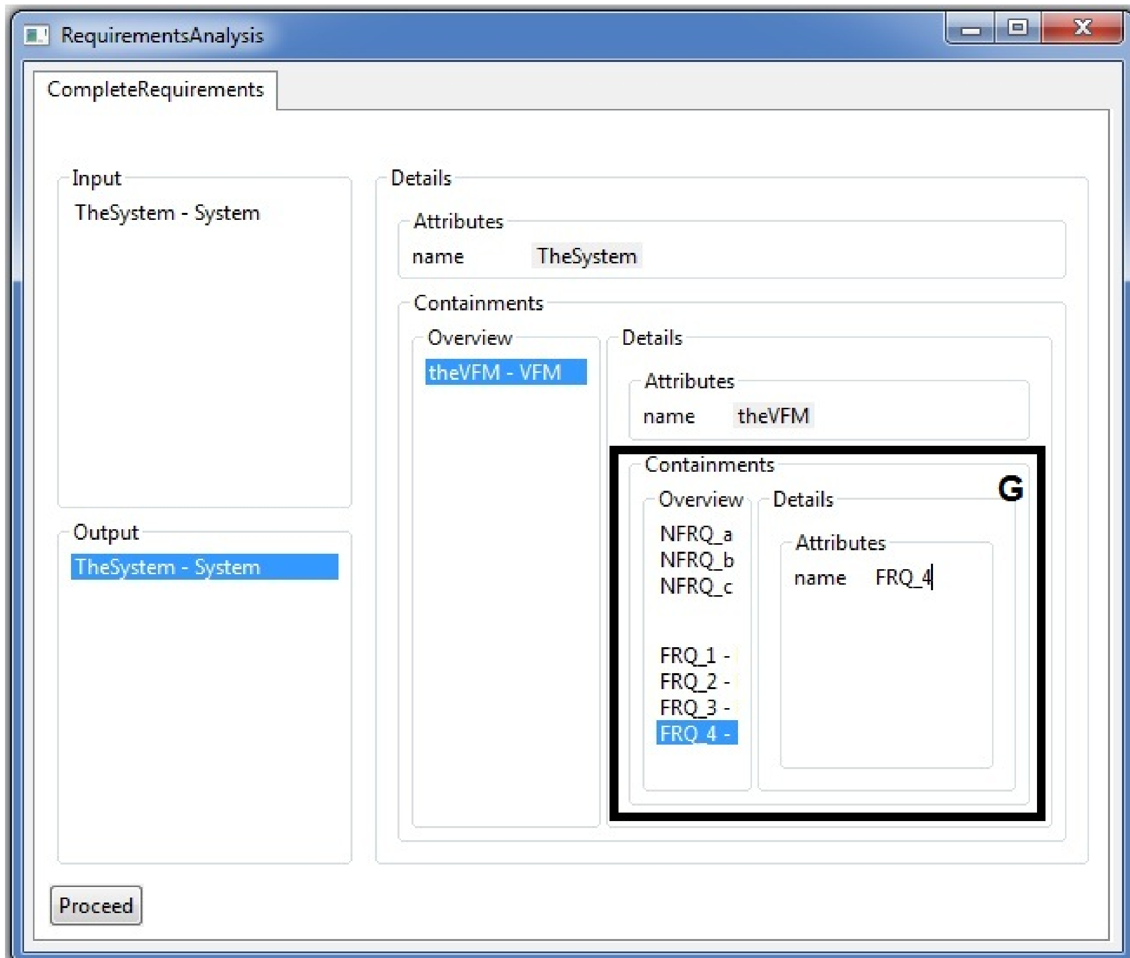


Figure 5.19: Generated Editor to support the MC: Complete Requirements

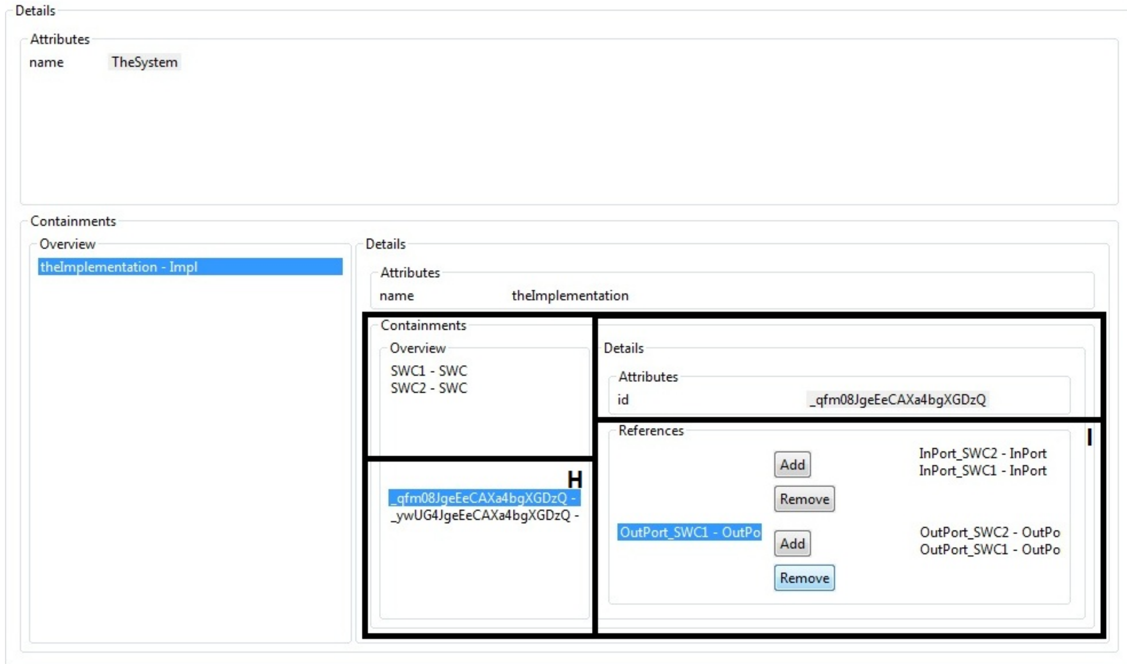


Figure 5.20: Generated Editor to support the MC: Define SW Architecture

influenced by the creation of new elements, but also the respective input artifact, which initially provides the MC with corresponding input elements.

5.7.1.3 Guideline Translation

In parallel, for all the modeled guidelines, the statement nodes are prepared to be evaluated in the context of a specific MC. Therefore, each MC is queried for guidelines to transform contained statements into an interpretable format, as discussed in Section 5.5. For example, the statement nodes which are illustrated in Figure 4.29, are transformed into the following OCL statements according to our translational semantics. Hereby, the structures of both statements look very similar: first, an instance of the meta model type *FunctionalReq*, which is the context element of the statement, is identified via the *instance-Transition*. This is translated into the following OCL sub-statement: *self.oclAsType(aMetaModel::FunctionalReq)*. The defined *AttributeProperty*, subsequently, is used to identify the relevant attribute feature of the respective instance object, i.e., the *name* of a *FunctionalReq*. By the means of OCL's *PropertyCallExp* expression, both parts are concatenated and represented using OCL's textual syntax as follows: *self.oclAsType(aMetaModel::FunctionalReq).name*. Finally, the defined *operation* value of the *AttributeProperty* specifies the concrete constraint for a functional requirement in this scenario. While for the upper statement node in Figure 4.29 the *operation* is *UNEQUAL* and the value is *null*, for the lower statement, the operation is defined as *regexMatch* and the value is defined as the following regular expression: $'(FRQ_)[a-zA-Z0-9]+'$. Using OCL's *OperationCall* expression, we can set its parameter using the information about *operation* and *value* and concatenate the former *PropertyCall* expression with the *OperationCall* expression. That way, the illustrated statement nodes result in the following two

OCL statements:

```
self.oclAsType(aMetaModel::FunctionalReq).name <> (null)
```

Listing 5.8: OCL statement: attribute operation

```
self.oclAsType(aMetaModel::FunctionalReq).name
.regexMatch('^(FRQ_)[a-zA-Z0-9]+') <>(null))
```

Listing 5.9: OCL statement: regular expression

To exemplify the derivation of a more complex OCL statement from our graphical syntax, we use the example, as illustrated in Figure 4.30. The statement expresses, that the number of functional requirements of an VFM must be equal to the number of functions defined in the functional analysis architecture (FAA). The root element of this statement is an element of type *System*, where we identify an instance using the *instanceTransition* first. To identify the systems's *VFM*, which is the container element for all types of requirements, the *AssociationTransition* was applied to navigate the meta model. Afterwards, the *instanceTransition* is applied for a second time to address a specific instance of the *VFM*, which is contained in the *System*, as identified before. Using OCL's *CallExp* expression, this results in the following OCL statement: *self.vfm*. This works similar for the identification of instance of an *FAA*, which results in the OCL statement: *self.faa*
In the context of an *VFM* instance, the graphical statement uses an *AssociationTransition*s to address functional requirements, while another *AssociationTransition* is used to address the functions of an *FAA*. As we aim at the comparison of the number instances of these two sets, the *iterationType* attribute of both *AssociationTransition*s is set to *size*, which implicitly is related with the *select* operator of an iterator expression in the context of sets, as considered in this scenario. Next, to address individual instances of functional requirements and functions, an *instanceTransition* is applied in either case. Based on that, each of the split paths results in one *IterateExpression*, which results in the following two OCL statements:

```
self.vfm.requirements->
select(l10|oclIsType(aMetaModel::FunctionalReq))->size()
```

Listing 5.10: OCL statement: iterator expression 1a

```
self.faa.functions->
select(l10|oclIsType(aMetaModel::Function))->size()
```

Listing 5.11: OCL statement: iterator expression 1b

Now, to compare the two sets, i.e., the number of contained instances, the *InstanceRelation*, which relates the two instance variables representing the members of respective sets, is translated accordingly. Therefore, the *InstanceRelation* is translated into an *OperationCall* expression, where the operator is set according to the *InstanceRelation* operation attribute *EQUAL*. This results in an OCL expression, which corresponds to OCL's textual syntax as follows:

```
self.vfm.requirements->
select(q1|oclIsType(aMetaModel::FunctionalReq))->size()
```

```
3 ==
4 self.faa.functions->
5 select(r1|oclIsType(aMetaModel::Function))->size()
```

Listing 5.12: OCL statement: operation call expression

Similar to the above translations, other statements are transformed into platform-specific code to be interpretable at runtime and to provide validation results for individual guidelines of performed MCs.

5.7.1.4 Traceability Setup

Finally, in order to trace dependencies between artifacts and to identify potential impact between artifacts, the artifact influence table is prepared according to the algorithm introduced in Section 5.6.2. Based on our process (cf. Figure 4.23), for each artifact the algorithm provides a list of influenced MCs, as illustrated in Table 5.4. In particular, note, that the artifact *Functional_Requirements* influences the two MCs: *CreateNonFunctionalRequirements* and *CompleteRequirements*.

5.7.2 Guidenace Application

After the environment is prepared through the creation of editors with method specific editor capabilities, the registration of artifacts interests, the generation of interpretable guidelines, and the creation of tracing dependencies, the process is ready to be executed. To demonstrate the behavior and the guidance capabilities at process runtime, we follow the sequence of MCs, which is defined in our process, and describe a random sequence of modeling events. Using the example of the analysis phase of our case-study process, we demonstrate the essential guidance capabilities, i.e., method-specific editing capabilities, artifact observation, guideline evaluation, and workflow coordination.

Influence	*Task List*
external_requirements	CreateFunctionalRequirements
Functional_Requirements	CreateNonFunctionalRequirements CompleteRequirements
NonFunctional_Requirements	CompleteRequirements
combined_Requirements	DefineFunctions
Functionality_Design	ConnectFunctions
Functional_Architecture	DefineFunctionTiming
FA_refined_by_timing	DeriveSWCs
SWC_description	DefineSWArchitecture
SW_Architecture	-

Table 5.4: Case Study Influence Table

Figure 5.21 to Figure 5.24 exemplify the sequential execution of the analysis phase according to the process, as defined in Figure 4.23. The editor depicted in Figure 5.21 supports the first MC in our process, i.e., *CreateFunctionalRequirements*. According to our process definition, this MC supports the creation of functional requirements, why the only represented model elements are the functional requirements contained in an *VFM* container element of the meta model's root elements *System*. As we defined the EUSA value *create* for functional requirements, the editor capability is manifested in a context menu of a list viewer, which illustrates available functional requirements. Due to the EUSA value *create*, a developer is allowed to *create* new elements and to *delete* existing elements, which are typed with *Functional_RQ*. Note, while the meta model specification provides functional and non-functional requirements to be contained in *VFM*, the editor only provides relevant information by neglecting non-functional requirements in this view.

Next, we create a functional requirement, using the context menu, as depicted in Figure 5.21.

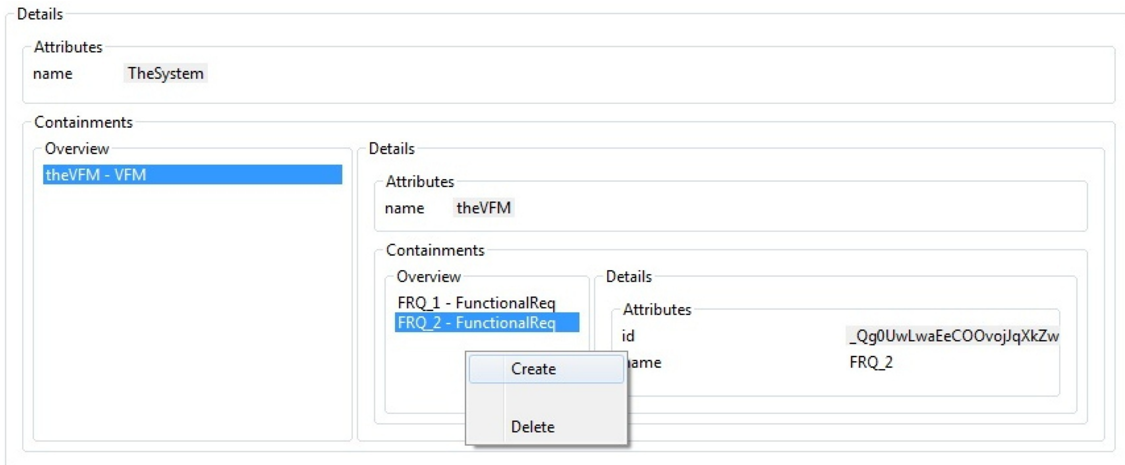


Figure 5.21: Execution of "Create Functional Requirements"

Subsequently, an instance of *Functional_RQ* is created and added to list of existing functional requirements, as depicted in Figure 5.22. As we defined the EUSA value *modify* for a functional requirement's name feature, we are allowed to modify the name of the just created element, while it is forbidden to modify the shown identifier attribute feature. In this scenario, we want to change the name of the just created functional requirement to "aFunctionalReq". Subsequently, we finalize this MC to perform a following design activity. However, each time we want to proceed from one MC to another, a validation is performed. As described in Section 4.8.3, the MC *CreateFunctionalRequirements* is associated with the guideline, as depicted in Figure 4.29, in order to ensure, that each functional requirements follows a particular naming convention. The guideline is interpreted following the guideline's edges starting from its *init* node. If the first statement node is reached, the statements context element type (*FunctionalRQ*) is identified in order to allow for querying the model for corresponding model elements. Afterwards, the OCL statement, which was derived during the preparation phase using the translational semantics, is forwarded to a standard OCL interpreter. The translated

statement (listing 5.8) is evaluated for any identified functional requirement. This results in a positive result,i.e., each guideline has a name unequal to "null", and triggers the guideline's *OR*-node, which triggers the *ORMerge*-node. Subsequently, the following statement, which is illustrated in listing 5.9, is validated. Therefore, the model is queried for functional requirements, on which the constraint is evaluated. However, this time, the result of the validation is negative,i.e., there are functional requirements, which do not follow the defined naming convention. As a result, the editor provides a developer with an error message, as depicted in Figure 5.22. The error message provides us with a hint to correct the model, whereupon we change the initial characters of the functional requirement's name to "FRQ_". Subsequently, due to the absence of failures, the process can be continued switching the process context from the actual MC to the next MC.

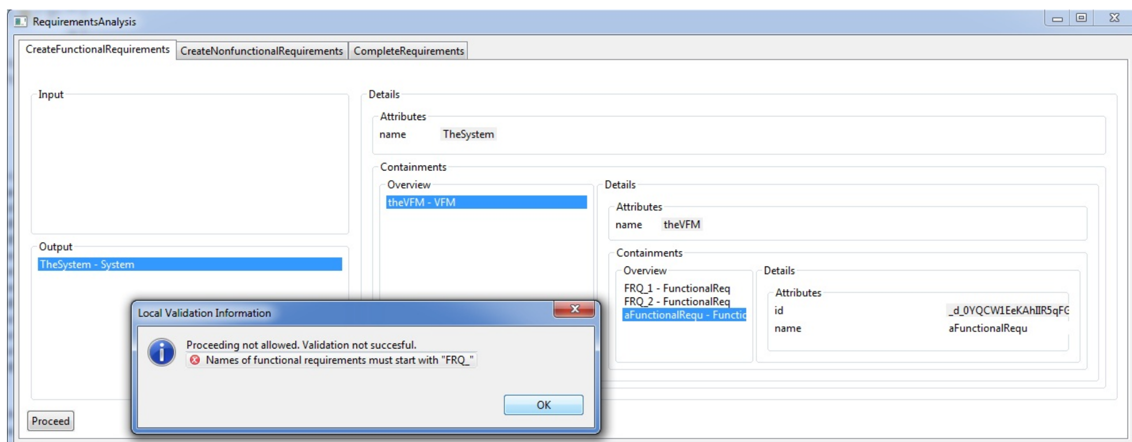


Figure 5.22: Validation of "Create Functional Requirements"

Now, the actual MC *CreateNonFunctionalRequirements*, is guided by another editor, which provides a developer with different capabilities, as illustrated in Figure 5.23. Contrasting the first editor, now a developer is enabled to derive non-functional requirements from functional requirements, which can be displayed using the editor's input master section in section *J*. The master section *K* enables to display output information, which are non-functional requirements in the context of the actual MC. According to the MC's specification, only non-functional requirements can be instantiated, modified or deleted. Furthermore, Figure 5.23 shows an MC-specific restriction of the meta model to relevant data. While the meta model specifies any requirement, i.e., functional and non-functional ones, to have an identifier attribute, the editor neglects this attribute due to its irrelevance in the context of this activity (section *L*).

After non-functional requirements are defined, and if all specified guidelines validated positively, the process proceeds with the MC *CompleteRequirements*. This MC is guided by the editor, which is depicted in Figure 5.24. One can see, that the editor, provides a view on both types of requirements (functional and non-functional ones) in order to validate potential correspondences between them. The EUSA attribution, which was made for this MC, enables the creation of additional requirements during this activity. We assume, that an additional functional requirement is defined during the *CompleteRequirements* ac-

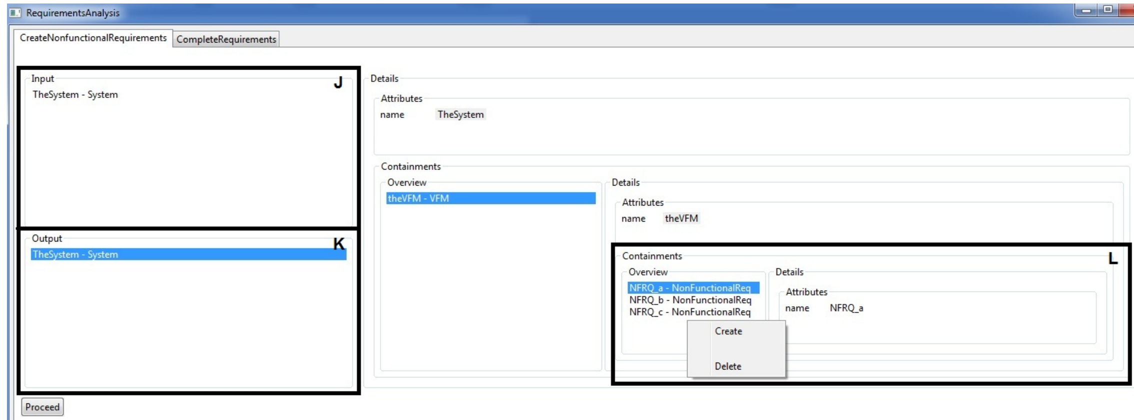


Figure 5.23: Execution of "Create Non-Functional Requirements"

tivity, in order to demonstrate the coordination and change impact resolution capabilities of our framework in the following.

After the MC *CompleteRequirements* is completed, the *Analysis* phase, which we defined

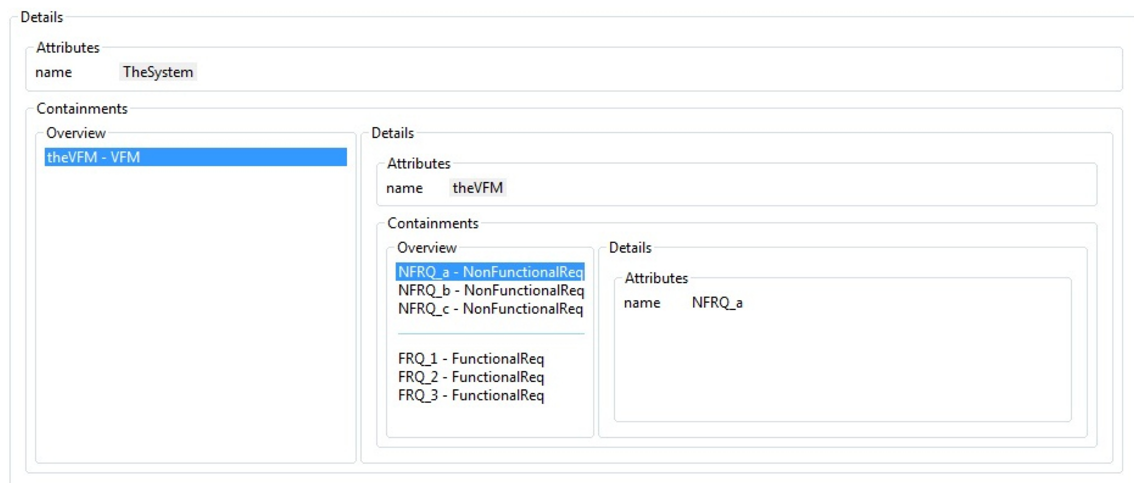


Figure 5.24: Execution of "Complete requirements"

as FPC, is finalized as well. The finalization of an FPC initiates the execution of a consistency evaluation, as described in Section 5.6.4. By analyzing the notification events, which were monitored during the actual finalized FPC, potential inconsistencies can be identified using the discussed strategies, whereupon the control-flow is coordinated flexibly. To demonstrate this, we use the example of two events for the creation and modification of a functional requirement in the context of two different MCs. We assume, that the first event occurs during the *CreateFunctionalRequirements* MC, when an additional functional requirement is created. Let the functional requirement's name be "*FRQ_4*". A corresponding event is sent to the observer and communicated to the artifact *Functional_Requirements*, which registered a respective interest before. According to the event structure, as introduced in Section 5.6.4, the concrete event looks as follows:
$Event(CreateFunctionalRequirements, Functional_Requirements, t_x, FRQ_4, create)$,

(with x>0)
If the created functional requirement *FRQ_4* would be modified during the subsequent MC *CompleteRequirements*, then the following second event would be communicated to the artifact *combined_requirements* according to its interests:
$Event(CompleteRequirements, combined_requirements, t_{x+n}, FRQ_4, modify)$,
(with x>0 and n>0)
According to the conflict identification scenario 3, that second event would indicate a potential conflict with the artifact *Functional_Requirements*, as one of its contained elements is affected. Looking up the artifact influence table, which was prepared in Table 5.4, results in two affected MCs *CreateFunctionalRequirements* and *CompleteRequirements*. Based on this information, the encompassing FPC is identified and can be initiated for a second time to check for the absence of conflicts or to resolve them. That way, the overall *Analysis* phase is performed for a second time. Subsequently, if no conflicts have been arisen during the second execution of the *Analysis* phase, the framework enables to continue with the default control-flow of the process and starts the *Design* phase, where the MCs are guided and monitored likewise. If no conflicts are produced during this phase, a developer is guided through the implementation phase finally.

After the complete process is performed, it can be analyzed by evaluating particular characteristics, such as the duration of individual MCs, as well as, the duration of the overall execution time of the process. Additionally, arisen conflicts can be analyzed for the identification of recurring pitfalls and the derivation of best practices, which can meet such conflicts. These best practices can be incorporated as additional guidelines within corresponding MCs, to guide future processes and projects.

6 Evaluation

6.1 Motivation

In this work, we followed the design-science paradigm of information systems research, “which seeks to extend the boundaries of human and organizational capabilities by creating new and innovative artifacts” [HMPR04]. Following the two base research activities of design science research, as mentioned in [TK08], we described construction details of our approach first. Now, that we demonstrated that our system can be constructed,, we will show how it was realized and evaluate our approach from different viewpoints, as proposed by Hevner et al. [HMPR04].

1. Architecture Analysis: To demonstrate the feasibility of our approach, we developed a prototype in an iterative way. In cooperation with our industrial partners, we designed a modular architecture, which satisfies required features of an CAME environment, as discussed in Section 2.4. For evaluation purpose, we first describe the resulting architecture in detail and analyze it using the Hazard Analysis of Software ARchitectural Designs (HASARD) method, in Section 6.2.

2. Scenario-based Evaluation: In Section 6.3, we use a scenario-based analysis [Zhu05] to evaluate our approach considering different stakeholder perspectives and challenging situations for the application of our CAME environment.

3. Dynamic Analysis: To analyze a further facet of our approach, we evaluate the runtime behavior of our approach in Section 6.4. After discussing the computational complexity of most critical components in our architecture, we discuss potential bottlenecks and solutions strategies to overwhelm the enormous data volume, which must be handled.

4. Descriptive Evaluation: In Section 6.5, we evaluate our approach with regard to most prominent process assessment standards and the capabilities, which come along with our approach. Using the examples of CMMI and Automotive SPICE, we describe, which parts of the respective standard can benefit from our approach. We conclude this section with a checklist for organizations, which enables them to decide whether or not our approach is reasonable in their environment.

5. Case Study: Finally, in Section 6.6, we demonstrate all facets of our CAME environment in a comprehensive case study. Starting from a set of loosely-coupled processes taken from industry, we demonstrate the setup and application of a software process line. We also show the technical design of required variants, which

subsequently enables us to exemplify the enactment of a member of the developed process line.

6.2 Architecture Analysis

In this section, we evaluate our framework from the architectural point of view. Therefore, we first describe its conceptual architecture, before we go into detail about the realized implementation, developed plugins, and used frameworks. Finally, we use the HASARD method to evaluate our architecture.Therefore, we describe potential problems, that could come up, and show how to prevent these problems.

6.2.1 Conceptual Architecture

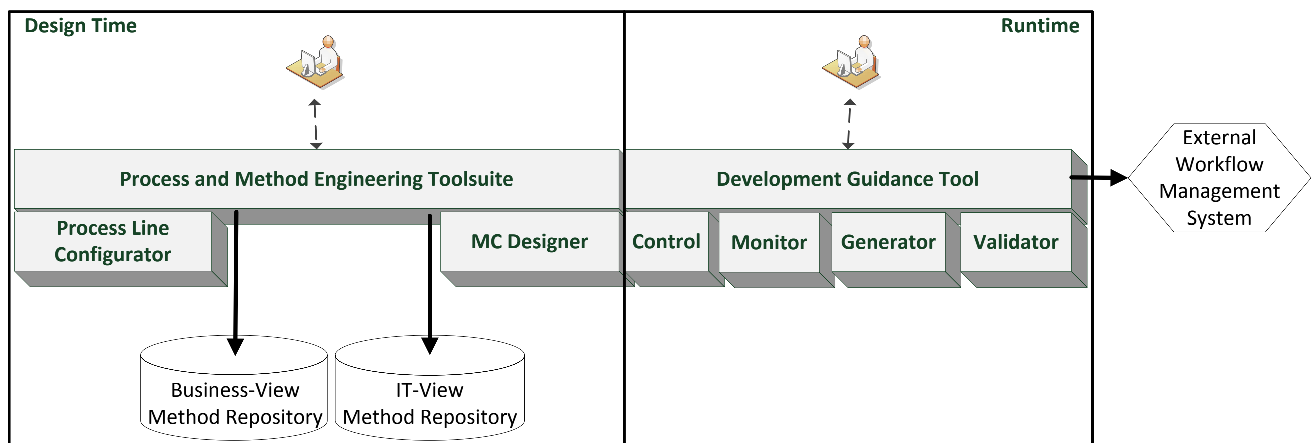


Figure 6.1: Conceptual Architecture

Our architecture is driven by the need to support the design time and the runtime of a development process. Therefore, our framework is subdivided into two parts, as illustrated in Figure 6.1. To enable SPLE capabilities and the extension of MFs on technical level, the left side of the figure shows a generic process and method engineering tool suite, by which a method engineer can interact to establish relevant design time information, as discussed in chapter 3 and chapter 4. The tool suite allows users to define processes and methods on business and on technical level, likewise. The tool suit is extended by additional components, which provide capabilities to configure, i.e., setting up and planning, the process line, and to extend basic MC information with computer-interpretable models. These components are not visible for the end user, since they are fully-integrated with the tool suite. All design artifacts are persisted in two distinct parts of a repository, by which business level information can be managed independently from information on technical level, i.e., all stakeholders are supported adequately.

On the other side, runtime support (the right part if the figure) is provided by the development guidance module, which provides an interface for end users and third party tools to communicate with the framework and to implement capabilities, as discussed in chapter 5. The development guidance module, as core module, basically realizes a workflow management system, by which a workflow, which is composed of various FPCs, is executed and coordinated. Additionally, an external workflow engine is used for the management of SPCs. To enable the development guidance module for interpreting formalized MC accordingly, various sub-modules were developed. These sub-modules are used by the guidance module autonomously to validate artifacts, to generate editors, to monitor development activities, and to analyze monitored information, by which the workflow is controlled flexibly.

6.2.2 Prototypical Implementation

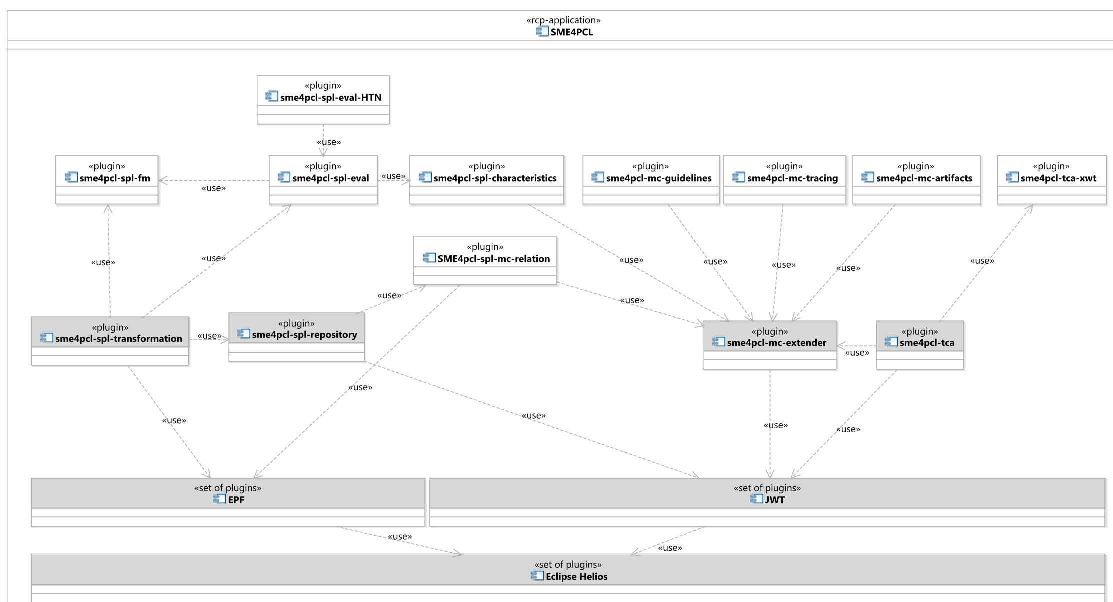


Figure 6.2: Plugin structure of the SME4PCL approach

We implemented our framework as a set of plugins on top of the open source eclipse platform, where we reuse various plugins of the eclipse *Helios* simultaneous release, such as EMF and Graphical Modeling Framework (GMF) among others. For the modeling of software development processes on business level, we reuse EPF. Although, other editors would be possible likewise, this part of our framework is based on EPF, to reuse already existing reference processes from current projects in the software development domain, such as AUTOSAR and MAENAD. In contrast, on technical level, we use the capabilities of the JWT framework, which provides us not only with sufficient process modeling capabilities, but with an aspect-oriented extension mechanism, which enables us to integrate additional information, such as guidelines or MMVs. The extension of the JWT meta model is provided by an abstract extension plugin, *sme4pcl-mc-extender*. This

plugin is the basis for any extension plugin and enables other plugins to simply retrieve modeled information.

For SPLE, the *sme4pcl-spl-transformation* plugin provides main functionality. It uses the *sme4pcl-spl-repository* plugin to access existing information from business level and technical level. In more detail, the repository plugin stores process models modeled on the level of EPF and JWT and makes them accessible. During situational process engineering, the transformation plugin allows the selection of a reference process modeled with EPF on a business level. Based on the reference process, the repository is queried for potential variants on technical level modeled with JWT. The relation between VPs and variants, is realized by the *sme4pcl-spl-mc-relation* plugin. It realizes an aspect of the JWT meta model to link MCs in form of an JWT activity with an VP, which was defined in EPF. The transformation plugin uses the reference process and available variants to generate a feature model, which is realized by the *sme4pcl-spl-fm* plugin. Afterwards, the feature model is configured using an evaluation component, which is abstracted by the *sme4pcl-spl-eval* plugin to simply apply various planning algorithms. In our concrete case, we realized the evaluation using HTN and concertized the evaluation plugin with the *sme4pcl-spl-eval-HTN* plugin. This evaluation uses characteristics defined for each process line asset, as discussed in Section 3.3.3. The annotation of characteristics is realized through an additional aspect of the JWT meta model. Therefore, we implemented the *sme4pcl-spl-characteristics* plugin.

Beside plugins, which are relevant for SPLE, we implemented different plugins to extend the JWT meta model with required technical information models. The *sme4pcl-spl-guidelines* plugin extends action elements of JWT with capabilities to add guidelines, as discussed in Section 4.6. To extend artifact element of the JWT meta model, we developed the *sme4pcl-mc-artifacts* plugin to annotate meta model information, as discussed in Section 4.4. The annotated meta model information is used by the *sme4pcl-mc-tracing* plugin to establish the *FromTo* relationships between an element type of an MC's output artifact with an element type of the relevant input, as detailed in Section 4.4.3.

While the above plugins realize the design time part of our conceptual architecture, the *sme4pcl-tca* plugin realizes the architecture's runtime part, i.e., the process control, as discussed in chapter 5. It provides an interface to other tools and controls the control-flow of the development process. Therefore, it uses aspect information of the situational process to interpret them accordingly. To generate and execute editors, it uses the *sme4pcl-tca-xwt* plugin. This plugin uses MMVs and EUSAs information, as discussed in Section 4.5, to generate editor forms and individual editor capabilities based on the XWT framework of the eclipse *e4* project.

This modular architecture was chosen to simply enable the exchange and modification of single modules and to enable the validation of distinct functionalities without the need to set up the overall framework in critical environments.

6.2.3 Evaluation with HASARD

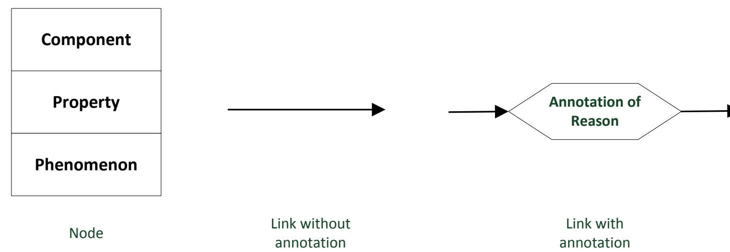


Figure 6.3: Notation of HASARD quality models according to [ZZHG]

To evaluate our framework's architectural design different approaches, such as Architecture Tradeoff Analysis Method (ATAM) [BCK03] or Software Architecture Analysis Method (SAAM) [BCK03], are available. However, to particularly focus on unknown problems of our software architecture, i.e., to discover quality features of our design without pre-specified requirements, we apply a method called HASARD [Zhu05].
HASARD is model-based analysis technique based on a safety analysis technique called hazard analysis. To evaluate software architectural designs the method makes use of quality models, which are represented by a diagrammatic notation, as illustrated in Figure 6.3. A quality model is a directed graph, which consists of nodes and two types of links. Each node is composed of three information specified in three compartments: the component which represents a system element, a quality-carrying property of the element, and an observable phenomenon of the property.
In contrast, links are directed edges between nodes. Basically, a link between a node A and a node B means, that the occurrence of A's phenomenon implies the occurrence of the phenomenon associated with B. Additionally, each node may have an optional annotation to give a rationale for that implication.

For the construction of quality models, HASARD provides a methodology, which consists of four steps. Before we detail our framework evaluation, where we applied HASARD, we shortly introduce the four steps of the method we performed.

1. **Hazard identification:** For the software development domain, a hazard can be defined as a condition, event, or circumstance, that could lead to an undesired or unplanned behavior of the software. During hazard identification, potential hazards of a system are identified. A prominent method to identify hazards is the so called Hazard and Operability Studies (HAZOP) method. HAZOP was developed in the 1960s as a method, which relies on determining questions of what-if nature. To determine relevant questions, the method provides a set of guide words, which can be applied to study features of the system of interest. HAZOP proposes the following guide words, whereas not all of them are applicable to any domain or property: *No, More, Less, As well as, Part of, Reverse, Other than, Early, Late, Before, After*. The questions result in potential misbehavior of the system. Such a failure

mode is considered as possible hazard, for which causes, consequences and recommendations are documented. All identified hazards are then utilized for the subsequent cause-consequence analysis.

2. **Cause-consequence analysis:** This kind of analysis progressively selects the hazards identified in the last step and investigates their causes and consequences until the analyst is satisfied with the coverage of the most threatening hazards. Thereby, potential interdependencies between consequences, which were identified in 1, are analyzed and documented. This might also bring up new hazards, that have not been recognized in the first step. The cause-consequence analysis can be performed forward (from a hazard to search for potential effects) and backwards (from observable hazard to the causes).

3. **Model assembling:** The identified hazards and interdependencies, which were identified during the last two steps, now are translated into a graphical representation, according to the notational elements, as discussed before. Each hazard (failure mode) becomes a node and each identified interdependency becomes a link with an optionally annotated cause.

4. **Quality concern analysis:** Finally, the property slot of the quality model's nodes must be filled. To do so, for each node the observable phenomenon is analyzed considering a list of quality attributes in order to identify the affected quality of the system, such as reliability or usability.

Following these steps, we performed HASARD as follows: first, we identified potential hazards with regard to our framework's main sub-systems, i.e., SPLE, CME, and runtime guidance. All potential hazards (failure modes) and corresponding guide words are summarized in table 6.4 to table 6.6. While table 6.4 summarizes all hazards, which belong to SPLE approach, such as what happens if no variants are defined, table 6.5 summarizes the identified hazards, which can occur during CME, such as a malformed statement model. Table 6.6 summarizes all hazards, which are relevant to the guidance or execution phase, such as the disability of the monitoring of modeling events.

Using the identified hazards, we performed a cause-consequence analysis and assembled a quality model from it. The resulting quality model is illustrated in Figure C.1. The quality model also includes the result of the quality concern analysis, where we identified the following quality attributes: Acceptance, Availability, Functionality, Usability, Comprehensibility, Reliability, and Correctness.

The quality model is used to identify the contribution factors for specific quality concerns. For example, a bad user acceptance results from an unavailable workflow engine, incorrect guidance, the necessity to wait for a long time until a situational process is planned, process re-planning, or to the disability to generate (correct) editors. Hereby, the latter factor depends on the correctness of annotated MMVs and corresponding EUSAs. The quality model is used to derive quality features of our design, such as the quality attributes that are affected by a design decision or the consequences of a particular failure

Ref	Guide word	Failure mode
SPLE1	No	Connection to repository does not exist
SPLE2		No variants are available
SPLE3		No charcteristics are annotated
SPLE4		No correct planning result during situational process engineering
SPLE5		Variants are not linked with variation points
SPLE6	More	Too much planning results
SPLE7		More variants match with the needs of one variation point
SPLE8		The repository provides huge number of variants
SPLE9		Each variant is annotated with a lot of characteristics
SPLE10	Less	There are not enough variants to fullfill all variation points of the reference process
SPLE11		Variants are not provided with enough characteristics
SPLE12	After	Repository changes after a process was configured
SPLE13		A planned process must be re-planned

Figure 6.4: Hazards belonging to SPLE components

Ref	Guide word	Failure mode
Design1	No	No meta model information is available
Design2		An artifact is not annotated with an MMV
Design3		An action is not annotated with any guideline
Design4		Meta model information can not be used to restrict the statement vocabulary
Design5	More	More guidelines defined than required
Design6	Less	Not enough guidelines defined
Design7	Part of	Only parts of a MC's fragments are refined on technical level
Design8	Other than	MMV are modeled without considering methological needs
Design9		Guideline statements were modeled incorrectly
Design10		Statement Code can not be generated correctly
Design11		The EUSAs of an MMV are not set accordingly
Design12	Before	Statement design is performed before MMV definition

Figure 6.5: Hazards belonging to CME components

Ref	Guide word	Failure mode
Exec1	No	No guidelines available
Exec2		No MMV available for individual artifacts
Exec3		No workflow engine available
Exec4		Customized editor can not be generated
Exec5		No connection between editor and framework
Exec6		No modeling events are monitored
Exec7	More	There were too much events monitored
Exec8	Less	Less modeling events mointored than required
Exec9	Other than	The guidelines have a different format than expected
Exec10		The framework provides guidance, which is not correct or different from expected guidance
Exec11		Generated editor behaves different from expected behavior
Exec12		The framework is not enabled to detect inconsistencies between artifacts
	After	After process is deployed, some MCs change

Figure 6.6: Hazards belonging to Guidance components

on other system components. Furthermore, we evaluate whether or not our introduced framework is capable to face all the objectives, as introduced in Section 1.2. For objective 1, where we demanded, that executable development processes must be constructed in reasonable time, our SPLE component provides support. After a considerable initial effort to set up the method repository, situational process engineering drastically reduces process engineering activities for up-coming projects. Of course, a meaningful repository takes a lot of time, but from our point of view, such a repository not only minimizes future design efforts, especially in combination with a planning tool, but also simplifies the management of methodological knowledge.
Considering objective 2, where we required to close the gap between various information sources, which are relevant to some development process, the CME component provides support and is a good starting point for further research. We demonstrated, that the information content of various MFs and today's control-flow semantics can be enhanced significantly. The extension of (textual) process model documentation with additional computer-interpretable information about statements, the allocation of meta model, or editor information in the context of MCs improve process knowledge more purposefully. Indeed, technical design is a very time-consuming task for method engineers, but in combination with the SPLE component and the associated repository, reuse and future return of this investment can be ensured.
Finally, our architecture also fulfills objective 3 by its guidance component. Through its capabilities to interpret MF information and to control the development process by considering traceability concerns, it definitely enables a process to be more than "paperware" only.

The HASARD analysis shows, that many factors influence the correct behavior of our framework. Therefore, a detailed analysis of the organizational process landscape is essential, before realizing the discussed concepts. Although, this is a time-consuming and

error-prone task, our approach accommodates the increasing complexity of knowledge management in today's enterprises providing a sustainable solution.

6.3 Scenario-based Evaluation

In this section, we apply a scenario-based evaluation to discuss individual properties of our framework to react on changing environments and conditions. In various realistic scenarios, we discuss the modifiability of the framework (Section 6.3.1), the reusability of design artifacts (Section 6.3.2), and the maintainability of design artifacts (Section 6.3.3).

6.3.1 Scenario-based Modifiability Evaluation

The following four scenarios were identified, to evaluate the modifiability of our framework. Thereby, we particularly focus on changing environmental characteristics.

Scenario 1: *The actual platform, on which our framework is based on, is the eclipse Helios release and various simultaneous releases of other plugins and frameworks, such as EMF, GMF, JWT, and EPF. Over time, different components can change, while the framework must be compatible with new components.*

During the prototypical implementation, we were required to deploy our framework on various eclipse-based platforms, to demonstrate the applicability of our approach in industrial context. Each platform was characterized by different releases of relevant plugins and frameworks. In some environments, even some components were missing at all. In spite of these different platforms, each time we were able to make our framework work. This is due to our industrial partner requires high flexibility from the framework and a high degree for integration ability from the very beginning of our joint project. From the technical point of view, this results from the usage of mature and well-known frameworks, such as EMF, GMF, and OCL. Since these technologies actually are used in the context of various industrial applications and since all the used technologies are characterized by a good compatibility, our framework is prepared to face various changed platform requirements.

Scenario 2: *Due to a change of an enterprise's tool landscape, there could be a need to change the process modeling tools from current tools, i.e., EPF and JWT, to other process modeling environments. The functionality of the general framework could be not compatible with the underlying process modeling tool.*

For this scenario, we must distinguish the change of the process modeler on business level from the change of the modeler on technical level. On business level, almost any process modeler can be used in combination with our framework without great efforts.

The only challenge, which must be mastered for this situation, is to adapt or to create a generation of the feature model in the context of situational process engineering, as discussed in Section 3.4. We discussed relevant process components on business level from a conceptual point of view. The conceptual view enables an easy adaption to a different PDL, such as BPMN, SPEM, or Architecture of Integrated Information Systems (ARIS). In our prototypical implementation, we exemplified a transformation of a reference process on business level, which was modeled using EPF (i.e., SPEM), and MCs modeled using the JWT into a feature model. When the business process modeler changes, the generation of the feature model must be adapted to the corresponding meta model of the respective modeler. Furthermore, a link between variation points on business level and variants on technical level must be established, as demonstrated in our prototype.
The change of the modeler on technical level is a more complex task. On technical level, we used JWT due to its excellent extension capabilities, which is provided by the aspect-oriented mechanism. An alternative modeler on that level, must likewise provide the capability to extend MFs, as discussed in chapter 4. Since other process modeling environments do not provide an as simple extension mechanism for most of the time, it would be more complex to integrate all additional information with the model. However, it is possible to change the technical process modeler, as well.

Scenario 3: *An increasing need to integrate more information with process models requires the framework to incorporate even more details. This induces the introduction of additional types of MFs, which must be annotated with technical and computer-interpretable information.*

This scenario is faced very simple due the extension mechanism of JWT. For each additional MF (given that the meta model of JWT does support this MF), which should be supported, the aspect-oriented mechanism easily enables the extension of the fragment, with additional information. Therefore, one has to extend the meta model with a new model to represent the desired information. Additionally, the guidance component of our architecture must be extended by a corresponding interpreter component. The development efforts of such an interpreter depend on the characteristics of the additional model.

Scenario 4: *For planning a situational process, various planning techniques are available. Different pros and cons of alternative techniques lead to the decision to exchange the planner for SPLE, which actually is HTN, by a different planner.*

To change the concrete planning technique for SPLE, it is sufficient to provide a new component, that interprets the generated feature model, which we introduced as intermediate model. In our future research, we plan to exchange the HTN planner with an Planning Domain Definition Language (PDDL) planner. That means, that we must write a transformation between the generated feature model and the PDDL. Afterwards, the planning result must be interpreted accordingly to configure the feature model, but other components are unaffected.

6.3.2 Scenario-based Reusability Evaluation

Subsequent scenarios focus the reuse capabilities of our framework. We discuss scenarios, in which particular process line assets are changed or must be changed due to environmental changes.

Scenario 5: *Although, an enterprise's reference processes are stable for most of the time, they can change from time to time due to changing product or customer requirements. For example, a reference process is enhanced through more agility and agile development phases in order to come up with shorter development cycles. Another example for changing reference processes is the shift from the waterfall style to a V-Model.*

The exchange of a reference process in our framework is not a complex task. After a new reference process is defined, some variation points could be reused, which means, that also the relationship between the VP and corresponding variants is still established. If the reference process introduces new variation points, the relationships to already existing variants and new variants, which face the particular needs of an VP must be established. Afterwards, SPLE works as usual without side effects to other components.

Scenario 6: *Customer requirements or standards and regulations change over time, so that the process behavior is influenced. This induces changing compliance requirements regarding required artifacts, analyses, and techniques.*

If process requirements change, one can simply re-plan the process with changed configuration criteria. Having a sound repository of MCs, which fulfill the requirements of the changed situation, enables to create and deploy an adapted process on the fly. Of course, running processes are obsolete and must be terminated, before a new process is started. Artifacts, which were created during the obsolete process, can be reused for modeling and traceability analyses, as before.

Scenario 7: *The application domain of our framework changes from, e.g., automotive to a different one, such as avionics, finance, or a general RCP application.*

Changing the application domain probably is the most time-consuming scenario. Although, some artifacts, such as individual GEs, VPs, and MCs, can be reused depending on the new domain, most artifacts must be created from scratch. In detail, this means, that all activities of process family engineering must be applied, as discussed in chapter 3. Therefore, a new reference process, which matches the needs of the new domain, must be defined, and the repository must be extended by missing MCs applying CME techniques from chapter 4. In parallel, distinguishing configuration criteria for process line assts must be defined considering the new domain requirements, before they are combined with the existing ones. The effort to set up the environment for a new domain, depends

on the differences between the two domains and the re-use factor of existing artifacts. As larger the overlap between the domains, as more artifacts can be re-used.

Scenario 8: *Individual parts of the guidance component are changed to other technologies or frameworks. For example, guidelines could be based on a language, which is different from OCL, or editors must be generated based on Standard Widget Toolkit (SWT) instead of XWT.*

Changing a guidance component does not influence existing design artifacts from business and technical level. The existing repository information, as well as configured processes can still be used, if the new guidance part is able to interpret respective information accordingly. Due to the modular architecture of our framework, single components can by exchanged simply without side effects. For example, to change the editor generation functionality from XWT-based editors to, e.g., SWT, the new generator only has to interpret existing MMVs accordingly. Furthermore, the generation must consider the communication interface of the framework to use relevant features, such as monitoring modeling events, validation and control. In contrast, when the validation is changed from, e.g., OCL, to a different language, such as EVL, the translational semantics, as discussed in Section 5.5.1.2, must be adapted correspondingly.

6.3.3 Scenario-based Maintainability Evaluation

To evaluate the maintainability of framework artifacts, the following four scenarios discuss situations, when individual circumstances induce a change of individual artifacts.

Scenario 9: *Guidelines, which are annotated with individual MCs, evolve over time. Due to changing requirements and lessons learned, guidelines must be kept in-line with processes performed in the future.*

Our approach enables a more easier management of guidelines and validation rules, than general modeling environments. Due to the locality of guidelines to a relevant MC, only a small set of guidelines must be considered for each development activity. New information or lessons learned, which concern an individual MC, can be realized purposefully. By adapting the graphical model of a guideline in the context of an MC, associated statements can be (re-)generated on the fly. Likewise, new guidelines can simply be added or removed to an existing MC. Hereby, the graphical representation of statements and guidelines, simplifies the understanding and adaption of guidelines. This also holds for employees, which have less technical knowledge.

Scenario 10: *The editors, which support the development process, are strongly connected with methodological needs. Therefore, also the capabilities of an editor change in parallel with the further development of individual methods.*

The increasing complexity of modern software systems, also causes improvements and further development of existing methods, which must be supported through editors. Since our approach provides capabilities to generate editors for the situation at hand based on MC information, such as MMVs and EUSAs, it becomes more easier to react on methodological evolution. By changing the meta model information and changing the associated EUSAs, situational editors can be generated more efficiently and are in-line with state-of-the-art methodological requirements. It is also possible, to integrated COTS tools with our framework, if their capabilities exceed the capabilities of standard generated editors. 3rd-party tools can implement the framework interface to notify modeling events and to use the other capabilities of the framework, such as validation and control. This only involves few extensions to be made in the COTS functionality: Modeling events must be notified, the framework must be enabled to initialize the editor, if it is required by some method, and it must be possible to send a command, which indicates the completion of an activity.

Scenario 11: *Changing product requirements cause changing design documents, which involve the change of underlying meta models. Changed meta model information must be aligned with the processes, that apply the changed meta model.*

A changed meta model influences the artifacts and the guidelines of individual MCs. If a meta model changes, two sub-scenarios can be identified. First, the meta model evolves to a new version, i.e., at least the name space of the meta model's elements changes. Therefore, to keep affected MMVs in-line with the new meta model, all elements in already designed MMVs must adapted adequately. If the new release of meta model provides additional elements, they can be added to existing MMVs. Second, if a meta model is changed, e.g., from UML to SysML, relevant artifacts, which use UML, must be provided with a new MMV, which provides respective artifacts with SysML-specific meta model elements. Since MMVs directly influence the statements of a guideline, i.e., they define the vocabulary, the statements must be adapted to follow the new vocabulary.

Scenario 12: *Lessons learned, project reviews, assessments, and new process requirements lead to the need to change the method repository.*

Lessons learned simply can be integrated with the framework, i.e., the repository, by the construction of additional MC or the adaption of existing ones. A detailed definition of the application scenario of a new MC by the means of annotated configuration criteria enables a correct involvement of lessons learned during the situational process engineering phase of SPLE. For subsequent projects, the planning considers the changed state of the method repository and uses the most recent lessons learned.

Summary of the Scenario-based Evaluation The discussed scenarios demonstrated the capabilities of our framework to be re-used, modified, and maintained. On the one hand, this results from abstraction levels, where a more abstract level only influences the

level below (e.g., the business level only affects the technical level). On the other hand, this results from the modular software architecture of our framework. Furthermore, since CME enables a high integration of various information sources into one target, maintainability of knowledge becomes more easier.

6.4 Dynamic Analysis

In this section, we discuss the complexity and runtime behavior of our approach. Due to missing objectivity in manually performed design activities, we first focus on the automated planning of a situational process in the context of SPLE. Afterwards, we discuss the runtime behavior and potential overhead of method-driven guidance.

6.4.1 Performance Evaluation of Software Process Line Engineering

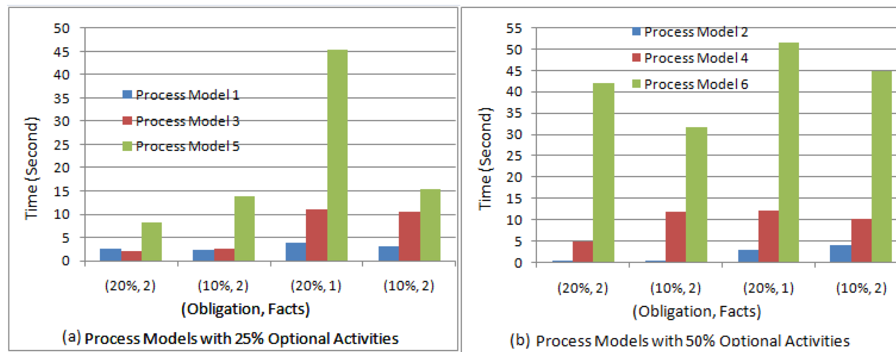


Figure 6.7: Evaluation Results for different process models

By the means of the case-study, as discussed in Section 3.6, we already evaluated the applicability of the SPLE approach in general. Additionally, in order to evaluate scalability and performance of SPLE, we conducted three experiments with a different number of VPs for a reference process and a various number of variants. In our tool chain support, three main automated activities are involved, which include the generation of feature models from reference process and variants, calculating ranks of concerns from relative importance (i.e., weight w in utility function), and finding an optimal configuration during the planning process. Generating feature models and executing S-AHP require polynomial time. Hence, we ignore them for the evaluation and only concentrate on finding optimal configuration for different situations.

To evaluate the derivation of situational processes from the process family, we generated six reference processes with 15, 25 and 50 abstract activities (i.e., VPs), where either 25% or 50% of activities were optional. Table 6.1 shows the descriptive information about these reference processes, including a total number of activities, the percentage of optional activities, as well as a number of variants for each process model. These process models are considered based on the analysis of our case study. We used between

1 to 4 variants for each VP in the reference process. Additionally, we considered 4 facts and 4 concerns. For each concern, 5 qualifier tags were defined. We defined obligations between variants in the reference process. All the variants within the repository are randomly annotated with 0 to 4 facts. Additionally, all the variants are annotated with 4 concerns.

We applied our process and method engineering tool suite, as well as the process line configurator, i.e., the planner, for generating the feature model and deriving situational processes based on random relative importances between soft-facts, concerns, and constraints over hard-facts. We used a computer with an Intel Pentium 4 CPU 3 GHz, 2GB of RAM, Windows XP, an up-to-date Java Runtime Environment6, and SHOP2 v2.8.

In our experiments, the independent variables are the total number of VPs (i.e., abstract activities) and variants for reference processes (40, 50 and 100), the percentage of optional activities in the process models (25% and 50%), the percentage of obligation (15% and 30%), and the number of hard-facts (2 and 3). The dependent variable in our experiment is the execution time for deriving situational processes.

Figure 6.7 shows results for the processes with 25% (a) and 50% (b) optional activities. As shown in the figure, our approach returns a situational process in feasible time. Results reveal that all factors (i.e., independent variables), i.e., size of process, number of obligations, number of selected factors, and the percentage of optional activities or VPs, influence process derivation. Generally, an increasing number of VPs and variants in the process models, as well as, the number of obligations has direct impact on increasing execution time. The former is because it forces the tool to evaluate more combinations of configurations and the latter is because it adds more complexity to the selection of variants. Interestingly, the results show, that the number of VPs and variants have larger impact on execution time than obligations. On the other hand, adding more constraints (selecting more hard facts) leads to a decreasing execution time. This was expected, since the increase of hard facts causes filtering more variants, which decreases the configuration space.

Comparing the results in part (a) and part (b) of Figure 6.7 shows, that with the same number of activities and variants in a process model, increasing the percentage of optional activities generally leads to higher execution time for finding the situational pro-

	No. of activities	Pct. of optional	No. of Variants
Process Model 1	15	25%	40
Process Model 2	15	50%	40
Process Model 3	25	25%	50
Process Model 4	25	50%	50
Process Model 5	50	25%	100
Process Model 6	50	50%	100

Table 6.1: Process models detail descriptions

cesses. However, as planning is independent from process execution, duration of planning is not a critical obstacle for large problem spaces, as well.

6.4.2 Runtime Complexity of the Application of Computational Method Engineering

Basically, the runtime overhead, which is induced by the application of information as annotated with MCs on technical level, is very low. As most data structures and platform-specific code is generated before the project is started, a developer's work is nearly not affected by the application of our framework.
In general, the coordination of short-dated SPCs is under the control of a workflow engine, which assigns activities in a procedural linear way. Only the coordination of FPCs, requires a more complex analysis of monitored design activities by applying the rules, as introduced in Section 5.6.4. This is influenced by the consistency evaluation phase (cf. Section 5.6.4) more, than the setup phase, which creates the influence table in linear time. The discussed evaluation of performed modeling events and their respective influence on individual artifacts must be performed by a pairwise comparison of all events. This means a runtime complexity of $O(n^2)$ with n being the number of all modeling events, which were performed during the project. However, this unwanted complexity can be reduced by ordering the events considering the associated time-stamp, as well as the affected data type. The sorting of events, which comprises a worst-case complexity of $O(n) \times \log(n)$, results in a time-line of data type specific events, which reduces the number of relevant events drastically. An additional optimization can be reached by the introduction of individual strategies, which are restricted to the events of a particular set of FPCs.
Contrasting the control of FPCs, the complexity of the application of technical MF information is negligible. The generation of editors, as discussed in Section 5.3 is performed before runtime, and does not influence the developers's work negatively.
Our observer mechanism likewise does not bring negative effects. The sending of notifications between an editor and the dispatcher is restricted through manual activities, which must be stored in the context of an artifact. Although, this requires more memory, we think, that nowadays this is a reasonable situation. On the other side, the retrieval of individual elements for validation depends on the applied model query language, but this should be realized with a acceptable complexity of $O(n)$, where n is a model's number of instance elements.
As statements of a guideline are likewise transformed into platform-specific code before runtime, the evaluation of them at runtime does not take more time, than conventional validation frameworks. In contrast, the application of statements, which are relevant to the actual method exclusively, reduces the number of statements to be validated drastically. Furthermore, the interpretation of a guideline is performed in linear time.

6.5 Descriptive Evaluation

In this section, we evaluate the reasonableness of introducing our approach. Therefore, in Section 6.5.1, we first describe the main characteristics of process assessment or improvement standards, using the example of CMMI and Automotive SPICE. In accordance with process management requirements to achieve different maturity levels, we discuss the main challenges, which enterprises can face by establishing our approach. Beside that, we discuss our approach with regard to its possibilities to improve an enterprise's process landscape. Finally, we provide a checklist for enterprises, which enables them to decide about the approach's reasonableness and discuss potential pitfalls (Section 6.5.3).

6.5.1 Process Improvement Standards and Maturity Levels

Although, our approach in general would be applicable to any kind of software development, there are some prerequisite characteristics, which an enterprise should fulfill to reasonable set up such a complex project. Therefore, since process assessment models, as discussed in the following, provide characteristics to attest the process-related level of expertise of an enterprise, we compare these characteristics with contributions and needs of our framework. We describe CMMI and Automotive SPICE as prominent members of process assessment models, as well as relevant characteristics, which an enterprise must fulfill to reach a particular level of expertise. Based on the expertises of an organization, we subsequently will discuss, how individual levels can be reached or supported using our approach.

6.5.1.1 Capability Maturity Model Integration

CMMI [CMM10] was developed by the Software Engineering Institute (SEI) at the Carnegie Mellon University to provide organizations with the essential elements for effective process improvement. Best practices are published as so called models, which focus on a particular area of interest. Actually, version 1.3 of CMMI supports three areas of interests (*development, acquisition, and services*), whereas development is most relevant to our work. The model refers to five process area categories, which are relevant to achieve an individual maturity level of an organization. These categories are *support, project management, process management, and engineering*. For each of these categories various process areas were identified, to provide best practices considering relevant activities, outcomes, and other generic practices. To achieve a particular maturity level, the achievement of the specific and generic practices associated with each predefined set of process areas must be measured. The five maturity levels of CMMI are defined in [CMM10] as follows:

- At maturity level 1 (**Initial**), processes are usually ad hoc and chaotic. The organization usually does not provide a stable environment to support processes. Success in these organizations depends on the competence and heroics of the people in the organization and not on the use of proven processes. In spite of this chaos, maturity level 1 organizations often produce products and services that work, but they frequently exceed the budget and the schedule documented in their plans.

- At maturity level 2 (**Managed**), the projects have ensured that processes are planned and executed in accordance with policy; the projects employ skilled people who have adequate resources to produce controlled outputs; involve relevant stakeholders; are monitored, controlled, and reviewed; and are evaluated for adherence to their process descriptions. The process discipline reflected by maturity level 2 helps to ensure that existing practices are retained during times of stress. When these practices are in place, projects are performed and managed according to their documented plans.
- At maturity level 3 (**Defined**), processes are well characterized and understood, and are described in standards, procedures, tools, and methods. The organization's standard processes, which are the basis for maturity level 3, are established and improved over time. These standard processes are used to establish consistency across the organization. Projects establish their defined processes by tailoring the organization's standard processes according to tailoring guidelines.
- At maturity level 4 (**Quantitatively Managed**), the organization and projects establish quantitative objectives for quality and process performance and use them as criteria in managing projects. Quantitative objectives are based on the needs of the customer, end users, organization, and process implementers. Quality and process performance is understood in statistical terms and is managed throughout the life of projects.
- At maturity level 5 (**Optimizing**), an organization continually improves its processes based on a quantitative understanding of its business objectives and performance needs. The organization uses a quantitative approach to understand the variation inherent in the process and the causes of process outcomes.

6.5.1.2 Automotive Software Process Improvement and Capability Determination

Automotive SPICE [Aut10] is property of the German Association of the Automotive Industry (VDA) and was developed by the Automotive Special Interest Group (AUTOSIG) since 2001 as an automotive-specific variant of the original SPICE, aka. ISO/IEC 15504. Similar to CMMI it defines a model for the assessment and the improvement of development processes with a particular focus on the assessment of the performance of control unit manufacturer in the automotive domain. The model mentions 3 assessable life-cycle processes: *Primary processes*, such as acquisition, supply, or engineering, *supporting processes*, such as configuration management or change request management, and *organizational processes*, such as management, process improvement, or reuse.
For the assessment, SPICE defines 5 capability levels, which are analyzed considering particular process attributes on each level. The capability levels of SPICE are defined in [Aut10] as follows:

- Level 0 (**Incomplete process**): The process is not implemented, or fails to achieve its process purpose. At this level, there is little or no evidence of any systematic achievement of the process purpose.

- Level 1 (**Performed process**): The implemented process achieves its process purpose.
- Level 2 (**Managed process**): The previously described Performed process is now implemented in a managed fashion (planned, monitored and adjusted) and its work products are appropriately established, controlled and maintained.
- Level 3 (**Established process**): The previously described Managed process is now implemented using a defined process that is capable of achieving its process outcomes
- Level 4 (**Predictable process**): The previously described Established process now operates within defined limits to achieve its process outcomes.
- Level 5 (**Optimizing process**): The previously described Predictable process is continuously improved to meet relevant current and projected business goals.

To assess the achievement of an individual capability level, level-specific process attributes must be proven using associated generic practices, which are defined in the standard alike.

6.5.2 Process Improvement with Situational Method Engineering for Process-Centric Languages

In general, assessment standards provide enterprises with a set of standard work products and activities, which must be conducted to support various process areas best. Conducting the main activities to produce required outcomes, thereby, bases the minimal level of maturity, on which further improvement activities can be achieved. Therefore, to ensure a standard-compliant execution of the primary process area of engineering, our approach enables the guaranteed compliance with required activities and work products, as defined in some standard. Automated support for workflow management not only ensures the accomplishment of relevant development activities, but also the construction of required work products. As a result, especially, the achievement of a standard's initial level for the engineering process area is enabled by our framework.

To achieve more mature levels, more and more measurement and optimization capabilities as well as an institutionalization of knowledge management are required from an enterprise to realize controlled and managed processes in all areas. Therefore, standards require the realization of an additional set of process attributes grouped into capability levels. The process attributes are features of a process, that can be evaluated on a scale of achievement, providing a measurable characteristics of process capability. Due to their generic characteristic, they are applicable to various process areas in most cases. The following lists generic practices, which are used for Automotive SPICE and defined in ISO/IEC 15504-2, to determine a distinct level of process maturity. From 36 practices, which are introduced by SPICE, we here focus on the 17 practices, which benefit from our approach. We refer to the respective practices by referencing a supporting component or technique, which we introduced in our approach:

GP 1.1.1 The guidance of a development process using a workflow management system, as discussed in Section 5.6, supports the achievement of the process outcomes

GP 2.1.2 The configuration (cf. chapter 3) and execution (cf. chapter 5) of situational and project-specific processes, supports planning and monitoring of the process performance

GP 2.1.3 Monitored information about modeling activities (cf. Section 5.4) and processing times of individual tasks, which are provided by the workflow management system (cf. Section 5.6), enable an analysis of these data to subsequently adjust the process performance

GP 2.1.4 The design of methods using role-oriented MFs, as discussed in Section 4.7, ensures the definition of responsibilities and authorities for performing the process.

GP 2.1.5 By linking the process model and methods with computer-interpretable guidelines (cf. Section 4.6) and documents, resources are identified and made available to perform the process according to the plan.

GP 2.2.1 The definition of MMV (cf. Section 4.4) and associated guidelines (cf. Section 4.6) clearly allows to define the requirements for the work products.

GP 2.2.2 Computational Method Engineering, as described in chapter 4, defines the requirements to document and control the work products.

GP 3.1.1 The definition of a reference process, as described in chapter 3, is a standard process that enables the deployment of a defined process.

GP 3.1.2 The design and configuration of process line assets to achieve a situational process life-cycle based on a reference process (cf. chapter 3), implicitly determines the sequence and interaction between processes, so that they work as an integrated system of processes.

GP 3.1.3 The design of methods using role-specific MFs, as discussed in Section 4.7, enables the identification of the roles and competencies for performing the standard process.

GP 3.1.4 By the definition of required editor capabilities (cf. Section 4.5), the identification of required infrastructure and work environment for performing the standard process is supported.

GP 3.2.1 By applying SPLE principles, as discussed in chapter 3, the deployment of a defined process, that satisfies the context-specific requirements of the standard process, is ensured.

GP 3.2.2 By the definition of a role-specific MF, which is annotated with skills and additional policies (cf. Section 4.7), the assignment and communication of roles, responsibilities, and authorities is enabled for performing a defined process.

GP 3.2.4 By the generation of customized editors (cf. Section 5.3), resources and information to support the performance of the defined process can be provided automatically.

GP 3.2.5 Situational processes with computer-interpretable information provide an adequate process infrastructure to support the performance of the defined process.

GP 3.2.6 Monitoring of modeling events and cycle time information from a workflow management system, enable to collect and analyze data about performance of the process to demonstrate its suitability and effectiveness.

GP 4.1.5 Monitoring of modeling events and cycle time information from a workflow management system, enable to collect product and process measurement results through performing the defined process.

However, our approach not only supports to achieve a more mature level in automotive SPICE, but it also can be applied to different domains, which is demonstrated in Table D.1. The table summarizes the potential support for generic practices (GPs) and specific practices (SPs) of the domain-independent CMMI in accordance with its defined process areas and maturity levels in its staged representation.

6.5.3 Checklist for Organizations

Beside the realization of individual practices of a specific assessment standard to prove high maturity, our approach is likewise applicable to more general organizational scenarios. However, the setup of a process line with technically annotated MCs to execute development processes requires huge initial efforts, which only make sense, if individual characteristics are given. Therefore, to enable an organization to decide whether the setup of such a detailed process knowledge base is meaningful or not, the following provides a checklist of most important indicators, for which our approach provides an organization with an added-value:

Checklist

⇒ **Defined Processes:** Most organizations already define a reference process. SPLE can directly be built on available processes to link a reference process with technical MCs and to bring processes into life. Setting up the overall process landscape from scratch, of course, can be a rather time-consuming task.

⇒ **Many processes:** The efforts for setting up a process line, does not pay off for organization with only few processes.

⇒ **Complex processes:** Simple projects which do not define a process with artifacts and guidelines in detail, are realized more easier without setting up a complex method repository. The reuse of defined MC and explicitly defined computational method engineering information, only pays off for organizations with rather complex processes, which are difficult to manage due to several information sources which must be observed during the development.

⇒ **Processes change:** For domains with frequently changing product requirements and changing compliance requirements, which influence the process model, setting up a process line reduces efforts considerably. In parallel, situational processes are in line with approved practices all the time.

⇒ **Multiple Collaborations:** When different experts are involved in the development of interrelated artifacts, introduced traceability capabilities and change impact analyses reduce inconsistencies and time-consuming artifact reviews.

⇒ **Guideline available:** If an organization already has defined guidelines and best practices, they can be moved from repositories or databases to the process model simply, i.e., automated guidance can be provided in reasonable time.

⇒ **Large-scale meta models:** Our approach mainly focuses on model-based software development processes. In particular, for a large number of complex meta models, which must be observed during the development process, the correct application of defined language elements can be ensured using our approach.

⇒ **Compliance requirements:** Many products and domains are influenced by the requirement to be compliant with a particular standard or law. The configuration of a situational process based on characteristic criteria, which cover these standards, and the execution of configured process ensures the compliance.

⇒ **Measurement and Optimization:** The usage of a workflow management system for development processes meets the requirement to monitor relevant process performance parameters. An organizational plan to optimize development processes benefits from controlling the process using our approach.

Especially, for large organizations with a huge number of defined processes, which frequently must be adapted to product-specific and regulatory needs, our approach fits well. Additionally, if the complexity of guidelines, best practices, and documentation about the correct application of meta models exceeds the capabilities of individual developers and if the development process has to be improved, our approach provides means. Beside that, the approach is independent from any domain, i.e., it is reasonably applicable to various domains, as long as some of the above criteria are given.

6.6 Case Study

As a second case study, we demonstrate the versatile applicability of our approach. Therefore, we use an example, which is different from the automotive sector. To enable domain-specific development of namely service oriented architectures, enterprise applications, and embedded systems, the three distinct process frameworks M^3SOA [MID09b], M^3EJB [MID09a], and M^3EE [MID09c] were established by a German enterprise. Using the example of these three different methods, our case study will demonstrate the establishment of a process line (M3), from which situational processes can be derived to support the situation at hand, i.e., to provide a domain-specific process, which fits project-specific needs. Beside the business-oriented design of our process line, we demonstrate the design of variants on technical level, which subsequently enables us to enact resulting process family members.

6.6.1 Setting up the process line for M3 and Situational Process Engineering

The loosely-coupled processes M^3SOA (Figure E.1), M^3EJB (Figure E.2), and M^3EE (Figure E.3) are characterized by a similar structure, which predestines them to be aggregated into one process line, from which situational processes can be configured for the situation at hand. The figures E.1 to E.3 depict the similar base structure of each single process on business level. One can see, that regardless of the application area, each process starts with an *Initiation* phase and a *System Evaluation* phase to elicit the requirements and an overall system specification. Likewise, in each process an *architecture projection* phase details the system under construction. The final phases depend on the concrete type of the system to be developed, i.e., the goal of the process. This either is the development of a service-oriented architecture, which targets the useful composition of services, the development of an enterprise software using EJB technology, or the development of an embedded system, which is characterized by hardware and software components. Independently, each process ends in a final *construction* phase, which is followed by the *implementation* of the system.

By comparing and analyzing the commonalities and differences of these three processes, we defined a reference architecture, which integrates main constituent parts on business level. Therefore, to meet the requirements of a process line, we simply merged

equal parts, such as the initiation phase or the system evaluation phase, while we combined other parts, such as the architecture projection and system deployment phase of the SOA process, or software architecture projection and software construction phases of the EE process. The resulting reference architecture is illustrated in Figure E.4. It represents the common structure for all process members of the M3 process line and provides five variation points, which can be fulfilled by situation-specific variants for SOA development, EJB development, or EE development.

Although, the process line defines the main phases of each process family member, the actual process realization depends on concrete methods, which support the development of either an SOA, an EJB application, or an embedded system best. Regarding methods, all three process frameworks follow the idea of MDA [Fra03], which proposes the stepwise refinement of models from an abstract stakeholder-specific level down to a platform-specific model. Therefore, the methods, models, and techniques to be used cause the variability in our process line. The variability is realized on technical level, where we created various variants, which enable the design on respective abstraction levels following the original method specifications. Thereby, regardless of M^3SOA, M^3EJB, or M^3EE each development phase requires three outcomes: a first artifact, which defines a level-specific system overview, a second artifact, which defines its dynamic behavior, and a third artifact, which defines the static structure of the system. During the development phases, these three artifacts are refined from one abstraction level to a more concrete level down to implementation. In accordance with this pattern, for each abstraction level, which for most of the time corresponds with a variation point in our reference architecture, we created scenario-specific process variant.

In this case study, we realized each variant as a sequential process, i.e., a complex MC, which consists of three MCs to produce relevant outcome and a fourth MC to combine and validate the results. For example, following the defined specification for EJB development during the initiation phase, Figure E.5 shows the corresponding variant. It shows the three activities to define the *Business_Context* (general overview), the *Business_Activities* (dynamic behavior), and the *Business_Entities* (static structure). Finally, the last MC of the sequence is used to release produced artifacts and to combine them into one required artifact or milestone.

Another example of the application of that pattern is given in Figure E.6. The figure shows the initiation-specific variant for supporting EE development. In contrast, the sequential process produces different output: *Initial_Requirements* defining the general overview, *System_Context* defining the static structure, and *System_States* defining the dynamic behavior.

Basically, we followed this pattern for the definition of each variant. However, we already mentioned, that some activities have to be combined to meet our process line requirements. An example is given in Figure E.7. As the reference architecture, does not distinguish between software and system architecture specification, as applied for the original embedded system process, we combined the two required abstraction levels into one variant. Therefore, the variant consists of six MCs in sequential order and it ends with seventh step to release the produced output. That way, we combine the two M^3EE-specific abstraction levels (Software Architecture Projection and Software Construction)

as follows: First we put the three predefined methods of the software architecture projection level of EE development in sequential order to define a base software architecture (general overview), the system interactions (dynamic behavior), and the decomposition of software units (static structure). Afterwards, the same variant is used for the refinement step. That means, that additional three succeeding methods are defined to produce a refined set of design software components (general overview), software states (dynamic behavior), and a detailed domain model (static structure). (Note, that in this scenario it will not affect the resulting process, whether a variant aggregates more methods, or if various methods are split into more variants to realize a connected sequence of variation points.)

After we created our reference architecture and corresponding variants, we define our configuration criteria, which is very simple for this scenario. As our process line, basically, was developed to enable either SOA development, EJB development, or embedded system development, the definitions of three hard facts is sufficient for that approach. Of course, during the course of time, various other variants would complement the process line for different scenarios. However, for now, we only have to annotate a respective hard fact (SOA, EJB,EE) with a variant. Finally, we link our variants with corresponding variation points of our reference architecture, from which we can derive a feature model, which is depicted in Figure 6.8. The feature model corresponds to our reference architecture and shows the dependencies between variants and variation point. Subsequently, a situational process can be generated automatically by binding appropriate variants to variation points considering selected hard facts.

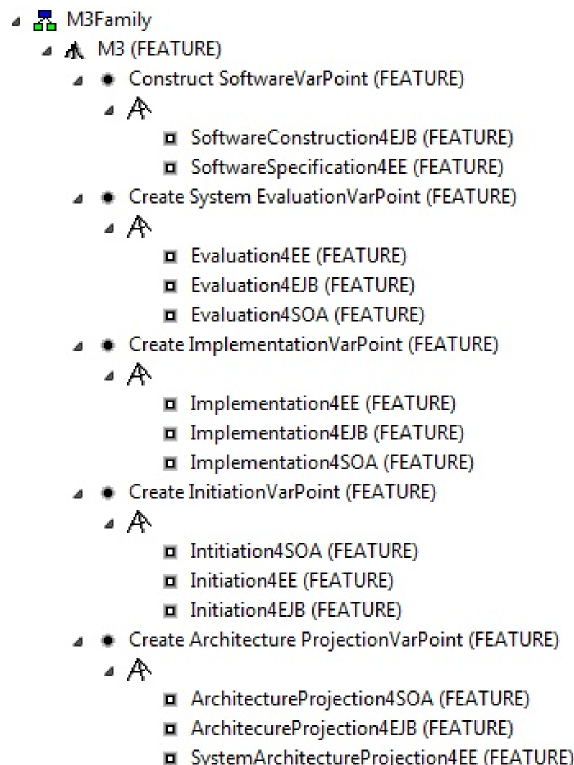


Figure 6.8: Feature Model for the M3 Process Line

Finally, we discuss adaption capabilities of our process line using the example of the SOA engineering process. The original SOA process only focuses the definition of a service-oriented architecture based on already available services. That means, that the process focuses on the orchestration and choreography of existing services more, than the development of services from scratch. To take the development of a new service into account, as well, the original method refers to the application of various technologies, such as, for example, EJB. As the SOA process ends in a *System Architecture*, which is likewise consumed by the *Software Construction* phase of the original EJB process (M^3EJB), we simply can combine the two processes by passing the result of the SOA process as input to the Software construction phase of the EJB process. To reach this, we can follow two different ways: The most simple way, is to adapt the resulting process for SOA development by adding missing EJB-specific process steps manually. That means, that we would select the EJB-specific variant for software construction from the variant repository and append it with the end of the already configured SOA process. For automation, the reference architecture could be extended by an additional variation point, likewise. By the means of an additional variation point, a respective EJB variant can bound if the set of annotated facts is adapted correspondingly. In both ways, we would be able to use the results of the SOA architecture projection phase, as input for the software construction of an EJB-based service.

6.6.2 Variant Design for M3

After we demonstrated the setup of our process line, the general design of variants, and the definition of configuration criteria, the following will focus the technical viewpoint of variants. Using the example of the Initiation phase of an SOA development, we will show a more detailed variant description, which subsequently is executed. The overall process of the variant is illustrated in Figure 6.9.

One can see, that this variant differs from our standard pattern, to provide a more detailed insight in technical design. To specify the tasks, which must be performed for SOA initiation, on a more fine-granular level, this variant is composed of ten MCs, which are performed in sequential order.
The first four MCs detail the individual steps to specify the general overview by the definition of use case diagram. The first step (CreateActors) focuses on the identification of stakeholders in form of actors. Subsequently, the system use cases have to be identified (CreateUCs), before stakeholders are assigned to use cases in which they are interested (RelateActorsWithUCs). Finally, inclusion and extension relationships have to be defined between individual use cases (DefineUsecaseRelationships).
The following four MCs detail the the definition of the dynamic behavior. Based on the previously defined use cases, activities are created to detail their behavior in the following (DefineActivities). Next, each activity is refined by relevant actions (DefineActions) and edges (DefineEdges). Finally, the control-flow of an activity is defined by relating actions via edges (DefineControlFlow).
The next MC details the definition of the static system structure in form of a class diagram

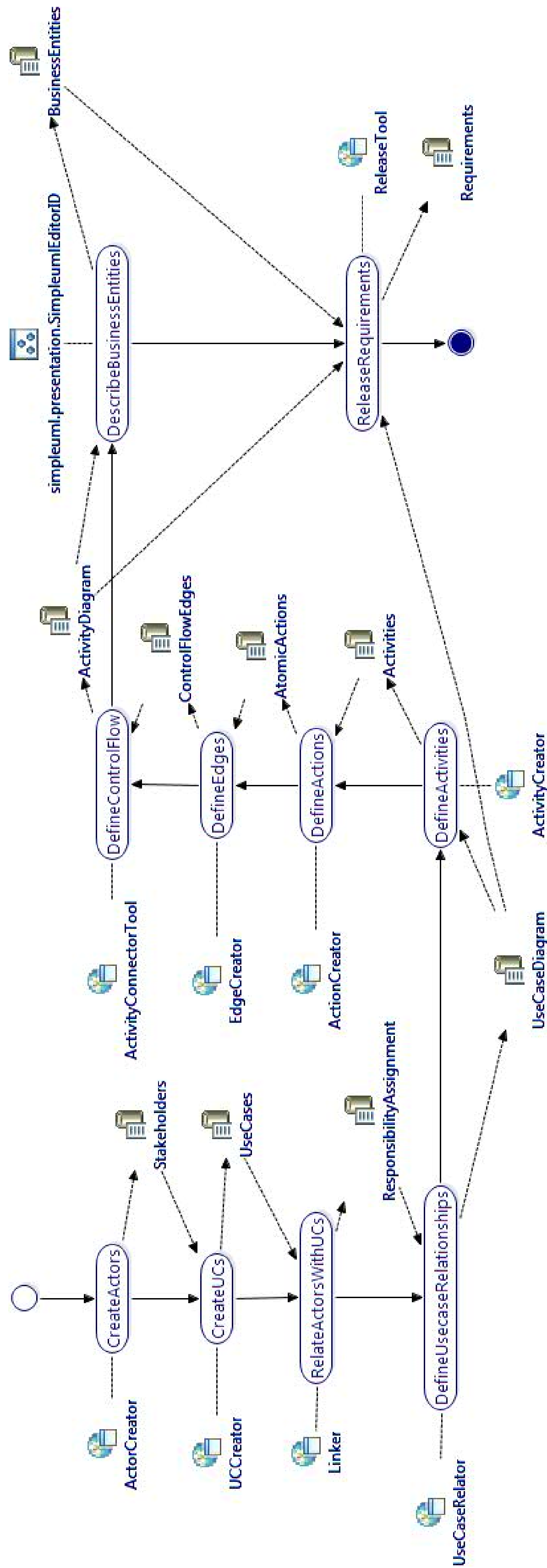


Figure 6.9: SOA-specific Variant: Initiation Phase

(DescribeBusinessEntities), before the general requirements artifact is validated and released for subsequent phase in the SOA development process (ReleaseRequirements).

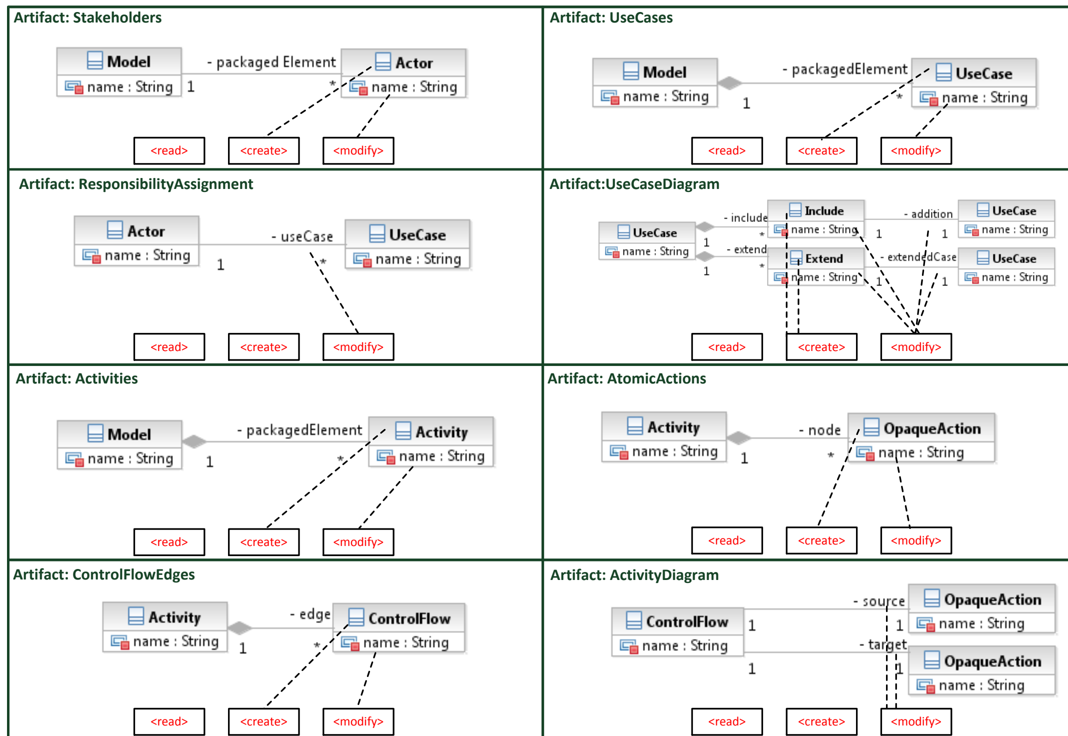


Figure 6.10: Artifact-specific Meta Model Views for SOA Initiation

Next, for each artifact of an MC, a corresponding MMV with annotated EUSA values is created, as illustrated in Figure 6.10 and Figure 6.11. For the SOA initiation phase, we decided to use the standard UML2.1 meta model for MMV definition. Therefore, for each artifact, which has to be created during the first eight MCs, Figure 6.10 depicts a corresponding MMV, i.e., an extract of the the UML meta model, which is relevant for a respective MC's output. The figure illustrates the eight artifact-specific MMVs and respective EUSA values (indicated by the dashed line). (Note, that the read-only EUSA is not indicated, as each element is read-only by default). For example, the artifact in the upper left part (Stakeholders), uses UML's *Model* element to identify elements of type *Actor* via the *packagedElement* association. By the means of EUSA values, we defined, that elements of type *Actor* can newly be created and the name of an actor can be modified. Any other elements or features are defined as read-only.
The definition of the other MMVs is realized equally. However, a particular case is given by the final MC ("ReleaseRequirements"), which not only releases and validates individual artifacts (general overview, dynamic behavior, and static structure), but it also aggregates the three artifacts into one artifact ("Requirements"), which is input to the subsequent design phase. To achieve, that all elements, which originally belong to the input artifacts, also belong to the output artifact "Requirements", *FromTo* relationships between elements of the output artifact and its input artifacts are defined, as discussed

in Section 4.4.3.

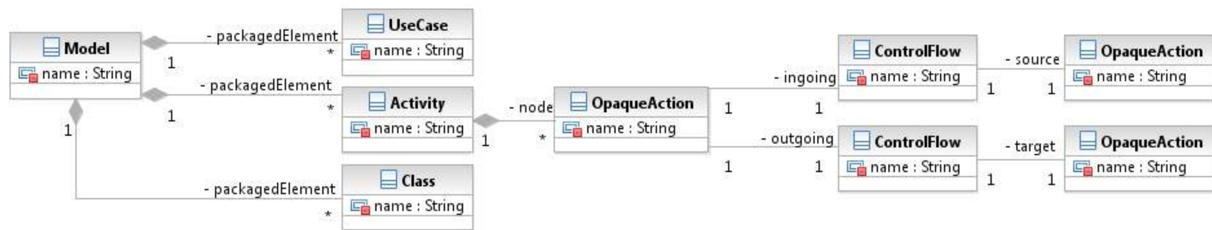


Figure 6.11: MMV for Requirements Artifact created during the SOA Initiation phase

Finally, method-specific guidelines were defined to ensure particular naming conventions, as demonstrated in Section 4.8. By the means of the graphical guideline notation, different statements in the form of the graphical statement notation or the textual syntax of OCL, can be combined and simply adapted when the need arises.

6.6.3 M3 Enactment

Using the above example of a process model on technical level, the following demonstrates the enactment of a situational process for SOA development. In our prototypical environment, we demonstrated the enactment of the overall M3 process family. However, this would go beyond the scope of this thesis.
Figure E.8 to Figure E.11 illustrate the results, which are generated by our editor generation part. The figures show the various generated editors, which were derived from the method-specific MMVs, providing the capabilities as defined by the means of the EUSA annotation. However, the figures do not only show the order of editors, in which they become active to enable the individual methods of the process, but also the integration of third-party editors. In the process model, which is depicted in Figure 6.9, we referenced a third-party tool explicitly to enable the MC *DescribeBusinessEntities*. Thereby, we relinquished the definition of an MMV. Instead, we provided our framework with an editor identifier ("simpleuml.presentation.SimpleumlEditorID"), which enables our framework to run an external editor using standard services of the eclipse platform. The external editor, which we required to be used for class diagram design, is a simple UML editor, as depicted in the upper part of Figure E.11.

The final method of the initiation phase has to validate and release produced artifacts. Therefore, the generated editor (lower part of Figure E.11) enables to view and modify most relevant elements (use cases, actions, and classes) of respective artifacts.

During the execution of the methods, the creation and modification of model elements is notified by the editors and stored in the history of respective artifacts, as discussed in Section 5.4. The following scenario, will demonstrate the identification of potential artifact inconsistencies and subsequent process coordination activities: during the second

MC (*CreateUCs*) (cf. Figure 6.9), we created the four use case elements "SellProduct", "AcceptOrder", "TransactOrder", and "DeliverProduct". Therefore, in the context of that MC, four notification events were stored in the history of the MC's output artifact "UseCases" to indicate, that the four use cases belong to that artifact. Now, we assume, that one or more uses cases would be modified (e.g., by changing their "name" feature) in the context of the final MC "ReleaseRequirements". A notification about such an event, would be assigned to the history of the output artifact "Requirements". After the workflow of our case study process is completed, the inconsistency identification strategies, as discussed in Section 5.6.4 would be processed. In this case, scenario 3 would reveal a potential conflict depending on the artifact "UseCases". Based on this information, the workflow management would restart the workflow to check for consistency and to validate all guidelines, which were defined for *UseCase* elements in the affected MC, i.e., "CreateUCs".

7 Related Work

Previous work already exists to enable continuous tool chains, as well as, flexible and dynamic process execution. In EDONA [OT08], an actual research project of the pole of competitiveness System@tic Paris-Area, the objective is the construction of an open platform facilitating the realization of chains of development by providing an interoperability and interchange architecture for automotive development processes and tools. EDONA's principle of the integration platform is to provide access to a common storage space accessible by any tool chain. Therefore, its goal is to provide a common meta model to define the data exchanged and integrated between the partners, a common technical architecture based on the Eclipse Equinox platform, and a set of more generic tools and tool inter-operation bridges.

In Aldazabal et al. [ABN+08], the authors suggest a service oriented middleware, called ModelBus, connecting model-based development tools and the services they offer. Thereby, process enactment and process orchestration tools can be used to create/orchestrate/monitor composite services by combining the different services from the different tools into a workflow described in a language, such as BPMN [BPM09]. Similar to EDONA, the focus is on model exchange and not on method engineering or enactment.

The SHAPE project [SHA10] investigates the development and realization of enterprise systems with ideas of model-driven engineering. As proposed by the MDA concept, it separates the modeling into the three abstraction levels Computation Independent Model (CIM), Platform Independent Model (PIM) and Platform Specific Model (PSM) and tries to fill the gap between them with model transformation. From this approach we borrow the idea of distinguishing the business domain from the IT domain.

Burmeister et al. follow an approach of applying multi-agent based technologies, namely BDI-agents, for business processes modeling and execution [BACR08]. Mainly, the usage of agent technology with its ability of flexibility and pro-activity shall provide agile behavior of the entire business process management system, whereas the "process plan" is described in terms of project goals, subgoals, and associated plans, which shall achieve the respective goals. We adapt the notion of business process modeling in the area of method engineering and enactment.

Contrasting the above strategies, in this thesis, we created an MDE based CAME environment to support method engineering, process enactment and execution capabilities, which enable flexible guidance for situational development processes. As CAME environments basically consist of the two parts (CAME part and CASE part) (cf. [NR08]) and

since automated tailoring is neglected by most of the existing CAME approaches, the following discussion distinguishes relevant approaches, which are related to these two parts:
Since SPLE, as described in chapter 3, is an essential part of our complete CAME environment, we first describe work, which is related to our automated process tailoring approach (Section 7.1). Subsequently, we discuss various complete CAME environments, as well as, some approaches, which focus particular parts of an CAME environment (Section 7.2). Finally, we summarize and compare most relevant approaches in Section 7.3.

7.1 Related Work on Method Engineering

As an one-size fits-all approach does not work for software development processes [Fir04], a multitude of strategies was developed to face the design and tailoring of development processes. Beside the idea of software process lines, as introduced by Rombach [Rom06] and realized as a configuration-based approach in chapter 3, various approaches have been proposed in the literature. Existing method engineering approaches, such as the assembly-based, the extension-based, and the paradigm-based approach (cf. [RDR03]) are distinguished in literature [HSR10, ABO11, GD12] from other approaches, such as configuration-based approaches [BE96,BPKJ07], instantiation-based approaches [KSP+09], generic-based approaches [RP96b,GP01], architecture-based approaches [MHAK08,PG08], and recovery tailoring approaches [HB01, Xu05, PNPS06]. Among existing approaches, configuration-based approaches focus on creating a target method by adding/removing elements from the base methods. In the following, we focus on the comparison of our approach with other configuration-based approaches.

Henninger et al. [HB01] adopt a case-based approach for tailoring the software process, where configuration (i.e., tailoring) is performed by specialization through an incremental set of antecedently tailored processes. Xu et al. [Xu05] define software process configuration as a knowledge-intensive activity, and analyze the benefits of knowledge management in this task. In this context, we encapsulate the knowledge by the means of feature models specifying the configuration space for situational methods.

Karlsson et al. [Kar04] describe a method to enable method configuration (i.e., method tailoring) using reusable configurations of a base method suitable for a particular characteristic of a development situation. The proposed method introduces Configuration Packages and Configuration Templates that predefine a combination of development tracks to facilitate reuse across commonly occurring situations. The process is preformed manually and the authors did not consider situational characteristics in the development process. In [WK04], Wistrand and Karlsson discuss the use of method components in a general way as building blocks for method engineering. The method engineer can use interfaces which are available for each method component to construct and configure situational methods.

SPEM [OMG08a] is a prominent member of modern process definition languages, which provides means for variability modeling and an assembly-based configuration of engineering processes. SPEM's meta model defines capabilities to manage libraries of

method content, i.e., different types of MFs and assembled MCs, and processes. Variability is added by the capabilities of variation and extension of particular SPEM elements to customize variability elements without directly modifying their original structures. Instead, variability enables a method engineer to describe the differences (additions, changes, omissions) relative to the original. The approach mainly encompasses structural variability without considering situational characteristics or automated assembly support.

In [BBG+05a], they present the results of the PESOA project, which developed a methodological foundation for process family engineering considering the variable behavior of application software in the e-business and automotive domain, following well-known product line engineering principles. They introduce a conceptual model to define variant-rich processes and a methodology called the PESOA process for developing, using, and maintaining families of processes. The methodology mainly is guided by the two parallel phases domain engineering and application engineering, as likewise applied in our approach and widely-accepted in product line engineering.

Washizaki [Was06] proposes another technique for establishing process lines by extending SPEM to express the commonality and variability of the process using UML activity diagrams. The approach is based on building a process line architecture or reference process, which is configurable via variants. The reference process is composed of a set of interrelated (variable) activities and conditional branches, which are resolved using project characteristics represented in, e.g., a feature model. New project-specific processes are enabled by selecting appropriate features, i.e., project characteristics, which are associated with an corresponding variant in the reference process.

In [GP07], Gonzalez-Perez describes how the major advantages of ISO/IEC International Standard 24744 can be applied for the implementation of method engineering solutions. Ahroni et al. [ARB08] proposed a holistic approach by enriching the ISO/IEC 24744 with the aim of enabling situational method engineering approaches to support method component representation, and tailoring them for situational methodologies. The presented approach enables the specification of both structural (product-related) and behavioral (process-related) aspects as well as procedural relationships between them. The proposed methodology layer defines the tailoring information properties, that are used in the endeavor layer to support the definition of situations to which a particular method component suites and the tailoring by the means of a stage concept.

The authors of [SCO07] first characterize a process family and propose a formal approach based on the Little-JIL process definition language. In their approach they distinguish 3 process definition concerns, namely the process steps, the performing roles, and the produced and consumed artifacts. Based on these concerns, they suggest the definition of a fixed process on which varying augmentations with elaborative steps are made to tailor the process. To generate process instances, additionally, they identified different process instance generating techniques.

Alegría et al. [ABO11] studied modeling process variability for further tailoring software process lines. Feature models are used to represent the variability within process models implemented using SPEM 2.0, whereas situational processes are resolved using

a context model and a particular transformation rule set. The authors discuss an MDE-based tailoring strategy, which helps to achieve a separation between the process modeling and process enactment, and reduce the complexity by reusing tailoring knowledge intensively. It is shown that adopting software product line engineering has potential to improve the project's productivity and quality, as well as the resulting software products.

Recently Method-oriented Architecture (MOA) [DIKS09, Rol09] was proposed, which empowered the assembly-based ME principles with a standard for describing methods as services in parallel with service discovery principles for finding distributed method components. Our approach also utilizes MOA principles to describe and discover methods corresponding to situational needs of a reference process. Furthermore, in [AMGB11], they introduced the concept of Families of Method-Oriented Architectures for addressing two major challenges of assembly-based method engineering for publication and sharing method components as well as management of variability within software methods. To this end, they proposed the idea of leveraging Service-Oriented Architecture (SOA) and SPLE principles for dealing with these challenges.

Alexeio et al. [AFSK11] describe a model-driven approach, which uses techniques from software product lines, to enable automatic variability management for software processes and their subsequent execution. Similar to our approach, they use the feature model notation in order to explicitly indicate variability within software processes modeled in UMA (an evolution of SPEM). Resulting feature models are configured using a product derivation tool, which resolves constraints between features to create the situational processes. Finally, tailored process models are transformed via model to text transformation into interpretable language. Unlike our approach, Alexeio et al. only focus process variabilities and the automated allocation of individual process activities. They do not consider the design on different levels of detail (business vs. technical), PSEE capabilities, or situational characteristics of particular methods.

In [JM05], a domain-specific process line is proposed for process tailoring and subsequent process refinement. In their Emergent Process Acquisition method, they discuss relevant steps for setting up and tailoring a process line.

In [MPRC08], they discuss an alternative interpretation of the feature model semantics in order to derive business processes more efficiently. They extend Process Family Engineering from [SP06] to automate the derivation of business processes from a process line, where variability is represented using a feature model. Therefore, they define a mapping between feature models and process models, which can be applied automatically to create the basic structure of a business process based on a feature model configuration.

In [CATP11], Cervera et al. introduce the basic idea of the *MOSKitt* project, which is, to the best of our knowledge, the only approach, which faces all the required components of an CAME environment, i.e., method design and configuration, as well as the CASE generation, at the same time. While the method design phase in *MOSKitt* is similar to the creation of our reference process on business level, method configuration is different. While in our approach configuration means to automatically link situational MCs on technical level with variation points of the RP, the configuration in *MOSKitt* means to link a generic process with available technical information, such as meta models, transforma-

tions, or editors. This enables the generation of CASE-specific parts. Hereby, Cervera et al. overcome the complexity of the CAME environment development by the means model-driven techniques, i.e., they use models and model transformation similar to our approach. However, while a process model is instantiated without considering variabilities explicitly to guide the development process, the main benefit of their model transformation is a static binding between process activities and already existing technical assets, such as meta models and editors.

In [AB12], Alegria and Bastarrica define a meta-process to produce a project-specific software process model in a planned way. The meta-process, which is called Context Adaptable Software Process EngineeRing (CASPER), is subdivided into the two main phases: domain engineering and application engineering. During the domain engineering phase, the context of a process line is defined. Afterwards, the variabilities and commonalities within the process line are identified and defined using a feature modeling approach, where entities or features are characterized by their application scenario. A process line architecture organizes a reference process for identified features and deals with variation points and variants as defined in the feature model. Finally, a model to model transformation resolves variability according to project-specific characteristics and a predefined transformation rule set.

7.2 Related Work on Computer-aided Method Engineering Environments

As initially discussed in Section 2.4.2, an CAME environment is composed of two parts: the basic CAME part, which supports ME activities, and an CASE part, which supports decision making and workflow management at process runtime. Unfortunately, the CAME approaches which came up during the past decades, do not focus CAME in all of its facets, as discussed e.g., in [NR08]. This drawback also was analyzed in various reviewing papers, which were published since the first CAME environments came up [GLB+86,ACF97,ADOV02,Gru02,NR08,MR12]. Most of them either focus the ME related part, the CASE part, or individual sub-parts. Following this distinction and due to the difficulty to contrast approaches with different orientations, we split our discussion of related work into these two main parts of an CAME environment, as well. Therefore, after we discussed various work, which is related in particular to ME and the assembling of situational methods from a method repository in the last section, we now focus on work, which is related to the CASE part of an CAME environment and split into the design and execution of an PSEE component and an CASE generator component (cf. [NR08]).

We start by giving a general overview about work, which supports individual parts of the CASE part of an CAME environment. Subsequently, we describe various approaches, which focus on the PSEE, before we describe individual approaches of holistic CAME environments.

7.2.1 Overview about Computer-aided Method Engineering constituent Parts

Beside ME activities, a holistic CAME environment, amongst others, should enable the design and execution of the following tasks: workflow management or control of the development process, generation of CASE tools, provisioning of guidelines or constraint evaluation, and traceability across the overall process model outcomes. For all of these tasks individual approaches, which though, are unrelated for most of the times, do exist.

Already in 1995, Kraut et al. [KS95] discussed characteristics of coordination in software development, such as scalability, interdependence, or uncertainty. To overcome these and other process-specific challenges, various evolutions of PDLs, such as APEL [DEA], JIL [Jaz97], UML4SPM [BGB06], or SPEM [OMG08a], were proposed and multiply discussed in literature (cf. [ZL01, BJF09]). While all these languages exclusively focus either the design or the enactment of development processes, they lack other CAME needs, such as the generation of CASE tools. To face this need of creating customized and method-specific user interfaces or editors, various other unrelated approaches were introduced during the last years, as well (cf. [ES97, Dav03, CCT+03, Van05, SMV07, Gro09, YLZX10]). However, they are completely detached from other relevant capabilities. This also holds for most of the known guideline or constraints formalisms, which were developed to ensure individual characteristics, qualities, or facts of process artifacts on various levels of abstraction. Indeed, in the area of computer-interpretable clinical guidelines (cf. [CKH08, PTB+03]), they intend the development of a guideline formalism. This approach, though, more conforms to a process definition, which guides clinical staff in required activities efficiently, similar to most PDLs as discussed before. Other approaches, such as OCL [OMG06a] and *Visual OCL* [KTW02], or similar academic efforts, such as [BJ06], [ALSS08], or [GRE10] particularly focus on the specification and the automated evaluation of constraints on design documents. Likewise, various approaches were developed to follow tracing dependencies, i.e., to enable traceability (cf. [CHCC03, ARNRSG06, GG07, BMMM08]).
All these isolated applications, though, do not fulfill the needs and requirements of an CAME environment, as they do only focus one concern, i.e., they are not integrated with each other. This disables them to support development processes efficiently for the situation at hand and brings us to the need of a Process-Centered Software Engineering Environments, a prestage or at least essential part of an CAME environment.

7.2.2 Approaches for Process-Centered Software Engineering Environments

As discussed in Section 2.4.2, an PSEE is an environment, which provides manifold support for software developers by the enactment of process models. Such environments include various services, which, amongst others, concern tasks, as discussed in the last section. Contrasting CAME environments in general, they neglect ME and/or CASE generation facilities, while they particularly focus the enactment of process models. A first generation of PSEE prototypes, where a process model is interpreted at run-

time to manage interaction and enforce consistency between documents, is given by, e.g., *Melmac* [Gru90], *Merlin* [PS92], *ADELE* [BEM93], *ALF* [DG94], *Serendipity* [GH98],or *SPADE* [Wes99]. In parallel, other approaches with different qualities, features and contexts, such as social interaction ([AAO94]), scalability ([BSK98]), analysis ([DG98]), method guidance [PWD+99], process evolution ([Cun00]), or the incorporation of domain knowledge ([DZR+04]), have been proposed in the field of PSEEs.

Although, many efforts have been spent to create an PSEE, which is accepted in various domains, especially, in industry, none of the mentioned approaches has reached a meaningful level. This is not only due to the high complexity and difficult usability of respective approaches, but also due to the restriction of a developer's creativity, which particularly is important in the area of software development. Therefore, lessons learned lead to newer approaches with trimmed functionality in order to focus today's needs more. In the following, we describe individual approaches, which came up during the recent years:

In [BGB05], Bendraou et al. introduced the *UML4SPM* meta model for software process modeling, which extends a subset of UML2.0 concepts without influencing the standard. Due to the restricted expressiveness of the former version of SPEM 1.0, they proposed an executable action semantics within activities using the expressiveness of the well-known and widely-accepted *UML2.0*. Based on UML4SPM and the executable metaprogramming language Kermeta [MFJ05], the authors introduced a framework to enable process modeling, simulation and execution. Providing an adapted version of *UML2.0* activity diagrams with an execution semantics using Kermeta enables the design and enactment of software development processes in terms of an PSEE. The great advantage of using *UML4SPM*, though, is to use the well-known standard *UML2.0*, which most communities are familiar with. That is also the reason, why we used the mature version of SPEM 2.0 for process modeling on business level.

The *Transforms* environment [MSMR09] was proposed by Maciel et al. to address an integrated approach for process modeling and enactment based on specializations of individual SPEM 2.0 concepts. They focus the design and the enactment of an MDA process, as well as, the execution of model transformations. Therefore, they extend SPEM's work product definition by four specializations: an *UMLModel*, which is produced or generated during the process execution; a *TransformationRule*, which contains the transformation rules to automatically process models during the process execution; an *ExtraModel*, which is used for documentation and additional information; and a *Profile*, which bases the modeling on each phase. By relating the specialized work product types with tasks, which are organized in a work breakdown structure, they are interpretable for a process-conform application. Similar to our approach, they focus on an MDA development process by detailing work products or artifacts of the development process with meta model or profile information.

Weber et al. propose a *semantic process- and artifact-oriented collaboration environment (SPACE)* to tailor and follow organization-specific process models in [WEB+09]. In the

wiki-based platform, collaborative creation of processes and artifacts is enabled through, e.g., visual templates and editors. Additional semantic annotations enable consistency checks, traceability between various process fragments to provide insights to the effects of individual changes, and personalized views to enable stakeholder-specific perspectives on information. Beside the process model, which specifies the default flow of activities, an artifact model, which comprises the internal structure of artifacts, i.e., data or relationships to other artifacts, is associated with the process model. Base on the artifact model, concrete templates are generated, that actively support developers in the definition of artifact instances following previously defined data structures.

The *ADvanced Artefact Management System (ADAMS)* [DFOT10] is a web based system, that integrates project modeling, resource allocation, and scheduling capabilities with artifact management features, such as configuration management, traceability, and artifact quality management. In contrast to traditional PSEEs, it is conceived for the definition and cooperative production of software artifacts, which is achieved in *ADAMS* by a fine-grained management of artifacts. A project management subsystem enables the definition of a schedule and allocates human resources to the artifacts. By the means of checklists, which are associated with artifacts, guideline functionality is supported as quality management capability. Contrasting our artifact design using MMV to define the concrete data structure, *ADAMS* provides composition links between abstract artifact descriptions and each of the contained sub-artifacts to define fine-grained hierarchy of artifacts.

In [ABN+08], the authors describe a service-oriented middleware called *ModelBus*, which is another idea to enable automated model-driven development processes. *ModelBus* is an environment, which connects services and allows clients to invoke those services. In collaboration with particular enactment tools, composite services can be created, orchestrated, and monitored within a single workflow. A workflow orchestration is realized by a transformation of SPEM-based models to the Business Process Execution Language (BPEL), as described in [SA09]. Using the *ModelBus* and the associated workflow orchestration tooling enables the modeling of a process, which subsequently can be executed in a distributed environment to use different modeling services (automated or human-based) according to a defined workflow. In parallel, the *ModelBus* makes models and meta models transparent to the individual modeling services.

From 2008 to 2011 the European research project *MOST - Marrying Ontology and Software Technology* [MOS11] developed an ontology-aware modeling environment, which provides means for semantic model validation, semantically-rich tracing, and software process guidance. Combining the meta modeling technical space with the ontology technical space (cf. [KBA02a]), it enables to compute next steps in a particular software development process. Therefore the guidance engine must be configured by a so-called SW-Process Guidance Ontology, which organizes knowledge about tasks, their preconditions and postconditions and how they are related to the modeling artifacts. Based on the actual process state, i.e., already defined model-based artifacts, accomplished tasks, and

the currently logged user/role, the guidance engine is enabled to compute open tasks, which are displayed to the user. The environment distinguishes between design time to model processes in the ontology and meta modeling technical spaces, and enactment, which guides the engineering process based on reasoning at runtime.

7.2.3 Approaches for Computer-aided Method Engineering Environments

As shown in the last section, PSEEs only address sub-parts of an CAME environment and do not provide a complete set of features, which support engineers in planning, design, enactment, and automation of development processes. Therefore, CAME environments complement PSEE with missing capabilities, as discussed in Section 2.4.2. One of the earliest representative of an CAME environment was the *Methodology Representation Tool (MERET)* [HO92]. *MERET* is a product-oriented approach, where methodological knowledge is specified and tailored using a semantic data-model called ASDM. Although, *MERET* is understood as CAME environment, it misses process enactment and CASE generation support. In parallel with *MERET*, a first academic prototype, called *MethodBase* [Har97b], was introduced. In *MethodBase*, they focus on the customization of complete methods, which enable process enactment and CASE generation, in contrast to e.g., *MERET*. Likewise, *Decamerone* [HB95] is another environment of the early days, which provides CAME and CASE capabilities in parallel. It consists of a repository containing method fragments and assembled methods, which are specified by the means of the *Method Engineering Language (MEL)* [BSH01]. Beside CASE generation for the enactment, *Decamerone* also considers means for configuration management and project scheduling. All of these earlier approaches, are rarely applied outside academia, but base the development of most recent CAME environments. As until today, to the best of our knowledge, almost no commercial tool does exist, which provides full capabilities, while different groups consistently attempt to provide new environments, this shows the reasonable importance of CAME. The most meaningful approaches, which came up during the last years are described in the following.

By restricting their focus on the requirements engineering domain, Si-Said et al. introduced a *Computer Aided Requirements Engineering (CARE)* environment named *MENTOR* [SSRG96]. Its core component is a guidance engine, which guides the design and application of process models for existing requirements engineering methodologies. *MENTOR* basically consists of four components: a method engineering environment for supporting method engineers in the modeling of product and process fragments of a method, an application engineering environment to support the enactment, the guidance engine, which guides method and application engineering in their work likewise, and a repository to organize and store methods. Enactment is based on a set of so-called ways-of-working, which is a tree representing goal-specific process knowledge using the contextual paradigm, which was developed as part of the NATURE process theory [JPJ+93]. That way, given the context of a product under development, the guidance system supports decision making during the creation of products, which are based on ER [Che76]

and static OMT [RBP+90] paradigms.

MERU is another CAME environment, which is described in [GP01]. The *Method Engineering Using Rules (MERU)* tool is based on a product model driven Method Requirements Specification (MRS), which subsequently is used for a technical instantiation. Similar to our distinction between the business level and the technical level, an implementation-independent MRS only describes the nature of a method and provides an additional abstraction, which is translated into an instantiation of a technical meta model called MVM, automatically. From this instantiation, the method is generated, and subsequently given as input to their CASE tool environment called *RAPID* [PS96]. The MRS in *MERU* is based on a meta model called MVM, which provides a generic view on *product elements*, *links* to connect product elements, and *constraints* to specify properties on links and product elements. Unlike other CAME environments, *MERU* does not propose the definition of a process model or method components, which must be combined by an method engineer manually. Instead, only the requirements of a method have to be specified using the generic Method Requirements Specification Language (MRSL). This exclusively bases the generation of a plan of instantiation, from which method fragments are generated automatically by matching predefined rules to get appropriate method fragments from the method base.

In 2003, Saeki [SAE03] introduced an CAME environment, which combines an assembly-based method engineering approach with an editor for specifying meta models and a generator for diagram editors. Similar to our approach, they propose an easy to learn, powerful language to describe the product and the process part of a method in a computer-interpretable way. While product descriptions specify the structure and the data types of a work product, process descriptions are used as guidelines for navigating developers through their activities. Contrasting our approach, where the MMV mechanism restricts existing meta models to needs of outputs and inputs of an MC, Saeki proposes the usage of class diagrams for the specification of activities' output data structure from scratch. For a product definition, OCL constraints can be defined to enable an artifact-centric validation mechanism, which is similar to the MC-specific guidelines in our approach. Similar to our approach, the result of evaluation of a specific constraint does influence the continuation of the overall process. For modeling the process, activity diagrams are used to guide the application of generated, method-specific editors, as well as, situational constraint evaluation. However, contrasting our approach, they neglect variability management as well as the individual facilities, such as consistency management, traceability, or monitoring. Recently, they enhanced their method repository component by version control and change management [Sae06].

MetaEdit+ [Tol06] was introduced in 1994. It is an CAME environment, which provides a meta modeling language to generate CASE support from high level models, and a tool suite for defining the method concepts, their properties, associated rules, symbols, checking reports, and generators. For meta modeling, *MetaEdit+* uses the *GOPPRR* (Graph-Object-Property-Port-Role-Relationship) language [TK08] as conceptual data model,

which specifies the product part of a method on meta level. For these product parts, rules and guidelines can be defined to guide the correct usage of manifold editor types, which are generated from the models defined in *GOPPRR*. *MetaEdit+* provides a couple of Method Management Tools [CMV96] to enable CAME capabilities, such as a Method Base, which organizes predefined method fragments and symbols needed for their representation, a Method Assembly System to edit and validate methods, and a environment generation system for delivering CASE tools. *MetaEdit+* is a product-oriented environment, which does not provide explicit enactment or process-related capabilities. Nevertheless, due to its simple application and the powerful design, validation, and reporting facilities, it is the only tool, from which a commercial version does exist until today.

In Section 7.1, we already mentioned the method engineering capabilities of the Eclipse-based modeling platform named MOSKitt [CATP11], which is also a complete CAME environment, since is enables the automated generation of CASE tools for a method's product part and the execution of a method's process part. A methodological framework, which consists of three phases is provided to realize this: During the method design phase an implementation-independent method specification is build using the SPEM standard, similar to our reference architecture on business level. During the method configuration phase, the resulting model is instantiated by linking the product and the process part of a method with specific technology-specific assets, that will be used for enacting the methods. That means, for example, that artifacts are linked with predefined meta models, and tasks are associated with existing editors or transformations. Finally, tool support is generated, during the method implementation phase. In this phase a process engine enables the execution of the process part modeled in SPEM, whereby editors, which were linked with individual tasks during the previous phase, are used for the creation and modification of artifacts. Contrasting our approach, where enactment is supported by an automated generation of editors and validation mechanisms, only predefined assets can be re-used and linked with the process model.

Beside process modeling, tool management, data management, and process execution, Polgar et al. particularly focus the certification of processes, in [PRSH09]. Similar to MOSKitt and our approach, process modeling is realized on two levels: the platform-independent level, which specifies the basic fragments of the methods, and the platform-specific level, which links the fragments with available technical assets. The platform-specific model, subsequently, is translated into computer-interpretable workflow model and a storage model, which contains information to connect tools and artifacts. For process certification, criteria, which are required by some standard, are formalized first. By linking the criteria with a representation of the process model in the ontology technical space, standard conformance can be checked using a reasoning tool.

7.3 Discussion and Comparison of the Approach with Related Work

The last sections have shown, that many related work on ME, PSEE, and CAME was developed during the last years. In the following, we discuss all these approaches in comparison with our approach namely Situational Method Engineering for Process-Centric Languages (SME4PCL). Due to the different focuses of the manifold approaches introduced above, we decided to follow these focuses in our comparison. Therefore, to evaluate the qualification of SME4PCL with regard to its ME facilities, we first compare the ME capabilities of our approach and other comprehensive CAME environments with capabilities of described ME approaches. Secondly, to evaluate the qualification of SME4PCL with regard to its PSEE facilities, we compare our approach and CAME environments with capabilities of described PSEE approaches. Finally, we focus the CAME environments themselves and evaluate our approach with regard to its general applicability as an CAME environment.
For each comparison, we discuss the underlying criteria and capabilities, which are partly inspired by categories from [NR08]. We use them for the comparison in a summarizing table to discuss strengths and weaknesses of our approach. While some criteria were answered by ✓(meaning, that the approach provides means to support this capability) or ✗(meaning, that the approach provides *no* means to support this capability), other criteria are answered by a concrete technique to support an individual feature or capability.

7.3.1 Comparison of Method Engineering Approaches

Table 7.1 illustrates a comparison of ME approaches (cf. Section 7.1), comprehensive CAME environments (cf. Section 7.2.3), and SME4PCL. The used criteria are described in the following:

1. Method Requirements Analysis (MRA): It is a generic phase of an ME process, which focuses on the identification of important features of the method which should be developed. During the MRA phase, the requirements, which a method should fulfill, should be defined in a formal way.

2. Method Design (MD): It is a generic phase of an ME process, which focuses on the specification of a general architecture of methods or processes, based on the requirements defined during the previous phase.

3. Method Implementation (MI): It is a generic phase of an ME process, which focuses on the assembly of a situational process from suitable methods or method fragments. The result of this phase is a set of products, which enable the enactment of the process to guide a developer.

4. Method Test (MT): It is a generic phase of an ME process, which focuses on the verification and/or validation of a newly developed process or method. Various characteristics, such as the conformance with requirements of the MRA phase, correctness, or feasibility can be tested during this phase.

5. SME Approach: As introduced in Section 2.4, there are several alternative ways to produce a method. This criteria is used to compare approaches with regard to their method building process.
6. Multi-Level Architecture (MLA): Methods and processes can be described on different levels of detail to fulfill the specific needs of individual stakeholders. Therefore, this criteria is used to indicate, whether or not an approach enables a design on different abstraction levels.
7. Automated Component Configuration (AutoConf): This criteria states, whether methods/processes must be configured by a method engineer manually or if an automated support for the configuration of situational methods and/or processes is available. Due to the growing number of methods, which are developed over time, we mean, that this must be a major criteria for future applications.
8. Situation Matching: To address the situational facet of a method, methods must be identifiable with regard to their situational applicability. Therefore, this criteria states, whether or not situational needs are considered by the approach. We refer to the respective mechanism if situation matching is supported.
9. Variability Handling: To prevent method engineers from reinventing the wheel, reuse and adaption of available methods is important. To reach this, variabilities and commonalities between methods and processes must be managed. We refer to this capability as variability handling and indicate, whether or not this feature is supported.

All the approaches, which are illustrated in Table 7.1, stress the significant importance of a sound decision making processes leading to situated methods. However, there are hardly ME approaches, which address and facilitate decision-making to achieve an automated or semi-automated configuration of situational methods following the required characteristics of a situation. Hence, our approach aims at providing automated decision-making for deriving effectively situated methods from process lines based on characteristics of a project. Moreover, one strong point of our approach with respect to other approaches is considering project characteristics as decision criteria and annotating process line assets with these characteristics. Hence, every situational process, which is created using our technique, ensures the satisfaction of hard facts and the optimization of soft-facts and concerns. Unlike traditional methods, feature models are generated in our approach and provide a basis for subsequent steps for building situational processes. Therefore, we introduced feature models in the role of an intermediate helper in our approach.

7.3.2 Comparison of Process-Centered Software Engineering Environments Approaches

While some of the presented approaches only do focus the ME part of an CAME environment, other approaches do only address process modeling and the subsequent enactment of the modeled process, i.e., they only represent the process support of CAME. Therefore,

Approach	MRA	MD	MI	MT	SME Approach	MLA	AutoConf	Situation Matching	Variability Handling
[Was06]	✓	✓	✓	✗	architecture based	✗	✓	requirement feature model	✓
[SCO07]	✗	✓	✓	✗	architecture based	✗	✗	✗	✓
[ABO11]	✗	✓	✗	✗	Rules	✗	✓	Context Model	✓
[AMGB11]	✓	✓	✓	✗	MOA	✓	✓	feature annotation	✓
[AFSK11]	✗	✓	✓	✗	architecture based	✓	✓	✗	✓
[JM05]	✓	✓	✗	✗	architecture based, assembly-based	✓	✗	✗	✓
[MPRC08]	✓	✓	✓	✗	assembly-based	✗	✓	✗	✓
[CATP11]	✗	✓	✓	✗	assembly-based, paradigm-based	✓	✗	✗	✗
[SSRG96]	✓	✓	✓	✗	assembly-based, paradigm-based	✗	✗	✗	✗
[GP01]	✓	✓	✓	✗	assembly-based	✓	✗	MRS	✗
[SAE03]	✗	✓	✓	✗	assembly-based	✗	✗	✗	✗
[Tol06]	✗	✓	✓	✗	assembly-based	✗	✗	✗	✗
[PRSH09]	✗	✓	✓	✗	assembly-based	✓	✗	✗	✗
[AB12]	✓	✓	✓	✗	architecture based	✗	✓	context attributes	✓
SME4PCL	✓	✓	✓	✓	architecture based, assembly based	✓	✓	feature annotation	✓

Table 7.1: Comparison of tailoring approaches

Table 7.2 illustrates a comparison of PSEE approaches (cf. Section 7.2.2), comprehensive CAME environments (cf. Section 7.2.3), and SME4PCL. The used criteria are described in the following:

1. Process Definition Language (PDL): This indicates the notation, which is used for process modeling.
2. Guidance: Guidance is referred to a capability, which helps developers in doing their work right or producing a valid outcome. Therefore this criteria states, whether or not an approach provides guidance facilities to a developer.
3. CASE Generation: CASE generation refers to the capability to create editors or other tooling, which allows for editing process artifacts, based on the information given in a process model.
4. Monitoring: This is a capability, which allows for the recording of activities performed during a process.
5. Consistency: This is a capability, which ensures the consistency between artifacts. As artifacts share information, modifications may cause negative impacts, which must be managed through a consistency management facility.
6. Data integration: This criteria concerns the capability to integrate product-specific information into the process model. Hereby, we distinguish the design of input (in) and output (out) artifacts of a method. While some approaches allow for the definition of artifacts using proprietary languages, some approaches support standard languages, such as UML. Alternatively, artifacts are referenced by static links in the file system, or artifacts are considered as black box representatives, without taking into account a method's product part in detail.
7. Execution: Whether an PSEE environment supports the execution feature or not, is determined by its capability to guide users through different activities defined in a process model in a correct order, i.e., to provide users with appropriate tasks.
8. Product-oriented: A classification means for PSEE strategies. Approaches, that focus on modeling the product-related part of methods, while they neglect process-specific parts and their enactment, are classified as product-oriented PSEE environments.
9. Process-oriented: Approaches, that deal with the process-related issues of methods and support the enactment of the process model.

Various alternative approaches are proposed to enable basic features of an PSEE environment, such as process modeling, CASE generation, and process execution. While some approaches focus individual parts more than others, there are hardly approaches, which provide meaningful support to each PSEE part. In particular, special features, such as guidance in form of a method-specific decision support system, or other specific benefits, which become possible if a workflow engine controls activities, are out of scope for most of available approaches. Therefore, in our approach, we focused on a more

Approach	PDL	Guidance	CASE Generation	Monitoring	Consistency	Data integration	Execution	Product-oriented	Process-oriented
[BGB05]	UML4SPM	✗	✗	✗	✗	in: ✗; out: ✗	✓	✗	✓
[MSMR09]	SPEM	✗	✗	✗	✗	in: ✗; out: UML	✓	✗	✓
[WEB+09]	SPACE	✗	✓	✗	✓	in: ✗; out: SPACE	✓	✗	✓
[DFOT10]	✗	✗	✗	✓	✓	in: ✗; out:reference	✗	✓	✗
[ABN+08]	SPEM	✗	✗	✗	✗	in: reference ; out:reference	✓	✗	✓
[MOS11]	BPMN	✓	✗	✗	✓	in: meta-model ; out:meta-model	✓	✗	✓
[CATP11]	SPEM	✓	✗	✗	✗	in: ✗; out:meta-model	✓	✗	✓
[SSRG96]	NATURE	✓	✗	✓	✗	in: ✗; out:ER, OMT	✓	✗	✓
[GP01]	✗	✓	✓	✗	✗	in: ✗; out:MVM	✗	✓	✗
[SAE03]	UML	✓	✓	✗	✗	in: ✗; out:UML	✓	✓	✗
[Tol06]	✗	✓	✓	✗	✗	in: ✗; out:GOPPRR	✗	✓	✗
[PRSH09]	SPEM	✗	✗	✗	✗	in: reference ; out:reference	✓	✗	✓
SME4PCL	SPEM, JWT	✓	✓	✓	✓	in: MMV ; o: MMV	✓	✗	✓

Table 7.2: Comparison of PSEE approaches

compressive environment, which enables additional features in a process-centered environment. By an explicit definition of expressive guidelines, which are associated with individual methods, a strong guidance mechanism is provided to developers to support their decision making and to prevent them from careless mistakes. Additionally, the monitoring of artifacts and modeling activities ensures consistency between artifacts and enables continuous process improvement through the analysis of monitored data.

7.3.3 Comparison of comprehensive Computer-aided Method Engineering Approaches

Finally, we compare our approach with regard to existing approaches, which consider themselves as a comprehensive CAME environment taking into account the CAME part and the CASE part likewise. Therefore, Table 7.3 illustrates a comparison of comprehensive CAME environments (cf. Section 7.2.3) and SME4PCL. The used criteria are described in the following:

1. ME Support: This concerns the general capability of an CAME environment to support ME activities.
2. Process Support: This concerns the general capability of an CAME environment to support process enactment or execution.
3. CASE Tool Generation: This concerns the general capability of an CAME environment to support the process-model-based generation of tools, which actively support a development process.
4. DSL-Integration: This concerns the capability of an CAME environment to link product-related information with the process model. While we consider the specification of meta-models from scratch as *ad-hoc* DSL-integration, we consider the integration of already existing meta models within the process model as *reuse-based* DSL-integration.
5. Traceability: This concerns the capability of an CAME environment to interrelate information spread across various artifacts of a development process for further analyzes, such as dependency analyses and change impact analyses.
6. Standard-based: This refers to the application of standards in respective approaches, in contrast to proprietary languages and formalisms.

As depicted in Table 7.3, our approach fulfills the main requirements to be an comprehensive CAME environment. To ensure the acceptance in an as large as possible community, we based our approach on standards, such as SPEM or OCL. Furthermore, to the best of our knowledge, our approach is the only one, which considers monitoring of modeling activities, by which an artifact history is established to enable automated tracing and artifact consistency management. Furthermore, the reuse and customization of available meta models or DSLs to use them according to a predefined process, is novel to this area. Although, other approaches integrate data alike, they either define a meta model from scratch or they reference relevant artifact using a static link in a file system (cf. Table 7.2).

Approach	ME Support	Process Support	CASE Tool Generation	DSL-Integration	Traceability	Standard-based
[CATP11]	✓	✓	✗	ad-hoc	✗	✓
[SSRG96]	✓	✓	✓	✗	✓	✗
[GP01]	✓	✓	✓	✗	✗	✗
[SAE03]	✓	✓	✓	ad-hoc	✗	✓
[Tol06]	✓	✗	✓	ad-hoc	✗	✗
[PRSH09]	✓	✓	✗	✗	✗	✓
SME4PCL	✓	✓	✓	reuse-based	✓	✓

Table 7.3: Comparison of CAME environments

8 Conclusions and Outlook

In this thesis we have shown, that the design and execution of software development process models can be achieved with manageable efforts by integrating additional information with process models and combining principles of model-driven engineering, method engineering, and software product lines. Thus, we addressed the objectives, that we raised in the introduction and are especially answering challenges, such as:

- Provision of effective means to manage and to evolve process descriptions for the situation at hand (by the application of our approach for software process line engineering),
- Support of developers in designing sound process descriptions, which enable effective guidance (by enabling general process information to be complemented by innovative computer-interpretable information on technical level),
- Support of the enactment and execution of sound process descriptions using computer power (by the interpretation of process model descriptions on the operational level), and
- Provision of a continuous tool chain (as demonstrated by the developed prototypical environment and the case studies).

In Section 8.1, we first summarize the achievements and contributions of this thesis. Afterwards, in Section 8.2, we describe future research possibilities that could be build upon this thesis.

8.1 Summary of Thesis

We developed solutions, that address the identified challenges and derived objectives by providing the following artifacts.

Software Process Line Engineering

In chapter 3, we successfully demonstrated the combination of method engineering techniques with software product line engineering, in order to establish a software development process line. We provided a methodology, which covers domain engineering, i.e., process family engineering, and application engineering, i.e., situational process engineering, in equal shares.
Therefore, we demonstrated the construction of an evolutionary process line repository,

which simply adopts existing process descriptions, while enabling the definition of new ones, for process family engineering. The assets stored in the repository are annotated with situational characteristics describing their application scenario, and used for the design of variable processes satisfying the needs of business-oriented and technical stakeholders.
The approach is enabled by automating the configuration and variability binding of a process family to derive situational members. Therefore, we developed a transformation, which associates business-centric process information with technical information into an intermediate feature model. The feature model is the input to the developed situational configuration mechanism, which is based on situational, priority-based characteristics of assets in the feature tree and HTN planning. Finally, we sketched our approach for validating the feasibility of derived processes with respect to available organizational resources.

Computational Method Engineering

In chapter 4, we detailed the modeling of software development processes on technical level. The additional technical information, which is required to subsequently provide an automated guidance system for software developers, concerns the control-flow semantics of these processes and contained activities, or method chunks.
We introduced a new control-flow semantics, which enables the flexible coordination of development activities based on work products' content and validation results. Furthermore, we introduced new meta models in order to integrate various information sources with a process model by the means of an aspect-oriented mechanism. Thereby, we complemented process knowledge by computer-interpretable product-specific information, tooling capabilities, and guidelines.
We introduced Meta Model Views, which enable the restriction of domain-specific languages, or meta models, to work product-specific needs, and allow for the definition traceability dependencies between the content of work products. Additionally, we provided a mechanism, which enables the specification of tooling capabilities with respect to individual activities and associated work products. Finally, we introduced a meta model and an associated graphical notation for the definition of context-specific guidelines. The guideline formalism enables the process-driven combination of various statements, which have to be ensured for the various output products of a development process.

Method-driven Guidance of Development Processes

In chapter 5, the automated processing of development process design models was described. We explained the generation of new artifacts and interpretations of process model information, which is provided with the technical design level to support guidance on operational level. Thereby, we developed a generic code generation, which derives platform-specific editor code from process models to provide development activities with relevant editor capabilities only.
In the context of each generated editor, we further demonstrated the application of defined guidelines on the operational level. Therefore, we described an execution semantics

of the process-centered guidelines, and provided a translational semantics for processing contained statements using OCL.
To monitor performed development activities and to assign affected model elements with respective work products, we developed the artifact observer mechanism. This mechanism provides editors with a standardized interface for information exchange and uses MMV information, as annotated with work products of the technical design level, for the assignment of model elements to corresponding artifacts. Finally, we demonstrated the usage of monitored information and validation results of guidelines, to enhance work product quality, to ensure the consistency between artifacts, and to assign development activities automatically.

8.2 Future Research

This thesis provides a sound overview about techniques and artifacts, which are required to enable complex model-driven development processes more efficiently. Although, we have demonstrated a working application of our approach, it bases further development.

Software Process Line Engineering

At the moment, our process line engineering approach statically links variation points and variants. To provide more flexibility and automation, the notion of method signatures or chunk descriptors (cf. [MR05, RPR98]) has to be further evaluated for matching the application scenario of variants in the context of the reference process. Thereby, the processing of predefined method chunks can be extended by additional means to configure method chunks from method fragments, as well.
Furthermore, different planning mechanisms have to be considered for process derivation in the future. While we have demonstrated the general applicability of planning techniques for the binding the variabilities in a software process line, other techniques could be more appropriate to face that challenge. Especially, if continuous process improvement becomes relevant to optimize the process line results, analysis data from past projects may influence the planning process and the quality of resulting processes considerably. Therefore, metrics are defined in the future, which are used for the analysis of monitored data available from performed projects. This information, for example, is used to perform trade-off analyses on planned processes, i.e., alternative process realizations using different methods.
Based on the results of this thesis, a national funded research project was initiated in cooperation with an industrial partner in 2012. The project aims at the development of a *Process Framework & Data Model based on a layered V-Model method*. Thereby, our future research will focus on more sophisticated mechanisms to customize development processes considering various process- and product-related standards, and other relevant needs to achieve a final product. Therefore, the introduced process line approach will be integrated with product line engineering in order to incorporate the product-related part of development on instance level, as well. A further goal of this project is the development of a mechanism to perform gap analyses to identify the discrepancies between

an enterprise's current processes and standard-conform processes, as provided by the framework automatically.

Computational Method Engineering

The presented approach focuses on individual method fragments to be extended by additional, computer-interpretable information. In the future, further method fragments will become more relevant, as well. For example, as sketched in Section 4.7, a detailed role-specific information model, which is associated with the process model, would enable user skill-specific assignment of activities, and the definition of activity-specific data access.
Furthermore, existing information models and transformations can be extended to face additional situational needs. For example, editor generation can be extended by additional transformations to support table-, tree-, or graph-based editors, by which work products can be processed in a more user centric manner. Likewise, the guideline formalism can be extended by additional operators and concepts, which enable further validation capabilities, or by an additional translational semantics to be compliant with validation frameworks or technical spaces, which are different from OCL or MDE.

Method-driven Guidance of Development Processes

In the future, it has to be evaluated, whether the applied activity-centric approach matches the needs of software development processes best. The procedural management of single activities, as discussed in this thesis, is easy to handle, and available technologies can be adopted simply. In the industrial context, though, often product-centric approaches are preferred. Therefore, it has to be evaluated, whether there are more appropriate execution mechanisms, or if different mechanisms must be combined to achieve optimal results. For example, in [SVHB05], we already sketched an idea of using agent technology for this. Especially, since agile development has more and more become of particular interest in most software development domains, we plan to evaluate the application of our ideas to this development paradigm, as well as the extension of introduced process execution mechanisms to enable agile development processes. In parallel, we plan the extension of traceability and consistency mechanisms to refine the management of multiple different concerns spread across various artifacts more efficiently.
Another objective, which we plan to face in the future, is the support of distributed development, where multiple developers are guided by one complex process model in order to process the same or different work products. Therefore, a more sophisticated model management must be established, which supports transaction concepts and multi-user access on models and artifacts.

The presented approach forms an appropriate foundation for the aforementioned enhancements of model-driven development environments and thus, serves as a suitable basis for further research.

Bibliography

[Aal98] AALST, Wil M P Van D.: The Application of Petri Nets to Workflow Management. In: *Journal of Circuits, Systems and Computers* 8 (1998), Nr. 1, 21–66. http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.30.3125 49

[AAO94] ALLOUI, I ; ARBAOUI, S ; OQUENDO, F: Process-centered environments: support for human-environment interaction and environment-mediated human cooperation. In: *Proceedings Ninth International Software Process Workshop*, 1994. – ISBN 0818667702, S. 110–113 267

[AB12] ALEGRIA, Julio Ariel H. ; BASTARRICA, Maria C.: Building software process lines with CASPER. In: *2012 International Conference on Software and System Process (ICSSP)*, IEEE, Juni 2012. – ISBN 978–1–4673–2352–9, 170–179 265, 274

[ABB+09] ANTONUCCI, Yvonne L. ; BARIFF, Martin ; BENEDICT, Tony ; CHAMPLIN, Brett ; DOWNING, Bruce D. ; FRANZEN, Jason ; MADISON, Daniel J. ; LUSK, Sandra ; SPANYI, Andrew ; TREAT, Mark ; ZHAO, J. L. ; RASCHKE, Robyn L.: *Guide to the Business Process Management Common Body of Knowledge*. Wettenberg : CreateSpace Independent Publishing Platform, 2009. – 234 S. 41

[ABN+08] ALDAZABAL, A ; BAILY, T ; NANCLARES, F ; SADOVYKH, A ; HEIN, C ; ESSER, M ; RITTER, T: Automated Model Driven Development Processes. In: *Proceedings of the ECMDA workshop on Model Driven Tool and Process Integration*, Fraunhofer IRB Verlag, 2008 261, 268, 276

[ABO11] ALEGRÍA, Julio A H. ; BASTARRICA, María C. ; OCHOA, Sergio F.: An MDE Approach to Software Process Tailoring. In: *Proceedings of the 2011 International Conference on Software and Systems Process* (2011), 43–52. http://dx.doi.org/10.1145/1987875.1987885. – DOI 10.1145/1987875.1987885. ISBN 9781450305808 262, 263, 274

[AC04] ANTKIEWICZ, Michal ; CZARNECKI, Krzysztof: FeaturePlugin: Feature Modeling Plug-in for Eclipse. (2004), 67 – 72. http://www.citeulike.org/user/hugoarboleda/article/2907986 93

[ACF97] AMBRIOLA, Vincenzo ; CONRADI, Reidar ; FUGGETTA, Alfonso: Assessing process-centered software engineering environments. In: *ACM Transactions on Software Engineering and Methodology* 6 (1997), Nr. 3, 283–328. http://dx.doi.org/10.1145/258077.258080. – DOI 10.1145/258077.258080. – ISSN 1049331X 63, 265

[ACT08] AHERN, Denni M. ; CLOUSE, Aaron ; TURNER, Richard: *CMMI Distilled: A Practical Introduction to Integrated Process Improvement (3rd Edition)*. Addison-Wesley Professional, 2008 23

[ADOV02] ARBAOUI, Selma ; DERNIAME, Jean-Claude ; OQUENDO, Flavio ; VERJUS, Hervé: A Comparative Review of Process-Centered Software Engineering Environments. In: *Annals of Software Engineering* 14 (2002), Nr. 1, 311–340. http://dx.doi.org/10.1109/ECBS.2012.11. – DOI 10.1109/ECBS.2012.11. ISBN 9781467309127 63, 64, 265

[AFSK11] ALEIXO, Fellipe A. ; FREIRE, Marília A. ; SANTOS, Wanderson C. ; KULESZA, Uirá: Automating the Variability Management, Customization and Deployment of Software Processes: A Model-Driven Approach. In: *Enterprise Information Systems* 73 (2011), 372–387–387. http://dx.doi.org/10.1007/978-3-642-19802-1. – DOI 10.1007/978–3–642–19802–1. ISBN 978–3–642–19801–4 264, 274

[AGK06] ATKINSON, Colin ; GUTHEIL, Matthias ; KIKO, Kilian: On the Relationship of Ontologies and Models. In: *Workshop on Meta-Modeling (WoMM), Karlsruhe, Germany*, 2006 39

[AHM+13] ASADI, Mohsen ; HONKE, Benjamin ; MOHABBATI, Bardia ; EISENBARTH, Thomas ; GASEVIC, Dragan ; BAUER, Bernhard ; BAGHERI, Ebrahim: A Process Line Approach for Situational Process Engineering. In: *Submitted to JOURNAL OF SOFTWARE: EVOLUTION AND PROCESS* (2013) 78

[AHW03] AALST, Wil M P Van D. ; HOFSTEDE, Arthur H M. ; WESKE, Mathias: Business Process Management : A Survey. In: *Business Process Management* 2678 (2003), Nr. 1, 1–12. http://dx.doi.org/10.1007/3-540-44895-0_1. – DOI 10.1007/3–540–44895–0_1. – ISBN 9783540403180 41, 42, 49

[Ali07] ALI NIKNAFS, Mohsen A.: Ontology-Based Method Engineering. In: *International Journal of Computer Science and Network Security. IJCSNS* 7 (2007), Nr. 8. http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.121.3234 61

[ALSS08] AMELUNXEN, Carsten ; LEGROS, Elodie ; SCHÜRR, Andy ; STÜRMER, Ingo: Checking and Enforcement of Modeling Guidelines with Graph Transformations. In: *Applications of Graph Transformations with Industrial Relevance* 5088 (2008), Nr. 4, 313–328. http://dx.doi.org/10.1016/j.jvlc.2009.04.005. – DOI 10.1016/j.jvlc.2009.04.005. – ISBN 9783540890195 40, 266

[AMB+04] ABRAN, Alain ; MOORE, James W. ; BOURQUE, Pierre ; DUPUIS, Robert ; TRIPP, Leonard L. ; ABRAN, Alain (Hrsg.) ; BOURQUE, Pierre (Hrsg.) ; DUPUIS, Robert (Hrsg.) ; MOORE, James W. (Hrsg.) ; TRIPP, Leonard L. (Hrsg.): *Guide to the Software Engineering Body of Knowledge*. Bd. 19759. IEEE, 2004. – 204 S. http://www.computer.org/portal/web/swebok/htmlformat 51, 52, 102, 145

[Amb05] AMBLER, Scott W.: *The Elements of UML 2.0 Style*. Cambridge University Press, 2005. – 146 S. http://www.amazon.com/Elements-UML-TM-2-0-Style/dp/0521616786. – ISBN 9780521616782 144

[AMGB11] ASADI, Mohsen ; MOHABBATI, Bardia ; GAŠEVIĆ, Dragan ; BAGHERI, Ebrahim: Developing Families of Method-Oriented Architecture. In: *Engineering Methods in the ServiceOriented Context* (2011), 168–183. http://www.springerlink.com/index/P55246557J475811.pdf 264, 274

[ARB08] AHARONI, Anat ; REINHARTZ-BERGER, Iris: A Domain Engineering Approach for Situational Method Engineering. In: *Conceptual Modeling ER 2008* 5231 (2008), 455–468. http://www.springerlink.com/content/d53h076r07187338 263

[ARNRSG06] AIZENBUD-RESHEF, Netta ; NOLAN, Brian T. ; RUBIN, Julia ; SHAHAM-GAFNI, Y: Model traceability. In: *IBM Systems Journal* 45 (2006), Nr. 3, 515–526. http://dx.doi.org/10.1147/sj.453.0515. – DOI 10.1147/sj.453.0515. – ISSN 00188670 266

[ASJW05] ALLWEYER, Thomas ; SCHEER, August-Wilhelm ; JOST, Wolfram ; WAGNER, Karl: Von Prozessmodellen zu lauffähigen Anwendungen. In: *Von Prozessmodellen zu lauffähigen Anwendungen* (2005), 173–195. http://dx.doi.

org/10.1007/b138752. – DOI 10.1007/b138752. ISBN 3540234578 43

[ATE08] ATESST: *Advancing Traffic Efficiency and Safety through Software Technology*. www.atesst.org. Version: 2008 23

[Aut10] AUTOMOTIVE SIG: Automotive SPICE Process Assessment Model / Verband der Automobilindustrie e.V. 2010. – Forschungsbericht. – 146 S. 52, 248

[AUT12] AUTOSAR: *Automotive open system architecture*. www.autosar.org. Version: 2012 23, 24, 25, 26, 28, 29, 30, 322

[B0́5] BÉZIVIN, Jean: On the unification power of models. In: *Software & Systems Modeling* 4 (2005), Mai, Nr. 2, 171–188. http://dx.doi.org/10.1007/s10270-005-0079-0. – DOI 10.1007/s10270–005–0079–0. – ISSN 1619–1366 13

[BACR08] BURMEISTER, Birgit ; ARNOLD, M ; COPACIU, Felicia ; RIMASSA, Giovanni: BDI-agents for agile goal-oriented business processes. In: *AAMAS '08: Proceedings of the 7th international joint conference on Autonomous agents and multiagent systems*. Richland, SC : International Foundation for Autonomous Agents and Multiagent Systems, 2008, S. 37–44 261

[Bae04] BAEYENS, Tom: *The State of Workflow*. jBoss.org/jBPM, 2004 45

[Bae05] BAETEN, J.C.M.: A brief history of process algebra. In: *Theoretical Computer Science* 335 (2005), Mai, Nr. 2-3, 131–146. http://dx.doi.org/10.1016/j.tcs.2004.07.036. – DOI 10.1016/j.tcs.2004.07.036. – ISSN 03043975 50

[BAGS10] BAGHERI, Ebrahim ; ASADI, Mohsen ; GASEVIC, Dragan ; SOLTANI, Samaneh: Stratified Analytic Hierarchy Process : Prioritization and Selection of Software Features. In: *Software Product Lines Going Beyond* 6287 (2010), Nr. 3, 300–315. http://dx.doi.org/10.1007/978-3-642-15579-6. – DOI 10.1007/978–3–642–15579–6. ISBN 9783642155789 69, 89, 94, 105

[BAH+11] BAGHERI, Samaneh S. ; ASADI, Mohsen ; HATALA, Marek ; GASEVIC, Dragan ; EBRAHIM: Automated Planning for Feature Model Configuration based on Stakeholders' Business Concerns. In: *26th IEEE/ACM ASE*, 2011, S. 4 94, 95

[Bat05] BATORY, Don: Feature Models, Grammars, and Propositional Formulas. In: *Software Product Lines* 3714 (2005), Nr. March, 7–20. http://dx.doi.org/10.1007/

11554844_3. – DOI 10.1007/11554844_3. – ISBN 9783540289364 69

[BB01] BRETON, E ; BEZIVIN, J: Model driven process engineering. In: *Computer Software and Applications Conference 2001 COMPSAC 2001 25th Annual International* TUCS, IEEE, 2001. – ISBN 0769513727, 225–230 50, 56

[BBG+05a] BAYER, Joachim ; BUHL, Winfried ; GIESE, Cord ; LEHNER, Theresa ; OCAMPO, Alexis ; PUHLMANN, Frank ; RICHTER, Ernst ; SCHNIEDERS, Arnd ; WEILAND, Jens ; WESKE, Mathias: *Process Family Engineering: Modeling variant-rich processes*. 2005 263

[BBG05b] BEYDEDA, Sami ; BOOK, Matthias ; GRUHN, Volker ; BEYDEDA, Sami (Hrsg.) ; BOOK, Matthias (Hrsg.) ; GRUHN, Volker (Hrsg.): *Model-Driven Software Development*. Springer, 2005. – 464 S. http://dx.doi.org/10.1007/3-540-28554-7_9. http://dx.doi.org/10.1007/3-540-28554-7_9. – ISBN 3642065023 35

[BCK03] BASS, Len ; CLEMENTS, Paul ; KAZMAN, Rick: *Software Architecture in Practice*. 2. Amsterdam : Addison-Wesley Professional, 2003. – 560 S. http://www.amazon.com/dp/0321154959. – ISBN 0321154959 235

[BE96] BELKHATIR, N ; ESTUBLIER, J: *Supporting reuse and configuration for large scale software process models* 262

[BEFH12] BAUER, Bernhard ; EISENBARTH, Thomas ; FRENZEL, Christoph ; HONKE, Benjamin: Resource-oriented Consistency Analysis of Engineering Processes. In: MACIASZEK, Leszek A. (Hrsg.) ; CUZZOCREA, Alfredo (Hrsg.) ; CORDEIRO, José (Hrsg.): *Proceedings of the 14th International Conference on Enterprise Information Systems, Volume 3*. Wroclaw, Poland : SciTePress, 2012, S. 206–211 101, 104

[BEM93] BELKHATIR, N ; ESTUBLIER, J ; MELO, W L.: *Software process model and work space control in the Adele system* 267

[BFG+02] BOSCH, Jan ; FLORIJN, Gert ; GREEFHORST, Danny ; KUUSELA, Juha ; OBBINK, J H. ; POHL, Klaus: Variability Issues in Software Product Lines. In: *Software ProductFamily Engineering* 2290 (2002), 13–21. http://dx.doi.org/10.1007/3-540-47833-7_3. – DOI 10.1007/3–540–47833–7_3. – ISBN 3540436596 66, 72

[BFK+99] BAYER, Joachim ; FLEGE, Oliver ; KNAUBER, Peter ; LAQUA, Roland ; MUTHIG, Dirk ; SCHMID, Klaus ; WIDEN, Tanya ; DEBAUD, Jean-Marc: PuLSE: a methodology to

develop software product lines. In: *Proceeding SSR '99 Proceedings of the 1999 symposium on Software reusability* Bd. 99, ACM, 1999. – ISBN 1581131011, 122–131 69

[BG04] Brickley, Dan ; Guha, R V.: *RDF Vocabulary Description Language 1.0: RDF Schema*. W3C Recommendation. http://www.w3.org/TR/rdf-schema/. Version: 2004 39

[BGB05] Bendraou Reda ; Gervais Marie-Pierre ; Blanc Xavier: UML4SPM: A UML2.0-Based Metamodel for Software Process Modelling. In: *MoDELS 2005*, 2005, S. 17 – 38 267, 276

[BGB06] Bendraou, Reda ; Gervais, Marie-Pierre ; Blanc, Xavier: UML4SPM: An Executable Software Process Modeling Language Providing High-Level Abstractions. In: *Enterprise Distributed Object Computing Conference 2006 EDOC 06 10th IEEE International* Bd. 6, Ieee, 2006. – ISBN 076952558X, 297–306 266

[BGL+04] Bachmann, Felix ; Goedicke, Michael ; Leite, Julio ; Nord, Robert ; Pohl, Klaus ; Ramesh, Balasubramaniam ; Vilbig, Alexander: A meta-model for representing variability in product family development. In: *Software ProductFamily Engineering* 3014 (2004), 66–80. http://dx.doi.org/10.1007/978-3-540-24667-1_6. – DOI 10.1007/978–3–540–24667–1_6. – ISBN 3540219412 66

[BHH+04] Bechhofer, Sean ; Harmelen, Frank van ; Hendler, Jim ; Horrocks, Ian ; McGuiness, Deborah L. ; Patel-Schneider, Peter F. ; Stein, Lynn A.: *OWL Web Ontology Language - Reference*. W3C Recommendation. http://www.w3.org/TR/owl-ref/. Version: 2004 39

[BHP03] Bühne, Stan ; Halmans, Günter ; Pohl, Klaus: Modelling Dependencies between Variation Points in Use Case Diagrams. In: *PROCEEDINGS OF 9TH INTL. WORKSHOP ON REQUIREMENTS ENGINEERING - FOUNDATIONS FOR SOFTWARE QUALITY* (2003), 43. http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.14.7645 66, 70

[BHS04] *Kapitel* Descriptio. In: Baader, Franz ; Horrocks, Ian ; Sattler, Ulrike: *Handbook on Ontologies*. Berlin Heidelberg : Springer-Verlag, 2004 (International Handbooks on Information Systems), S. 3–29 39

[BJ06] BEZIVIN, J ; JOUAULT, F: Using ATL for Checking Models. In: *Electronic Notes in Theoretical Computer Science* 152 (2006), März, 69–81. http://dx.doi.org/10.1016/j.

entcs.2006.01.015. – DOI 10.1016/j.entcs.2006.01.015. – ISSN 15710661 40, 266

[BJF09] BENDRAOU, R ; JÉZÉQUEL, J M. ; FLEUREY, F: Combining Aspect and Model-Driven Engineering Approaches for Software Process Modeling and Simulation. (2009) 266

[BJJ+01] BOSCH, J. ; JARING, M. ; JOHNSSON, S. ; SCHMID, K. ; THIEL, S. ; THOME, B. ; TROSCH, S.: Task 1.2: Domain Analysis: Consortium-Wide Deliverable on Scoping, Deliverable of ESAPS Project / Eureka 2023 Programme, ITEA Project 99005. 2001. – Forschungsbericht 81

[BL01] BERNERS-LEE, Tim: The Semantic Web. In: *Scientific American* (2001) 39

[BL04] BRACHMAN, Ronald J. ; LEVESQUE, Hector J.: *{K}nowledge {R}epresentation and {R}easoning*. Elsevier, 2004 39

[BLP04] BÜHNE, Stan ; LAUENROTH, Kim ; POHL, Klaus: Why is it not Sufficient to Model Requirements Variability with Feature Models? In: *Workshop on Automotive Requirements Engineering AURE04* 04 (2004), Nr. September, 5–12. http://www.sse.uni-essen.de/wms/pubs/AuRE_cameraready.pdf 66

[BM04] BECKETT, Dave ; MCBRIDE, Brian: *RDF/XML Syntax Specification (Revised)*. W3C Recommendation. http://www.w3.org/TR/rdf-syntax-grammar/. Version: 2004 39

[BMMM08] BLANC, Xavier ; MOUNIER, Isabelle ; MOUGENOT, Alix ; MENS, Tom: Detecting model inconsistency through operation-based model construction. In: DWYER, Matthew B. (Hrsg.) ; GRUHN, Volker (Hrsg.): *Proceedings of the 13th international conference on Software engineering ICSE 08* Bd. 1, ACM Press, 2008. – ISBN 9781605580791, 511–520 266

[BPKJ07] BECKER, Jörg ; PFEIFFER, Daniel ; KNACKSTEDT, Ralf ; JANIESCH, Christian: Configurative Method Engineering - On the Applicability of Reference Modeling Mechanisms in Method Engineering. In: *Engineering* (2007), 1–12. http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.91.4198&rep=rep1&type=pdf 56, 262

[BPM09] BPMN: *Business Process Model and Notation, Version 2.0 - Beta 1*. http://www.omg.org/spec/BPMN/2.0/, August 2009 47, 261

[Bri96] BRINKKEMPER, Sjaak: Method engineering: engineering of information systems development methods and tools. In: *Information and Software Technology* 38 (1996), Nr. 4, 275–280. http://dx.doi.org/10.1016/0950-5849(95)01059-9. – DOI 10.1016/0950–5849(95)01059–9. – ISBN 3540422153 54, 56, 57, 58

[Bro06] BROY, Manfred: Challenges in automotive software engineering. In: *Proceeding of the 28th international conference on Software engineering ICSE 06* Bd. 2006 ACM, ACM Press, 2006 (ICSE '06). – ISBN 1595933751, 33–42 13

[BSH98] BRINKKEMPER, Sjaak ; SAEKI, Motoshi ; HARMSEN, Frank: Assembly Techniques for Method Engineering. In: *Advanced Information Systems Engineering* (1998), S. 381 23

[BSH01] BRINKKEMPER, Sjaak ; SAEKI, Motoshi ; HARMSEN, Frank: A Method Engineering Language for the Description of Systems Development Methods. (2001), Juni, 473–476. http://dl.acm.org/citation.cfm?id=646089.680061. ISBN 3–540–42215–3 269

[BSK98] BEN-SHAUL, Israel Z. ; KAISER, Gail E.: Federating Process-Centered Environments: The Oz Experience. In: *Automated Software Engineering* 5 (1998), Nr. 1. http://dx.doi.org/10.1023/A:1008610426207. – DOI 10.1023/A:1008610426207 267

[BTRC05] BENAVIDES, David ; TRINIDAD, Pablo ; RUIZ-CORTÉS, Antonio: Automated Reasoning on Feature Models. In: *LNCS Advanced Information Systems Engineering 17th International Conference CAiSE 2005* 01 (2005), Nr. 2, 491–503. http://dx.doi.org/10.1007/11431855_34. – DOI 10.1007/11431855_34. – ISBN 3540260951 69

[CAF04] CAFE: *CAFE - From Concepts to Application in System-Family Engineering, (http://www.esi.es/Cafe/)*. ITEA project. http://www.esi.es/Cafe/. Version: 2004 23

[CATP11] CERVERA, Mario ; ALBERT, Manoli ; TORRES, Victoria ; PELECHANO, Vicente: Turning Method Engineering Support into Reality. Version: 2011. http://dx.doi.org/10.1007/978-3-642-19997-4. In: RALYTÉ, Jolita (Hrsg.) ; MIRBEL, Isabelle (Hrsg.) ; DENECKÈRE, Rébecca (Hrsg.): *Engineering Methods in the Service-Oriented Context* Bd. 351. Berlin, Heidelberg : Springer Berlin Heidelberg, 2011. – DOI 10.1007/978–3–642–19997–4. – ISBN 978–3–642–19996–7, 138–152 264, 271, 274, 276, 278

[CCT+03] Calvary, Gaëlle ; Coutaz, Joëlle ; Thevenin, David ; Limbourg, Quentin ; Bouillon, Laurent ; Vanderdonckt, Jean: A Unifying Reference Framework for multi-target user interfaces. In: *Interacting with Computers* 15 (2003), Nr. 3, 289–308. http://dx.doi.org/10.1016/S0953-5438(03)00010-9. – DOI 10.1016/S0953–5438(03)00010–9. – ISSN 09535438 266

[CE00] Czarnecki, Krysztof ; Eisenecker, Ulrich: *Generative Programming, Methods, Tools, and Applications*. Addison-Wesley, 2000. – 864 S. 69

[CHCC03] Cleland-Huang, Jane ; Chang, Carl K. ; Christensen, Mark: Event-based traceability for managing evolutionary change. In: *IEEE Transactions on Software Engineering* 29 (2003), Nr. 9, 796–810. http://dx.doi.org/10.1109/TSE.2003.1232285. – DOI 10.1109/TSE.2003.1232285. – ISSN 00985589 188, 266

[Che76] Chen, Peter Pin-shan: The Entity-Relationship Model: Toward a Unified View of Data. In: *ACM Transactions on Database Systems* 1 (1976), 9—–36. http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.123.1085 269

[CHE04] Czarnecki, Krzysztof ; Helsen, Simon ; Eisenecker, Ulrich: Staged Configuration Using Feature Models. In: *Software Product Lines* 3154 (2004), Nr. 2, 266–283. http://dx.doi.org/10.1007/b100081. – DOI 10.1007/b100081. – ISBN 3540229183 69

[CHE05] Czarnecki, Krzysztof ; Helsen, Simon ; Eisenecker, Ulrich: Formalizing cardinality-based feature models and their specialization. In: *Software Process: Improvement and Practice* 10 (2005), Nr. 1, 7–29. http://dx.doi.org/10.1002/spip.213. – DOI 10.1002/spip.213. – ISSN 10774866 69, 90

[CHW96] Ciancarini, Paolo ; Hankin, Chris ; Wegner, Peter: *Coordination as constrained interaction (extended abstract) - Coordination Languages and Models - Lecture Notes in Computer Science*. http://dx.doi.org/10.1007/3-540-61052-9_37. Version: 1996 44

[CKH08] *Kapitel* Computer-i. In: Clercq, Paul D. ; Kaiser, Katharina ; Hasmann, Arei: *Computer-based Medical Guidelines and Protocols: A Primer and current Trends*. IOS Press, 2008, S. 22–43 266

[CKO92] CURTIS, Bill ; KELLNER, Marc I. ; OVER, Jim: Process Modeling. In: *Communications of the ACM, Special issue on analysis and modeling in software development* 35 (9) (1992), S. 75–90 43

[cmm08] CARNEGIE MELLON SOFTWARE ENGINEERING INSTITUTE (SEI): Capability Maturity Model Integration (CMMI). 2008. – Forschungsbericht 52, 108

[CMM10] CMMI PRODUCT TEAM: CMMI for Development / Software Engineering Institute. 2010. – Forschungsbericht. – 482 S. 247

[CMV96] CONSTANTOPOULOS, Panos ; MYLOPOULOS, John ; VASSILIOU, Yannis: MetaEdit+ A fully configurable multi-user and multi-tool CASE and CAME environment. Version: 1996. http://dx.doi.org/10.1007/3-540-61292-0. In: *Advanced Information Systems Engineering* Bd. 1080. Berlin, Heidelberg : Springer Berlin Heidelberg, 1996. – DOI 10.1007/3–540–61292–0. – ISBN 978–3–540–61292–6, 1–21 271

[CN01] CLEMENTS, Paul C. ; NORTHORP, Linda M.: *Software Product Lines: Practices and Patterns*. Addison Wesely, 2001 23, 52, 64, 65, 69, 70, 81, 82

[Com00] COMMITTEE, Standards: IEEE-Std-1471-2000 Recommended Practice for Architectural Description of Software-Intensive Systems. In: *IEEE httpstandards ieee org* (2000), Nr. IEEE-Std-1471-2000. http://scholar.google.com/scholar?hl=en&btnG=Search&q=intitle:IEEE+Recommended+Practice+for+Architectural+Description+of+Software-Intensive+Systems#1. ISBN 0738125180 115

[Cun00] CUNIN, Pierre-Yves: The PIE project: An introduction. Version: 2000. http://dx.doi.org/10.1007/BFb0095008. In: CONRADI, Reidar (Hrsg.): *Software Process Technology* Bd. 1780. Springer Berlin Heidelberg, 2000. – DOI 10.1007/BFb0095008. – ISBN 978–3–540–67140–4, 1–5 267

[Dav87] DAVIS, Stan: *Future Perfect*. Boston, Massachusetts : Addison-Wesley, 1987 65

[Dav93] DAVENPORT, Thomas H. ; PRESS, Harvard Business S. (Hrsg.): *Process innovation: reengineering work through information technology*. Harvard Business School Press, 1993. – 337 S. http://dx.doi.org/10.1016/

0923-4748(94)90026-4. http://dx.doi.org/10.1016/0923-4748(94)90026-4. – ISBN 0875843662 41

[Dav03] DAVIS, James: GME: The Generic Modeling Environment. In: *Companion of the 18th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications - OOPSLA '03*. New York, New York, USA : ACM Press, Oktober 2003. – ISBN 1581137516, 82 266

[DEA] DAMI, S ; ESTUBLIER, J ; AMIOUR, M.: Apel: A Graphical Yet Executable Formalism for Process Modeling. In: *Automated Software Engineering* 5, Nr. 1, 61–96. http://dx.doi.org/10.1023/A:1008658325298. – DOI 10.1023/A:1008658325298. – ISSN 1573–7535 266

[def13] *Definition of METHODOLOGY*. http://www.merriam-webster.com/dictionary/methodology. Version: 2013 53

[DF94] DOWSON, Mark ; FERNSTRÖM, Christer: Towards requirements for enactment mechanisms. Version: 772, 1994. http://dx.doi.org/10.1007/3-540-57739-4. In: WARBOYS, Brian C. (Hrsg.): *Software Process Technology* Bd. 772. 772. Berlin, Heidelberg : Springer Berlin Heidelberg, 1994. – DOI 10.1007/3–540–57739–4. – ISBN 978–3–540–57739–3, 90–106 63

[DFOT10] DE LUCIA, Andrea ; FASANO, Fausto ; OLIVETO, Rocco ; TORTORA, Genoveffa: Fine-grained management of software artefacts: the ADAMS system. In: *Software Practice and Experience* 40 (2010), Nr. 11, 1007–1034. http://dx.doi.org/10.1002/spe.986. – DOI 10.1002/spe.986. – ISSN 1097024X 268, 276

[DG94] DERNIAME, Jean-Claude ; GRUHN, Volker: Development of process-centered IPSEs in the ALF project. In: *Journal of Systems Integration* 4 (1994), April, Nr. 2, 127–150. http://dx.doi.org/10.1007/BF01975433. – DOI 10.1007/BF01975433. – ISSN 0925–4676 267

[DG98] DEITERS, Wolfgang ; GRUHN, Volker: *Process Management in Practice Applying the FUNSOFT Net Approach to Large-Scale Processes*. http://dx.doi.org/10.1023/A:1008654224389. Version: 1998 267

[Die12] DIEBOLD, Philipp: *A Framework for Goal-oriented Process Configuration*, University of Kaiserslautern, Master Thesis, 2012. – 136 S. 52

[DIKS09] DENECKERE, Rebecca ; IACOVELLI, Adrian ; KORNYSHOVA, Elena ; SOUVEYET, Carine: From Method

Fragments to Method Services. In: *Proceedings of EMM-SAD08* (2009). http://arxiv.org/abs/0911.0428 61, 264

[DKH08] De Clercq, Paul ; Kaiser, Katharina ; Hasman, Arie: Computer-Interpretable Guideline formalisms. In: *Studies In Health Technology And Informatics* 139 (2008), 22–43. http://www.pubmedcentral.nih.gov/articlerender.fcgi?artid=2858861&tool=pmcentrez&rendertype=abstract. – ISSN 09269630 150

[DO05] Derniame, Jean-Claude ; Oquendo, Flavio: Key Issues and New Challenges in Software Process Technology. In: *European Journal for the Informatics Professional* 5 (2005), Nr. 5 63

[Dor93] Dorling, Alec: SPICE: Software Process Improvement and Capability Determination. In: *Software Quality Journal* 2 (1993), S. 209–224 23

[Dow87] Dowson, M.: Iteration in the software process; review of the 3rd International Software Process Workshop. (1987), März, 36–41. http://dl.acm.org/citation.cfm?id=41765.41770. ISBN 0–89791–216–0 51

[Dow93] Dowson, Marc: Consistency Maintenance in Process Sensitive Environments. In: *Proc. Workshop on PSE Architectures*, 1993 62

[DPS03] De Capitani Di Vimercati, S ; Paraboschi, S ; Samarati, P: Access control: principles and solutions. In: *SoftwarePractice and Experience* 33 (2003), Nr. 5, 397–421. http://dx.doi.org/10.1002/spe.513. – DOI 10.1002/spe.513. – ISSN 1097024X 167

[DS98] Deneckère, Rébecca ; Souveyet, Carine: Patterns for Extending an OO Model with Temporal Features. Version: 1998. http://dx.doi.org/10.1007/978-1-4471-0895-5. In: Rolland, Collete (Hrsg.) ; Grosz, George (Hrsg.): *OOIS 98*. London : Springer London, 1998. – DOI 10.1007/978–1–4471–0895–5. – ISBN 978–1–85233–046–0, 201–218 60

[DSB04] Deelstra, Sybren ; Sinnema, Marco ; Bosch, Jan: A Product Derivation Framework for Software Product Families. In: VanDerLinden, F (Hrsg.): *5th Workshop on Product Family Engineering PFE5* Bd. 3014, Springer, 2004 (LECTURE NOTES IN COMPUTER SCIENCE). – ISBN 3540219412, 473–484 72

[DZR+04] DE OLIVEIRA, K M. ; ZLOT, F ; ROCHA, A R. ; TRAVASSOS, G H. ; GALOTTA, C ; DE MENEZES, C S.: Domain-oriented software development environment. In: *Journal of Systems and Software* 10 (2004), Nr. 2, 145–161. http://dx.doi.org/10.1016/S0164-1212(03)00233-4. – DOI 10.1016/S0164–1212(03)00233–4. – ISSN 01641212 267

[EBB05] ERIKSSON, Magnus ; BÖRSTLER, Jürgen ; BORG, Kjell: The PLUSS Approach - Domain Modeling with Features, Use Cases and Use Case Realizations. In: OBBINK, Henk (Hrsg.) ; POHL, Klaus (Hrsg.): *Software Product Lines* Bd. 3714. Berlin, Heidelberg : Springer Berlin Heidelberg, 2005 (Lecture Notes in Computer Science). – ISBN 978–3–540–28936–4, 33–44 69

[EBLSp10] ELSNER, Christoph ; BOTTERWECK, Goetz ; LOHMANN, Daniel ; SCHRÖDER-PREIKSCHAT, Wolfgang: Variability in Time - Product Line Variability and Evolution Revisited. In: *Fourth International Workshop on Variability Modelling of Softwareintensive Systems VaMoS Celebrating 20 Years of Feature Models*, 2010 66

[EMF11] *Eclipse Modeling Framework Project- EMF - Home*. http://www.eclipse.org/modeling/emf/. Version: 2011 118

[EPF10] *Eclipse Foundation, EPF Composer*. http://www.eclipse.org/epf/. Version: 2010 108

[Erl05] ERL, Thomas: *Service-Oriented Architecture: Concepts, Technology, and Design*. Prentice Hall Publisher, 2005 61

[ES97] EBERT, Jürgen ; SÜTTENBACH, Roger: Meta-CASE in Practice: a Case for KOGGE. (1997), 203–216. http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.49.4892 266

[Fam05] FAMILIES: *FAct-based Maturity through Institutionalisation Lessons-learned and Involved Exploration of System-family engineering*. ITEA project. http://www.esi.es/Families/. Version: 2005 23

[FHKS09] FRIEDRICH, Jan ; HAMMERSCHALL, Ulrike ; KUHRMANN, Marco ; SIHLING, Marc: *Das V-Modell XT*. Springer Berlin Heidelberg, 2009 http://www.springer.com/computer/swe/book/978-3-642-01487-1 23

[Fir04] FIRESMITH, Donald: Creating a Project-Specific Requirements Engineering Process. In: *Engineering* 3 (2004), Nr. 5, 27–39. http://dx.doi.org/10.5381/jot.2004.3.5.c4. – DOI 10.5381/jot.2004.3.5.c4. – ISSN 16601769 262

[FK06] FURUKAWA, Y ; KAWAMURA, S: Automotive electronics system, software, and local area network. In: *HardwareSoftware Codesign and System Synthesis 2006 CODESISSS 06 Proceedings of the 4th International Conference*, ACM Press, 2006 (2006 4th IEEE/ACM/IFIPHardware/Software Codesign and System Synthesis (CODES+ISSS)). – ISBN 1595933700, 2 13

[FKN+92] FINKELSTEIN, Anthony ; KRAMER, Jeff ; NUSEIBEH, B ; FINKELSTEIN, L ; GOEDICKE, Michael: Viewpoints: A Framework for Integrating Multiple Perspectives in System Development. In: *International Journal of Software Engineering and Knowledge Engineering* 2 (1992), Nr. 1, 31–57. http://dx.doi.org/10.1142/S0218194092000038. – DOI 10.1142/S0218194092000038 115

[Fra03] FRANKEL, David S.: *{M}odel {D}riven {A}rchitecture - Applying {MDA} to Enterprise Computing*. OMG Press, 2003 254

[Fug00] FUGGETTA, Alfonso: Software process: a roadmap. In: *ICSE '00: Proceedings of the Conference on The Future of Software Engineering*. New York, NY, USA : ACM, 2000. – ISBN 1–58113–253–0, S. 25–34 51

[GB02] GEYER, Lars ; BECKER, Martin: On the Influence of Variabilities on the Application-Engineering Process of a Product Family. Version: 2379, Juli 2002. http://dx.doi.org/10.1007/3-540-45652-X. In: CHASTEK, Gary J. (Hrsg.): *Software Product Lines* Bd. 2379. 2379. Berlin, Heidelberg : Springer Berlin Heidelberg, Juli 2002. – DOI 10.1007/3–540–45652–X. – ISBN 978–3–540–43985–1, 1–14 66

[GD12] GUPTA, Daya ; DWIVEDI, Rinky: Method configuration from situational method engineering. In: *ACM SIGSOFT Software Engineering Notes* 37 (2012), Mai, Nr. 3, 1–11. http://dx.doi.org/10.1145/180921.2180934. – DOI 10.1145/180921.2180934. – ISSN 0163–5948 262

[GDD09] GASEVIC, Dragan ; DEVEDZIC, Vladan ; DJURIC, Dragan: *Model Driven Engineering and Ontology Development*. Springer Berlin Heidelberg, 2009. – 378 S. 39

[GDDD04] GASEVIC, Dragan ; DJURIC, Dragan ; DEVEDZIC, Vladan ; DAMJANOVIC, Violeta: Approaching OWL and MDA Through Technological Spaces. In: *Workshop in Software Model Engineering (WiSME), Lisbon, Portugal*, 2004 35

[GEM10] GALSTER, M ; EBERLEIN, A ; MOUSSAVI, M: Systematic selection of software architecture styles. In: *IET Soft-*

ware 4 (2010), Nr. 5, 349. http://dx.doi.org/10.1049/iet-sen.2009.0004. – DOI 10.1049/iet–sen.2009.0004. – ISSN 17518806 40

[GF94] GOTEL, O ; FINKELSTEIN, A: An analysis of the requirements traceability problem. In: *Proceedings of IEEE International Conference on Requirements Engineering* pp (1994), 94–101. http://eprints.ucl.ac.uk/749/ 212

[GFD98] GRISS, M L. ; FAVARO, J ; D'ALESSANDRO, M: Integrating feature modeling with the RSEB. In: *Proceedings Fifth International Conference on Software Reuse Cat No98TB100203* (1998), 76–85. http://dx.doi.org/10.1109/ICSR.1998.685732. – DOI 10.1109/ICSR.1998.685732. – ISBN 0818683775 68

[GG07] GALVAO, I ; GOKNIL, Arda: Survey of Traceability Approaches in Model-Driven Engineering. In: *11th IEEE International Enterprise Distributed Object Computing Conference EDOC 2007* 07pages (2007), 313–313. http://dx.doi.org/10.1109/EDOC.2007.42. – DOI 10.1109/EDOC.2007.42. – ISBN 0769528910 266

[GH98] GRUNDY, John C. ; HOSKING, John G.: Serendipity: Integrated Environment Support for Process Modelling, Enactment and Work Coordination. In: *Automated Software Engineering* 5 (1998), Nr. 1, 27–60. http://dx.doi.org/10.1023/A:1008606308460. – DOI 10.1023/A:1008606308460 267

[GHJ94] GAMMA, Erich ; HELM, Richard ; JOHNSON, Ralph E.: *Design Patterns. Elements of Reusable Object-Oriented Software.* 1. Amsterdam : Addison-Wesley, 1994. – 395 S. 40

[GHJV95] GAMMA, Erich ; HELM, Richard ; JOHNSON, Ralph ; VLISSIDES, John ; ADDISON-WESLEY (Hrsg.): *Design patterns: elements of reusable object-oriented software*. Addison-Wesley, 1995. – 395 S. http://www.cs.up.ac.za/cs/aboake/sws780/references/patternstoarchitecture/Gamma-DesignPatternsIntro.pdf. – ISBN 0201633612 188

[GLB+86] GREEN, C. ; LUCKHAM, D. ; BALZER, R. ; CHEATHAM, T. ; RICH, C.: Report on a knowledge-based software assistant. (1986), August, 377–428. http://dl.acm.org/citation.cfm?id=31870.31893. ISBN 0–934613–12–5 265

[GP01] GUPTA, Daya ; PRAKASH, Naveen: Engineering Methods from Method Requirements Specifications.

In: *Requirements Engineering* 6 (2001), Nr. 3, 135–160. http://dx.doi.org/10.1007/s007660170001. – DOI 10.1007/s007660170001. – ISSN 09473602 59, 262, 270, 274, 276, 278

[GP07] GONZALEZ-PEREZ, Cesar: Supporting Situational Method Engineering with ISO / lEe 24744 and the Work Product Pool Approach. Version: 2007. http://dx.doi.org/10.1007/978-0-387-73947-2_3. In: RALYTÉ, Jolita (Hrsg.) ; BRINKKEMPER, Sjaak (Hrsg.) ; HENDERSON-SELLERS, Brian (Hrsg.): *Situational Method Engineering Fundamentals and Experiences* Bd. 244. Springer Boston, 2007, 7–18 263

[GRE10] GROHER, Iris ; REDER, Alexander ; EGYED, Alexander: Incremental Consistency Checking of Dynamic Constraints. In: ROSENBLUM, David S. (Hrsg.) ; TAENTZER, Gabriele (Hrsg.): *Fundamental Approaches to Software Engineering* Bd. 6013, Springer, 2010 (Lecture Notes in Computer Science). – ISBN 9783642120282, S. 203–217 266

[Gro09] GRONBACK, Richard C. ; GAMMA, Erich (Hrsg.) ; NACKMAN, Lee (Hrsg.) ; JOHN WIEGAND (Hrsg.): *Eclipse Modeling Project: A Domain-Specific Language (DSL) Toolkit*. Addison-Wesley Professional, 2009. – 736 S. http://www.amazon.com/Eclipse-Modeling-Project-Domain-Specific-Language/dp/0321534077. – ISBN 0321534077 266

[Gru90] GRUHN, Volker: Managing software processes in the environment MELMAC. In: *SIGSOFT Softw. Eng. Notes* 15 (1990), Nr. 6, S. 193—-205 267

[Gru93] GRUBER, Tom R.: A Translation Approach to Portable Ontology Specifications. In: *Knowledge Acquisition* 5 (2) (1993), S. 199–220 39

[Gru02] GRUHN, Volker: Process-Centered Software Engineering Environments, A Brief History and Future Challenges. In: *Annals of Software Engineering* 14 (2002), Nr. 1-4, 363–382. http://dx.doi.org/10.1023/A:1020522111961. – DOI 10.1023/A:1020522111961. – ISSN 10227091 63, 64, 265

[GS07] GOMAA, Hassan Gomaa H. ; SHIN, Michael E Shin Michael E.: *Automated Software Product Line Engineering and Product Derivation*. http://dx.doi.org/10.1109/HICSS.2007.95. Version: 2007 72

[GSC+04] GREENFIELD, Jack ; SHORT, Keith ; COOK, Steve ; KENT, Stuart ; CRUPI, John: *Software Factories: Assembling Appli-*

cations with Patterns, Models, Frameworks, and Tools. 1. Wiley, 2004. – 696 S. 13

[GT98] GEORGAKOPOULOS, Dimitrios ; TSALGATIDOU, Aphrodite: Technology and Tools for Comprehensive Business Process Lifecycle Management. In: DOÄ§AÇ, Asuman (Hrsg.) ; KALINICHENKO, Leonid (Hrsg.) ; ÖZSU, M. T. (Hrsg.) ; SHETH, Amit (Hrsg.): *Workflow Management Systems and Interoperability*. Berlin, Heidelberg : Springer Berlin Heidelberg, 1998. – ISBN 978–3–642–63786–5 42

[Har97a] HARMSEN, Anton F.: *Situational Methhod Engineering*, University of Twente, Diss., 1997 23

[Har97b] HARMSEN, Frank: *Situational Method Engineering*, Universiteit Twente, Diss., 1997. http://dx.doi.org/10.1007/s00766-005-0019-0. – DOI 10.1007/s00766–005–0019–0. – 359–368 S 54, 56, 58, 269

[Hau05] HAUMER, Peter: *Eclipse Process Framework Composer Part 1: Key Concepts*. http://www.ibm.com/developerworks/rational/library/dec05/haumer/. Version: 2005 49, 322

[HB95] HARMSEN, F ; BRINKKEMPER, S: *Design and implementation of a method base management system for a situational CASE environment*. http://dx.doi.org/10.1109/APSEC.1995.496992. Version: 1995 269

[HB01] HENNINGER, Scott ; BAUMGARTEN, Kurt: A Case-Based Approach to Tailoring Software Processes. In: *Proc 4th International Conference on CBR ICCBR01* (2001), 249. http://www.springerlink.com/content/620dmf7xc88k6tbr 262

[HBO94] HARMSEN, Frank ; BRINKKEMPER, Sjaak ; OEI, Han: Situational Method Engineering for Information System Project Approaches. Version: 1994. http://www.csa.com/partners/viewrecord.php?requester=gs&collection=TRD&recid=0147219CI. In: VERRIJN-STUART, A A. (Hrsg.) ; OLLE, William T. (Hrsg.): *Methods and Associated Tools for the Information Systems Life Cycle*. Elsevier Science Inc., 1994 (September). – ISBN 0444820744, 169–194 52, 53, 54, 56

[HC93] HAMMER, Michael ; CHAMPY, James ; HAMMER, Michael (Hrsg.): *Reengineering the Corporation: A Manifesto for Business Revolution*. Harper Business, 1993. – 90–91 S. http://dx.doi.org/10.1016/

S0007-6813(05)80064-3. http://dx.doi.org/10.1016/S0007-6813(05)80064-3. – ISBN 0060559535 41

[HF92] HUMPHREY, Watts ; FEILER, Peter H.: Software process development and enactment: Concepts and definitions / SEI Institute, Pittsburgh, USA. 1992 (SEI-92-TR-4). – Forschungsbericht 51

[HKS06] HORROCKS, Ian ; KUTZ, Oliver ; SATTLER, Ulrike: The Even More Irresistible SROIQ. In: *Proc.of the 10th Int. Conf. on Principles of Knowledge Representation and Reasoning (KR˜2006)*, AAAI Press, 2006. – ISBN 978–1–57735–271–6, 57–67 39

[HMG11] HEGDE, Rajeshwari ; MISHRA, Geetishree ; GURUMURTHY, K. S.: An Insight into the Hardware and Software Complexity of ECUs in Vehicles. In: WYLD, David C. (Hrsg.) ; WOZNIAK, Michal (Hrsg.) ; CHAKI, Nabendu (Hrsg.) ; MEGHANATHAN, Natarajan (Hrsg.) ; NAGAMALAI, Dhinaharan (Hrsg.): *Advances in Computing and Information Technology* Bd. 198. Berlin, Heidelberg : Springer Berlin Heidelberg, 2011 (Communications in Computer and Information Science). – ISBN 978–3–642–22554–3, 99–106 13

[HMPR04] HEVNER, A ; MARCH, S ; PARK, J ; RAM, S: Design Science in Information Systems. In: *MIS Quarterly 28* 1 (2004), S. 75–105 17, 231

[HO92] HEYM, M ; OSTERLE, H: A semantic data model for methodology engineering. In: *1992 Proceedings of the Fifth International Workshop on ComputerAided Software Engineering*, 1992. – ISBN 0818629606 269

[Hol95] HOLLINGSWORTH, David: *The Workflow Reference Model*. Workflow Management Coalition Specification, Document Number TC00-1003. http://www.wfmc.org/standards/docs/tc003v11.pdf. Version: 1995 45, 322

[Hon08] HONKE, Benjamin: *Avoidance of inconsistencies during the virtual integration of vehicle software*, University if Augsburg, Diploma Thesis, 2008. http://www.informatik.uni-augsburg.de/de/lehrstuehle/swt/vs/publikationen/reports/2012-08/TR-2012-08-AUTOSAR.pdf. – 192 S. 15

[HR07] HONKE, Benjamin ; ROSER, Stephan: *Workflow Generation Framework* . http://sourceforge.net/projects/wf-codegen/. Version: 2007 47

[HSGP08] HENDERSON-SELLERS, Brian ; GONZALEZ-PEREZ, Cesar: Comparison of Method Chunks and Method Fragments

for Situational Method Engineering. In: *Software Engineering 2008 ASWEC 2008 19th Australian Conference on*, IEEE Computer Society, 2008. – ISBN 9780769531007, 479–488 58

[HSR10] Henderson-Sellers, B ; Ralyté, J: Situational Method Engineering: State-of-the-Art Review. In: *Journal Of Universal Computer Science* 16 (2010), Nr. 3, 424–478. http://www.jucs.org/jucs_16_3/situational_method_engineering_state/jucs_16_03_0424_0478_henderson.pdf 51, 53, 54, 55, 56, 58, 60, 262, 322

[HT06] Hailpern, B. ; Tarr, P.: Model-driven development: The good, the bad, and the ugly. In: *IBM Systems Journal* 45 (2006), Nr. 3, 451–461. http://dx.doi.org/10.1147/sj.453.0451. – DOI 10.1147/sj.453.0451. – ISSN 0018–8670 13

[IBM10] *IBM Rational Method Composer*. http://www-01.ibm.com/software/awdtools/rmc/. Version: 2010 108

[{IEEE Computer Society}90] {IEEE Computer Society}: *IEEE SA - 610.12-1990 - IEEE Standard Glossary of Software Engineering Terminology*. http://standards.ieee.org/findstds/standard/610.12-1990.html. Version: 1990 134, 212

[ISO02] ISO: Standard for the Exchange of Product model data, ISO standard 10303 / International Organization for Standardization. 2002. – Forschungsbericht 23

[ISO05] ISO: *Family of standards for quality management systems, ISO standard 9000*. http://www.iso.org/iso/iso$\delimiter"026E30F$_catalogue/catalogue$\delimiter"026E30F$_tc/catalogue$\delimiter"026E30F$_detail.htm?csnumber=42180. Version: 2005 108

[ISO07] ISO/IEC: *Software Engineering. Metamodel for Development Methodologies*. 2007 52, 58

[ISO08a] ISO: ISO/IEC 15504 International Standard Information Technology - Software Process Assessment: Part 1-Part 7 / International Organization for Standardization. 2008. – Forschungsbericht 108

[ISO08b] ISO: Systems and software engineering - Software life cycle processes, ISO/IEC 12207 / International Organization for Standardization. 2008. – Forschungsbericht 52

[ISO10] ISO: Road vehicles – Functional safety, ISO standard 26262 / International Organization for Standardization. 2010. – Forschungsbericht 31, 74, 108

[Jaz97] The design of a next-generation process language. Version: 1997. http://dx.doi.org/10.1007/3-540-63531-9. In: JAZAYERI, Mehdi (Hrsg.) ; SCHAUER, Helmut (Hrsg.): *Software Engineering âĂŤ ESEC/FSE'97* Bd. 1301. Berlin, Heidelberg : Springer Berlin Heidelberg, 1997. – DOI 10.1007/3–540–63531–9. – ISBN 978–3–540–63531–4, 142–158 266

[JB96] JABLONSKI, Stefan ; BUSSLER, Christoph: *Workflow Management - Modeling Concepts, Architecture and Implementation*. Thomson Computer Press, 1996 41, 43, 112

[JEA+07] JORDAN, Diane ; EVDEMON, John ; ALVES, Alexandre ; ARKIN, Assaf ; ASKARY, Sid ; BARRETO, Charlton ; BLOCH, Ben ; CURBERA, Francisco ; FORD, Mark ; GOLAND, Yaron ; GUIZAR, Alejandro ; KARTHA, Neelakantan ; LIU, Canyang K. ; KHALAF, Rania ; KÖNIG, Dieter ; MARIN, Mike ; MEHTA, Vinkesh ; THATTE, Satish ; RIJN, Danny van d. ; YENDLURI, Prasad ; YIU, Alex: *Web Services Business Process Execution Language Version 2.0*. OASIS Standard. http://docs.oasis-open.org/wsbpel/2.0/OS/wsbpel-v2.0-OS.pdf. Version: April 2007 45, 47

[Jen03] JENSEN, Kurt: Coloured Petri nets: status and outlook. (2003), Juni, 1–2. http://dl.acm.org/citation.cfm?id=1760066.1760068. ISBN 3–540–40334–5 49

[JM05] JAUFMAN, Olga ; MUNCH, Jurgen: Acquisition of a Project-Specific Process. In: BOMARIUS, Frank (Hrsg.) ; KOMI-SIRVIÖ, Seija (Hrsg.): *6th International Conference on Product Focused Software Development and Process Improvement PROFES 05* Bd. 3547, Springer Berlin Heidelberg, 2005 (LNCS). – ISBN 9783540262008, 328–342 264, 274

[JPJ+93] JARKE, Matthias ; POHL, Klaus ; JACOBS, Stephan ; BUBENKO JR., Janis A. ; ASSENOVA, Petia ; HOLM, Peter ; WANGLER, Benkt ; ROLLAND, Colette ; PLIHON, Véronique ; SCHMITT, Jean-Roch ; SUTCLIFFE, Alistair G. ; JONES, Sara ; MAIDEN, Neil A. ; TILL, David ; VASSILIOU, Yannis ; CONSTANTOPOULOS, Panos ; SPANOUDAKIS, Giorgios: Requirements Engineering: An Integrated View of Representation, Process, and Domain. (1993), September, 100–114. http://dl.acm.org/citation.cfm?id=645384.651477. ISBN 3–540–57209–0 269

[JR91] JENSEN, Kurt ; ROZENBERG, Grzegorz: *High-level Petri nets: theory and application*. 1991 http://dl.acm.org/citation.cfm?id=127935. – ISBN 3–540–54125–x 49

[JVN06] JANSEN-VULLERS, M ; NETJES, M: Business Process Simulation - A Tool Survey. In: *Management* 834 (2006), Nr. 1-2, 77–96. http://dx.doi.org/10.1.1.87.8291. – DOI 10.1.1.87.8291. – ISBN 1463715091098 100

[JWT10] *Eclipse Foundation, Java Workflow Tooling*. http://www.eclipse.org/jwt/. Version: 2010 46

[Kar04] KARLSSON, F: Method configuration: adapting to situational characteristics while creating reusable assets. In: *Information and Software Technology* 46 (2004), Nr. 9, 619–633. http://dx.doi.org/10.1016/j.infsof.2003.12.004. – DOI 10.1016/j.infsof.2003.12.004. – ISBN 3536121357 262

[KBA02a] KURTEV, I ; BEZIVIN, J ; AKSIT, M.pp: Technological Spaces: an Initial Appraisal. In: *Proceedings of the Confed. International Conferences CoopIS, DOA and ODBASE*. Irvine, CA, USA, 2002 268

[KBA02b] KURTEV, Ivan ; BÉZIVIN, Jean ; AKSIT, Mehmet: Technological spaces: An initial appraisal. In: *Language* 2002 (2002), 1–6. http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.109.332 35

[KCH+90] KANG, K C. ; COHEN, S G. ; HESS, J A. ; NOVAK, W E. ; PETERSON, A S. ; NOWAK, W ; PETERSON, S: Feature-Oriented Domain Analysis (FODA) Feasibility Study. (1990). http://www.citeulike.org/user/rmic/article/6565373 68, 82, 90

[KDS07] KORNYSHOVA, Elena ; DENECKÈRE, Rébecca ; SALINESI, Camille: Method Chunks Selection by Multicriteria Techniques: an Extension of the Assembly-based Approach. Version: 2007. http://dx.doi.org/10.1007/978-0-387-73947-2. In: RALYTÉ, Jolita (Hrsg.) ; BRINKKEMPER, Sjaak (Hrsg.) ; HENDERSON-SELLERS, Brian (Hrsg.): *Situational Method Engineering: Fundamentals and Experiences* Bd. 244. Boston, MA : Springer US, 2007. – DOI 10.1007/978–0–387–73947–2. – ISBN 978–0–387–73946–5, 64–78 61

[Ken02] KENT, Stuart: Model Driven Engineering. In: *Integrated Formal Methods* (2002), S. 286–298 35

[Kes09] KESKIN, Ugur: In-vehicle communication networks: a literature survey. In: *Computer Science Report* (2009). http://alexandria.tue.nl/repository/books/652514.pdf 13

[KKB96] KUENG, Peter ; KAWALEK, Peter ; BICHLER, Peter: How to compose an object-oriented business process model? In: *Proceedings of the IFIP WG8.1/WG8.2 Working Conferencec*, 1996 42

[KKGL10] KÜHNE, Stefan ; KERN, Heiko ; GRUHN, Volker ; LAUE, Ralf: Business process modeling with continuous validation. In: *Journal of Software Maintenance and Evolution Research and Practice* 22 (2010), Nr. 22, 547–566. http://dx.doi.org/10.1002/smr.517. – DOI 10.1002/smr.517. – ISSN 15320618 100

[KKL+98] KANG, Kyo C. ; KIM, Sajoong ; LEE, Jaejoon ; KIM, Kijoo ; SHIN, Euiseob ; HUH, Moonhang: FORM: A feature-oriented reuse method with domain-specific reference architectures. In: *Annals of Software Engineering* 5 (1998), Nr. 1, 143–168. http://dx.doi.org/10.1023/A:1018980625587. – DOI 10.1023/A:1018980625587. – ISSN 1573–7489 68

[KKSB06] KIM, Young-Gab Kim Young-Gab ; KIM, Jin-Woo Kim Jin-Woo ; SHIN, Sung-Ook Shin Sung-Ook ; BAIK, Doo-Kwon Baik Doo-Kwon: *Managing Variability for Software Product-Line*. http://dx.doi.org/10.1109/SERA.2006.45. Version: 2006 69

[KLD02] KANG, K C. ; LEE, Jaejoon Lee J. ; DONOHOE, P: *Feature-oriented product line engineering*. http://dx.doi.org/10.1109/MS.2002.1020288. Version: 2002 66, 69, 70

[KRPP10] KOLOVOS, Dimitrios ; ROSE, Louis ; PAIGE, Richard ; POLACK, Fiona: *The Epsilon Book*. Eclipse Generative Modeling Technologies (GMT) project, 2010 41, 166

[KS95] KRAUT, Robert E. ; STREETER, Lynn A.: Coordination in software development. In: *Communications of the ACM* 38 (1995), März, Nr. 3, 69–81. http://dx.doi.org/10.1145/203330.203345. – DOI 10.1145/203330.203345. – ISSN 00010782 266

[KS98] KOTONYA, Gerald ; SOMMERVILLE, Ian: Requirements Engineering : Processes and Techniques (Worldwide Series in Computer Science). In: *Star* (1998), 294. http://www.amazon.com/Requirements-Engineering-Processes-Techniques-Worldwide/dp/0471972088. ISBN 0471972088 104

[KSP+09] KILLISPERGER, Peter ; STUMPTNER, Markus ; PETERS, Georg ; GROSSMANN, Georg ; STÜCKL, Thomas: Meta

Model Based Architecture for Software Process Instantiation. In: WANG, Qing (Hrsg.) ; GAROUSI, Vahid (Hrsg.) ; MADACHY, Raymond (Hrsg.) ; PFAHL, Dietmar (Hrsg.): *International Conference on Software Process* Bd. 5543, Springer Berlin Heidelberg, 2009 (Lecture Notes in Computer Science May). – ISBN 9783642016790, 63–74 262

[KTW02] KIESNER, Christiane ; TAENTZER, Gabriele ; WINKELMANN, Jessica: Visual OCL: A Visual Notation of the Object Constraint Language. In: *Available Online URL htt ptfs cs tuberlin devocl* (2002). http://user.cs.tu-berlin.de/~vila/www_ss04/Papers/VOCLSpec2.pdf 166, 266

[KW92] KUMAR, K ; WELKE, R J.: Methodology Engineering R: a proposal for situation-specific methodology construction. In: COTTERMAN, W W. (Hrsg.) ; SENN, J A. (Hrsg.) ; John Wiley & Sons, Inc. (Veranst.): *Challenges and strategies for research in systems development*. John Wiley & Sons, Inc., 1992, S. 269 54, 56

[Laf96] LAFFRA, Chris: *Advanced Java: Idioms, Pitfalls, Styles and Programming Tips*. Prentice Hall Ptr, 1996. – 270 S. 40

[Lau10] LAUTENBACHER, Florian: *Semantic Business Process Modeling - Principles, Design Support and Realization*, University of Augsburg, Ph.D. Thesis, 2010. – 358 S. 38, 42, 46, 110

[Lid11] LIDDLE, Stephen W.: Model-Driven Software Development. Version: 2011. http://dx.doi.org/10.1007/978-3-642-15865-0. In: EMBLEY, David W. (Hrsg.) ; THALHEIM, Bernhard (Hrsg.): *Handbook of Conceptual Modeling*. Berlin, Heidelberg : Springer Berlin Heidelberg, 2011. – DOI 10.1007/978–3–642–15865–0. – ISBN 978–3–642–15864–3, 17–54 35, 322

[LKL10] LEE, Jihyun ; KANG, Sungwon ; LEE, Danhyung: a Comparison of Software Product Line Scoping Approaches. In: *International Journal of Software Engineering and Knowledge Engineering* 20 (2010), Nr. 05, 637. http://dx.doi.org/10.1142/S021819401000489X. – DOI 10.1142/S021819401000489X. – ISSN 02181940 70

[Lon93] LONCHAMP, J: A Structured Conceptual and Terminological Framework for Software Process Engineering. In: *1993 Proceedings of the Second International Conference on the Software ProcessContinuous Software Process Improvement* (1993), 41–53. http://dx.doi.org/10.1109/SPCON.1993.236823. – DOI 10.1109/SPCON.1993.236823. ISBN 0818636009 42

[LQA+] LIMA REIS, C.A. ; QUITES REIS, R. ; ABREU, M. ; SCHLEBBE, H. ; NUNES, D.J.: *Flexible software process enactment support in the APSEE model*. IEEE Comput. Soc. – 112–121 S. http://dx.doi.org/10.1109/HCC.2002.1046363. http://dx.doi.org/10.1109/HCC.2002.1046363. – ISBN 0–7695–1644–0 43

[LT09] LIGGESMEYER, Peter ; TRAPP, Mario: Trends in Embedded Software Engineering. In: *IEEE Software* 26 (2009), Mai, Nr. 3, 19–25. http://dx.doi.org/10.1109/MS.2009.80. – DOI 10.1109/MS.2009.80. – ISSN 0740–7459 13

[Mö02] MÜNCH, Jürgen: *Process Modeling*. Kaiserslautern, 2002 50

[MA02] MUTHIG, Dirk ; ATKINSON, Colin: Model-Driven Product Line Architectures. In: *Software Product Lines* 2379 (2002), 110–129. http://dx.doi.org/10.1007/3-540-45652-X_8. – DOI 10.1007/3–540–45652–X_8. ISBN 3540439854 66

[MAE12] MAENAD: *Model-based Analysis & Engineering of Novel Architectures for Dependable Electric Vehicles*. http://www.maenad.eu/. Version: 2012 30, 32, 322

[MC94] MALONE, Thomas W. ; CROWSTON, Kevin: The interdisciplinary study of coordination. In: *ACM Comput. Surv.* 26 (1994), Nr. 1, S. 87—-119 44

[MD06] MOOR, A de ; DELUGACH, H ; KROGSTIE, J (Hrsg.) ; HALPIN, T A. (Hrsg.) ; PROPER, H A (. (Hrsg.): *Software Process Validation: Comparing Process and Practice Models*. 2006 50

[MDK99] MONTANGERO, Carlo ; DERNIAME, Jean-Claude ; KABA, Ali B.: The software process: Modelling and technology. Version: 1999. http://www.springerlink.com/content/dj72r21120r04475. In: MONTENEGRO, Carlo (Hrsg.): *Software process principles methodology and Technology* Bd. 1500. Springer-Verlag, 1999. – ISBN 3540655166, 1–13 56

[MFJ05] MULLER, Pierre-Alain ; FLEUREY, Franck ; JÉZÉQUEL, Jean-Marc: Weaving executability into object-oriented meta-languages. Version: Oktober 2005. http://dx.doi.org/10.1007/11557432. In: BRIAND, Lionel (Hrsg.) ; WILLIAMS, Clay (Hrsg.): *Model Driven Engineering Languages and Systems* Bd. 3713. Berlin, Heidelberg : Springer Berlin Heidelberg, Oktober 2005. – DOI 10.1007/11557432. – ISBN 978–3–540–29010–0, 264–278 267

[MHAK08] MOAVEN, Shahrouz ; HABIBI, Jafar ; AHMADI, Hamed ; KAMANDI, Ali: Towards an architecture-centric approach for method engineering. (2008), Februar, 74–79. http://dl.acm.org/citation.cfm?id=1722603.1722618. ISBN 978–0–88986–716–1 262

[MID09a] *Die MID ModellierungsMethodik M3 EJB*. http://www.mid.de/fileadmin/mid/PDF/Poster/Poster_MID_M3EJB_01.pdf. Version: 2009 253

[MID09b] *Die MID ModellierungsMethodik M3 SOA*. http://www.mid.de/fileadmin/mid/PDF/Poster/Poster_MID_M3SOA_01.pdf. Version: 2009 253

[MID09c] *MID ModellierungsMethodik M3 für EE*. http://www.mid.de/fileadmin/mid/PDF/Poster/Poster_MID_M3EE_01.pdf. Version: 2009 253

[Mil99] MILNER, Robin: *Communicating and Mobile Systems: the Pi-Calculus*. Bd. 13. Cambridge University Press, 1999. – 161 S. http://dx.doi.org/10.1108/00251740310499555. http://dx.doi.org/10.1108/00251740310499555. – ISBN 0521658691 50

[Mil10] MILANOVIĆ MILAN: *Modeling Rule-Driven Service Oriented Architectures*, University of Belgrade, Ph.D., 2010. http://milan.milanovic.org/papers/ModelingRule-DrivenServiceOrientedArchitectures-PhD.pdf. – 270 S. 145

[MIS98] MISRA CONSORTIUM: *Guidelines for the use of the programming language C in vehicle based systems*. http://www.misra.org.uk/. Version: 1998 40

[MKP98] MARTTIIN, Pentti ; KOSKINEN, Minna ; PENTTI MARTTIIN, Minna K.: Similarities And Differences Of Method Engineering And Process Engineering Approaches. In: *Process Engineering* (1998), 1991—-1998. http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.33.8115 54

[Mos92] MOSSES, Peter D.: *Cambridge Tracts in Theoretical Computer Science*. Bd. 26: *Action Semantics*. Cambridge University Press, 1992 36

[MOS11] MOST: *Marrying Ontology and Software Technology; European Commission Information and Communication Technologies research project in Seventh Research Framework Programme*. https://217.74.68.230/index.php. Version: 2011 268, 276

[MP07] METZGER, Andreas ; POHL, Klaus: *Variability Management in Software Product Line Engineering*. http://dx.doi.org/10.1109/ICSECOMPANION.2007.83. Version: 2007 (ICSE '06) 66

[MPRC08] MONTERO, Ildefonso ; PENA, Joaquin ; RUIZ-CORTES, Antonio: *From Feature Models to Business Processes*. IEEE, 2008. – 605–608 S. http://dx.doi.org/10.1109/SCC.2008.130. http://dx.doi.org/10.1109/SCC.2008.130. – ISBN 978–0–7695–3283–7 264, 274

[MPSH08] MOTIK, Boris ; PATEL-SCHNEIDER, Peter F. ; HORROCKS, Ian: *{OWL} 2 {W}eb {O}ntology {L}anguage: Structural Specification and Functional-Style Syntax*. W3C Working Draft. http://www.w3.org/TR/owl2-syntax/,asof2008-04-16. Version: April 2008 39

[MR05] MIRBEL, Isabelle ; RALYTE, Jolita: *Situational method engineering: combining assembly-based and roadmap-driven approaches*. Secaucus, NJ, USA : Springer-Verlag New York, Inc., 2005. – 58–78 S. http://dx.doi.org/http://dx.doi.org/10.1007/s00766-005-0019-0. http://dx.doi.org/http://dx.doi.org/10.1007/s00766-005-0019-0. ISSN 0947–3602 23, 58, 87, 281

[MR12] MATINNEJAD, Reza ; RAMSIN, Raman: An Analytical Review of Process-Centered Software Engineering Environments. In: *2012 IEEE 19th International Conference and Workshops on Engineering of Computer-Based Systems*, IEEE, April 2012. – ISBN 978–1–4673–0912–7, 64–73 63, 265

[MSMR09] MACIEL, R S P. ; SILVA, B C D. ; MAGALHAES, P F. ; ROSA, N S.: *An Integrated Approach for Model Driven Process Modeling and Enactment*. http://dx.doi.org/10.1109/SBES.2009.18. Version: 2009 267, 276

[Nau63] NAUR, P: Revised Report on the Algorithmic Language ALGOL 60. In: *Communications of the ACM* 6 (1963), Nr. 1, 1–17. http://www.masswerk.at/algol60/report.htm 139

[NCLMA99] NAU, Dana ; CAO, Yue ; LOTEM, Amnon ; MUNOZ-AVILA, Hector: SHOP: Simple Hierarchical Ordered Planner. In: *Systems Research* 2 (1999), 1–6. http://www.cs.umd.edu/projects/shop/. ISBN 1558606130 95

[Nic] NICK RUSSELL, Arthur H. M. Ter H.: Workflow data patterns. http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.80.6634 44

[NR08] NIKNAFS, Ali ; RAMSIN, Raman ; BELLAHSÈNE, Zohra (Hrsg.) ; LÉONARD, Michel (Hrsg.): *Lecture Notes in Computer Science*. Bd. 5074: *Computer-Aided Method Engineering: An Analysis of Existing Environments*. Berlin, Heidelberg : Springer Berlin Heidelberg, 2008. – 525–540–540 S. http://dx.doi.org/10.1007/978-3-540-69534-9. http://dx.doi.org/10.1007/978-3-540-69534-9. – ISBN 978–3–540–69533–2 59, 62, 63, 261, 265, 272, 322

[NSSLW05] NAVET, N ; SONG, Y ; SIMONOT-LION, F ; WILWERT, C: *Trends in Automotive Communication Systems*. http://dx.doi.org/10.1109/SBMOMO.1995.509718. Version: 2005 13

[OGC07] OGC (OFFICE OF GOVERNANCE COMMERCE): *ITIL - IT Infrastructure Library*. 2007 108

[OMG03] OMG: *MDA Guide Version 1.0.1*. http://www.omg.org/mda/. http://www.omg.org/docs/omg/03-06-01.pdf. Version: 2003 13, 35

[OMG05] OMG, Object Management G.: *MOF Query, Views, and Transformation (QVT) Adopted Specification*. ptc/05-11-01, November 2005 37

[OMG06a] OMG: *Object Constraint Language, Version 2.0*. http://www.omg.org/spec/OCL/2.0/, 2006 41, 159, 166, 197, 198, 266, 323

[OMG06b] OMG, Object Management G.: *Meta Object Facility (MOF) Core Specification, Version 2.0*. http://www.omg.org/docs/formal/06-01-01.pdf. Version: 2006 36, 118

[OMG07] OMG: *MOF 2.0 / XMI Mapping Specification, v2.1.1*. http://www.omg.org/technology/documents/formal/xmi.htm, 2007 23

[OMG08a] OMG: *Software Process Engineering Meta-Model , version 2.0*. http://www.omg.org/spec/SPEM/2.0/, 2008 47, 48, 49, 52, 57, 74, 145, 262, 266, 322

[OMG08b] OMG: *Systems Modeling Language (OMG SysML\texttrademark)*. http://www.omgsysml.org, November 2008 23

[OMG11a] OMG, Object Management G.: *Unified Modeling Language (UML) Specification: Infrastructure Version 2.4.1*. http://www.omg.org/spec/UML/2.4.1/Infrastructure/PDF. Version: 2011 46, 118

[OMG11b] OMG, Object Management G.: *Unified Modeling Language (UML) Specification: Superstructure Version

2.4.1. http://www.omg.org/spec/UML/2.4.1/Superstructure/PDF. Version: 2011 46

[OPE09] OPEN PROCESS FRAMEWORK REPOSITORY ORGANIZATION (OPFRO): *OPEN Process Framework (OPF)*. http://www.opfro.org/. Version: 2009 52, 58, 108

[Ost87] OSTERWEIL, Leon J.: Software processes are software too. In: *International Conference on Software Engineering* 9 (1987), März, 2–13. http://dl.acm.org/citation.cfm?id=41765.41766. ISBN 0–89791–216–0 14, 51

[OT08] OUGIER, F ; TERRIER, F: EDONA: an Open Integration Platform for Automotive Systems Development Tools. In: *ERTS 2008 - Toulouse* (2008) 261

[PBV05] POHL, K ; BÖCKLE, G ; VAN DER LINDEN, F ; SPRINGER-VERLAG (Hrsg.): *Software product line engineering: foundations, principles, and techniques*. Springer, 2005. – 467 S. http://dx.doi.org/10.1007/3-540-28901-1. http://dx.doi.org/10.1007/3-540-28901-1. – ISBN 9783540243724 23, 64, 65, 66, 67, 69, 70, 72, 322

[Pet81] PETERSON, James L.: *Petri Net Theory and the Modeling of Systems*. Prentice-Hall, 1981. – 290 S. 206

[PG07] PRAKASH, Naveen ; GOYAL, S.B: Towards a Life Cycle for Method Engineering,. In: *Eleventh International Workshop on Exploring Modeling Methods in Systems Analysis and Design (EMMSAD'07)*, 2007, S. 27–36 59, 61

[PG08] PRAKASH, Naveen ; GOYAL, S. B.: Method architecture for situational method engineering. In: *2008 Second International Conference on Research Challenges in Information Science*, IEEE, Juni 2008. – ISBN 978–1–4244–1677–6, 325–336 262

[PGR99] PETRIE, Charles ; GOLDMANN, Sigrid ; RAQUET, Andreas: Agent-Based Project Management. Version: 1999. http://www-cdr.stanford.edu/ProcessLink/papers/DPM/dpm.html. In: *Lecture Notes in AI 1600*. Springer-Verlag: Heidelberg, Germany, 1999 209

[PNPS06] PARK, Soojin ; NA, Hoyoung ; PARK, Sooyong ; SUGUMARAN, Vijayan: A semi-automated filtering technique for software process tailoring using neural network. In: *Expert Systems with Applications* 30 (2006), Nr. 2, 179–189. http://dx.doi.org/10.1016/j.eswa.2005.06.023. – DOI 10.1016/j.eswa.2005.06.023. – ISSN 09574174 262

[PRSH09] POLGÁR, B ; RÁTH, I ; SZATMÁRI, Z ; HORVÁTH, Á: Model-based Integration, Execution and Certification of Development Tool-chains. In: *Model Driven Tool and Process Integration* (2009). http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.158.6881&rep=rep1&type=pdf#page=41 271, 274, 276, 278

[PS92] PEUSCHEL, Burkhard ; SCHÄFER, Wilhelm: Concepts and implementation of a rule-based process engine. In: *Proceedings of the 14th international conference on Software engineering - ICSE '92*. New York, New York, USA : ACM Press, Juni 1992. – ISBN 0897915046, 262–279 267

[PS96] PRAKASH, Naveen ; SABHARWAL, Sangeeta: Building CASE Tools For Methods Represented As Abstract Data Types. In: *OOIS*, 1996 270

[PTB+03] PELEG, Mor ; TU, Samson ; BURY, Jonathan ; CICCARESE, Paolo ; FOX, John ; GREENES, Robert A. ; HALL, Richard ; JOHNSON, Peter D. ; JONES, Neill ; KUMAR, Anand ; MIKSCH, Silvia ; QUAGLINI, Silvana ; SEYFANG, Andreas ; SHORTLIFFE, Edward H. ; STEFANELLI, Mario: Comparing Computer-interpretable Guideline Models: A Case-study Approach. In: *Journal of the American Medical Informatics Association* 10 (2003), Nr. 1, 52–68. http://www.pubmedcentral.nih.gov/articlerender.fcgi?artid=150359&tool=pmcentrez&rendertype=abstract 266

[PWD+99] POHL, Klaus ; WEIDENHAUPT, Klaus ; DÖMGES, Ralf ; HAUMER, Peter ; JARKE, Matthias ; KLAMMA, Ralf: PRIME—toward process-integrated modeling environments. In: *ACM Transactions on Software Engineering and Methodology* 8 (1999), Nr. 4, 343–410. http://dx.doi.org/10.1145/322993.322995. – DOI 10.1145/322993.322995. – ISSN 1049331X 267

[Rad92] RADIO TECHNICAL COMMISSION FOR AERONAUTICS RTCA, European O.: RTCA: Software Considerations in Airbone Systems and Equipment Certification Standard Document no. DO-178B/ED-12B. 1992. – Forschungsbericht 74

[RAH06] RUSSELL, Nick ; AALST, Wil M P Van D. ; HOFSTEDE, Arthur H M.: Exception Handling Patterns in Process-Aware Information Systems. In: *Business* BPM-06-04 (2006), 06–04. http://dx.doi.org/10.1.1.90.4048. – DOI 10.1.1.90.4048 44

[Ral04] RALYTÉ, Jolita: Towards situational methods for information systems development: engineering reusable method chunks. In: *Proceedings of the International Conference on Information Systems Development (ISDâĂŹ04)*, Vilnius Technika, 2004, S. 271–282 60

[RBKJ06] RALYTÉ, Jolita ; BACKLUND, Per ; KÜHN, Harald ; JEUSFELD, Manfred A.: Method chunks for interoperability. In: *Information Systems Journal* 4215 (2006), 339–353. http://dx.doi.org/10.1007/11901181_26. – DOI 10.1007/11901181_26 83

[RBP+90] RUMBAUGH, James ; BLAHA, Michael ; PREMERLANI, William ; EDDY, Frederick ; LORENSEN, William: *Object-Oriented Modeling and Design*. Prentice-Hall, 1990 270

[RDR03] RALYTÉ, Jolita ; DENECKÈRE, Rébecca ; ROLLAND, Colette: Towards a Generic Model for Situational Method Engineering. In: *Advanced Information Systems Engineering* 2681 (2003), Nr. 1, 95–110. http://dx.doi.org/10.1007/3-540-45017-3. – DOI 10.1007/3–540–45017–3. – ISBN 9783540404422 59, 262

[RH06] RUSSELL, Nick ; HOFSTEDE, Arthur H M.: WORKFLOW CONTROL-FLOW PATTERNS A Revised View. In: *Business* 2 (2006), Nr. BPM-06-22, 06–22. http://dx.doi.org/10.1.1.93.6974. – DOI 10.1.1.93.6974. – ISBN 9781424481460 44

[RHE04] RUSSELL, Nick ; HOFSTEDE, Arthur H M. ; EDMOND, David: Workflow resource patterns. In: *Business* 3520 (2004), Nr. 5446, 13–17. http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.77.8969&rep=rep1&type=pdf 44

[RHER07] RACU, Razvan ; HAMANN, Arne ; ERNST, Rolf ; RICHTER, Kai: Automotive Software Integration. In: *2007 44th ACMIEEE Design Automation Conference* (2007), 545–550. http://dx.doi.org/10.1109/DAC.2007.375224. – DOI 10.1109/DAC.2007.375224. – ISBN 9781595936271 24

[RHW+11] ROSA, Marcello L. ; HOFSTEDE, Arthur H M T. ; WOHED, Petia ; REIJERS, Hajo A. ; MENDLING, Jan ; AALST, Wil M P Van D.: *Managing Process Model Complexity via Concrete Syntax Modifications*. http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=5738703. Version: 2011 44

[Rie03] RIEBISCH, Matthias: Towards a More Precise Definition of Feature Models. In: *Modelling Variability for ObjectOriented Product Lines* 22 (2003), Nr. 3, 64–76. http://dx.doi.org/10.1053/euhj.2000.2393. – DOI 10.1053/euhj.2000.2393. ISBN 0769521258 69

[Rol97] ROLLAND, Colette: A Primer For Method Engineering. In: *Proceedings of the INFormatique des ORganisations et Systèmes dâĂŹInformation et de Décision (INFORSID 1997)*. Toulouse, 1997, 10—-13 62, 63

[Rol98] ROLLAND, Colette: A Comprehensive View of Process Engineering. In: *Lecture Notes in Computer Science* 1413 (1998), Nr. c, 1–24. http://www.springerlink.com/index/XVER7FV40KKXMXJG.pdf. ISBN 9783540645566 51, 54

[Rol02] ROLLAND, Colette: A User Centric View of Lyee Requirements. In: FUJITA, Hamido (Hrsg.) ; JOHANNESSON, P (Hrsg.): *New Trends in Software Methodologies, Tools and Techniques*. Tokyo : IOS Press, 2002 60

[Rol09] ROLLAND, C: Method engineering: towards methods as services. In: *Software Process: Improvement and Practice* 14 (2009), Nr. 3, 143–164. http://dx.doi.org/10.1002/spip.416. – DOI 10.1002/spip.416. – ISBN 9783540795872 264

[Rom06] ROMBACH, Dieter: Integrated Software Process and Product Lines. In: *Unifying the Software Process Spectrum*, 2006 262

[Ros08] ROSER, Stephan: *Designing and Enacting Cross-organisational Business Processes: A Model-driven, Ontology-based Approach*, University of Augsburg, Diss., 2008 37

[RP96a] ROLLAND, C. ; PLIHON, V.: Using generic method chunks to generate process models fragments. In: *Proceedings of the Second International Conference on Requirements Engineering*, IEEE Comput. Soc. Press, 1996. – ISBN 0–8186–7252–8, 173–180 60

[RP96b] ROLLAND, Colette ; PRAKASH, Naveen: A proposal for context-specific method engineering. (1996), Januar, 191–208. http://dl.acm.org/citation.cfm?id=278337.278353. ISBN 0–412–79750–X 58, 262

[RPB99] ROLLAND, C ; PRAKASH, N ; BENJAMEN, A: A Multi-Model View of Process Modelling. In: *Requirements Engineering* 4 (1999), S. 169–187 57, 59

[RPR98] ROLLAND, Colette ; PLIHON, Véronique ; RALYTÉ, Jolita: Specifying the Reuse Context of Scenario Method Chunks. In: *CAiSE '98 Proceedings of the 10th International Conference on Advanced Information Systems Engineering*, 1998. – ISBN 3–540–64556–X, 191–218 54, 58, 87, 281

[RR01a] RALYTÉ, Jolita ; ROLLAND, Colette: An Approach for Method Reengineering. In: *Conceptual Modeling - ER 2001* 2224 (2001), Dezember, 471–484. http://dx.doi.org/10.1007/3-540-45581-7. – DOI 10.1007/3–540–45581–7. ISBN 978–3–540–42866–4 58, 59, 60, 61, 322

[RR01b] RALYTÉ, Jolita ; ROLLAND, Colette: An Assembly Process Model for Method Engineering. In: *CAiSE '01 Proceedings of the 13th International Conference on Advanced Information Systems Engineering* (2001), Juni, 267–283. http://dl.acm.org/citation.cfm?id=646089.680064. ISBN 3–540–42215–3 58, 61

[RSM95] ROLLAND, Colette ; SOUVEYET, Carine ; MORENO, Mario: Approach for defining ways-of-working. In: *Information Systems Journal* 20 (1995), Nr. 4, 337–359. http://dx.doi.org/10.1016/0306-4379(95)00018-Y. – DOI 10.1016/0306–4379(95)00018–Y. – ISSN 03064379 54

[RW05] ROZANSKI NICK ; WOODS EÓIN: *Software Systems Architecture: Working With Stakeholders Using Viewpoints and Perspectives*. 1. Addison-Wesley Professional, 2005. – 576 S. http://www.informit.com/store/product.aspx?isbn=0321112296 40

[RWM+11] ROSA, Marcello L. ; WOHED, Petia ; MENDLING, Jan ; HOFSTEDE, Arthur H M T. ; REIJERS, Hajo A. ; AALST, Wil M P Van D.: *Managing Process Model Complexity Via Abstract Syntax Modifications*. http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=6011689. Version: 2011 44

[SA09] SADOVYKH, Andrey ; ABHERVE, Antonin: MDE Project Execution Support via SPEM Process Enactment. In: *Model Driven Tool and Process Integration* (2009), 1–7. http://www.utwente.nl/ewi/ecmda2009/workshops/ECMDA2009-MDTPI.pdf#page=60. – ISSN 09290672 268

[Saa80] SAATY, T L.: Analytic Hierarchy Process. In: *Decision Analysis* 50 (1980), Nr. 1, 579–606. http://dx.doi.org/10.3414/ME10-01-0028. – DOI 10.3414/ME10–01–0028. – ISSN 00261270 94

[SAE03] SAEKI, M: CAME: The first step to automated method engineering. In: *Workshop on Process Engineering for Object-Oriented and Component-Based Development*. Anaheim, 2003 270, 274, 276, 278

[Sae06] SAEKI, Motoshi ; DUBOIS, Eric (Hrsg.) ; POHL, Klaus (Hrsg.): *Lecture Notes in Computer Science*. Bd. 4001: *Configuration Management in a Method Engineering Context*. Berlin, Heidelberg : Springer Berlin Heidelberg, 2006. – 384–398 S. http://dx.doi.org/10.1007/11767138. http://dx.doi.org/10.1007/11767138. – ISBN 978–3–540–34652–4 270

[SAJ02] SÖDERSTRÖM, Eva ; ANDERSSON, Birger ; JOHANNESSON, Paul: Towards a Framework for Comparing Process Modelling Languages. In: *Framework* 2348 (2002), 600–611. http://dx.doi.org/10.1007/3-540-47961-9_41. – DOI 10.1007/3–540–47961–9_41. – ISBN 9783540437383 46

[SB07] SPECIFICATION, O M G A. ; BARS, Change: OMG Unified Modeling Language (OMG UML). In: *Language* (2007), Nr. November, 1 – 212. http://www.omg.org/spec/UML/2.1.2/Infrastructure/PDF 23

[SB10] SAAD, Christian ; BAUER, Bernhard: Data-flow based Model Analysis. In: *Second NASA Formal Methods Symposium (accepted)* (2010) 100, 147

[Sca01] SCACCHI, Walt: Process Models in Software Engineering. Version: 2nd, 2001. http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.22.9145. In: MARCINIAK, John J. (Hrsg.): *Encyclopedia of Software Engineering*. 2nd. New York : John Wiley and Sons, 2001 51, 53

[SCFY96] SANDHU, Ravi S. ; COYNE, E J. ; FEINSTEIN, H L. ; YOUMAN, C E.: Role Based Access Control Models. In: *Computer* 29 (1996), Nr. 2, 38–47. http://dx.doi.org/10.1016/S1363-4127(01)00204-7. – DOI 10.1016/S1363–4127(01)00204–7. – ISSN 13634127 167

[SCO07] SIMIDCHIEVA, Borislava I. ; CLARKE, Lori A. ; OSTERWEIL, Leon J.: Representing Process Variation with a Process Family. In: WANG, Qing (Hrsg.) ; PFAHL, Dietmar (Hrsg.) ; RAFFO, David M. (Hrsg.): *International Conference on Software Process* Bd. 4470, Springer Berlin Heidelberg, 2007 (Lecture Notes in Computer Science). – ISBN 9783540724254, 109–120 263, 274

[SHA10] SHAPE: *Shape Project*. http://www.shape-project.eu, 2010 261

[SHK09] SUNYAEV, Ali ; HANSEN, Matthias ; KRCMAR, Helmut: Method Engineering: A Formal Description. In: *Information Systems Development* (2009), S. 645–654 23

[SIWyS93] SAEKI, Motoshi ; IGUCHI, Kazuhisa ; WEN-YIN, Kuo ; SHINOHARA, Masanori: A Meta-Model for Representing Software Specification & Design Methods. In: *Proceeding Proceedings of the IFIP WG8.1 Working Conference on Information System Development Process* (1993), September, 149–166. http://dl.acm.org/citation.cfm?id=647157.717270. ISBN 0–444–81594–5 58

[SK05] SIEBER, Tanja ; KOVACS, Laszlo: Technical documentation: Terms, problems and challenges in managing data, information and knowledge. In: *5th International Conference of Ph.D. students*. University of Miskolc, Hungary, 2005, S. 165–170 36

[SMV07] SOUSA, Kênia ; MENDONÇA, Hildeberto ; VANDERDONCKT, Jean: Towards Method Engineering of Model-Driven User Interface Development. In: *Work* 4849 (2007), 112–125. http://dx.doi.org/10.1007/978-3-540-77222-4. – DOI 10.1007/978–3–540–77222–4. ISBN 9783540772217 266

[SN09] SCOTT, Bill ; NEIL, Theresa: *Designing Web Interfaces: Principles and Patterns for Rich Interactions*. 1. O'Reilly Media, 2009. – 336 S. 139

[Sol12] SOLTANI, Samaneh: *Towards automated feature model configuration with optimizing the non-functional requirements*, SIMON FRASER UNIVERSITY, Master Thesis, 2012. – 73 S. 78

[Som07] SOMMERVILLE, Ian: *Software Engineering*. Boston : Addison-Wesley, 2007 52

[SOSF04] SADIQ, Shazia ; ORLOWSKA, Maria ; SADIQ, Wasim ; FOULGER, Cameron: Data flow and validation in workflow modelling. (2004), Januar, 207–214. http://dl.acm.org/citation.cfm?id=1012294.1012317 42, 100

[SP06] SCHNIEDERS, Arnd ; PUHLMANN, Frank: Variability Mechanisms in E-Business Process Families. In: *Engineering* LNI 85 (2006), 583–601. http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.83.5010&rep=rep1&type=pdf 264

[SR09] SIM, Susan E. ; RAYCRAFT, Derek J.: Cross-Artifact Traceability Using Lightweight Links Sukanya Ratanotayanon. In: *TEFSE 09 Proceedings of the 2009 ICSE Workshop on Traceability in Emerging Forms of Software Engineering* (2009), 57–64. http://dx.doi.org/10.1109/TEFSE.2009.5069584. – DOI 10.1109/TEFSE.2009.5069584. ISBN 9781424437412 213

[SRF+] SIMMONDS, D. ; REDDY, R. ; FRANCE, R. ; GHOSH, S. ; SOLBERG, A.: *An Aspect Oriented Model Driven Framework*. IEEE. – 119–130 S. http://dx.doi.org/10.1109/EDOC.2005.5. http://dx.doi.org/10.1109/EDOC.2005.5. – ISBN 0–7695–2441–9 38, 110

[SS94] SANDHU, R S. ; SAMARATI, P: Access Control: Principles and Practice. In: *IEEE Communications Magazine* 32 (1994), Nr. 9, 40–48. http://dx.doi.org/10.1109/35.312842. – DOI 10.1109/35.312842. – ISSN 01636804 167

[SSRG96] SI-SAID, Samira ; ROLAND, Colette ; GROSZ, Georges: MENTOR: A Computer Aided Requirements Engineering Environment. In: *Advanced Information Systems Engineering* 1080 (1996), Mai, 22–43. http://dl.acm.org/citation.cfm?id=646084.679556. ISBN 3–540–61292–0 52, 269, 274, 276, 278

[SVB05] SVAHNBERG, Mikael ; VAN GURP, Jilles ; BOSCH, Jan: A taxonomy of variability realization techniques. In: *Software Practice and Experience* 35 (2005), Nr. 8, 705–754. http://dx.doi.org/10.1002/spe.652. – DOI 10.1002/spe.652. – ISSN 00380644 66

[SVC06] STAHL, Thomas ; VOELTER, Markus ; CZARNECKI, Krzysztof: *Model-Driven Software Development: Technology, Engineering, Management*. 1. Wiley, 2006. – 446 S. 13

[SVHB05] SEEMUELLER, Holger ; VOOS, Holger ; HONKE, Benjamin ; BAUER, Bernhard: SITUATIONAL METHOD ENGINEERING APPLIED FOR THE ENACTMENT OF DEVELOPMENT PROCESSES An Agent based Approach. In: *Applied Sciences* (2005), S. 399–405 282

[SW94] SAEKI, Motoshi ; WENYIN, Kuo ; WIJERS, Gerard (Hrsg.) ; BRINKKEMPER, Sjaak (Hrsg.) ; WASSERMAN, Tony (Hrsg.): *Specifying software specification & design methods*. 1994 23

[tim09] *Timing Model Project - TIMMO:(www.timmo.org)*. 2009 33, 322

[TK08] TOLVANEN, Juha-Pekka ; KELLY, Steven: *Domain-Specific*

Modeling: Enabling Full Code Generation. 1. Wiley-IEEE Computer Society, 2008. – 444 S. 231, 270

[Tol98] TOLVANEN, Juha-Pekka: *Incremental Method Engineering with Modeling Tools*, University of Jyväskylä, Finland, Dissertation,, 1998. – 301 S. 60

[Tol06] TOLVANEN, Juha-pekka: MetaEdit + : Integrated modeling and metamodeling environment for domain-specific languages. In: *Proceeding OOPSLA '06 Companion to the 21st ACM SIGPLAN symposium on Object-oriented programming systems, languages, and applications* (2006), 690–691. http://dx.doi.org/10.1145/1176617.1176676. – DOI 10.1145/1176617.1176676. ISBN 159593491X 270, 274, 276, 278

[TZD09] TRAN, Huy ; ZDUN, Uwe ; DUSTDAR, Schahram: Vb-Trace: using view-based and model-driven development to support traceability in process-driven SOAs. In: *Software Systems Modeling* 10 (2009), Nr. 1, 5–29. http://dx.doi.org/10.1007/s10270-009-0137-0. – DOI 10.1007/s10270–009–0137–0. – ISSN 16191366 213

[UG96] USCHOLD, Mike ; GRUNINGER, Michael: Ontologies: Principles, methods and applications. In: *Knowledge Engineering Review* 11 (1996), Nr. 2, 93–136. http://dx.doi.org/10.1109/MIS.2002.999223. – DOI 10.1109/MIS.2002.999223. – ISSN 02698889 39

[Van02] VAN DER LINDEN, F: Software product families in Europe: the Esaps & Cafe projects. In: *IEEE Software* 19 (2002), Nr. 4, 41–49. http://dx.doi.org/10.1109/MS.2002.1020286. – DOI 10.1109/MS.2002.1020286. – ISSN 07407459 65, 69

[Van05] VANDERDONCKT, Jean: A MDA-Compliant Environment for Developing User Interfaces of Information Systems. In: PASTOR, O (Hrsg.) ; FALCAO E CUNHA, J (Hrsg.) ; Springer (Veranst.): *Advanced Information Systems Engineering* Bd. 3520 Springer, Springer, 2005 (Lecture Notes in Computer Science). – ISBN 3540260951, 16–31 266

[VBS01] VAN GURP, J ; BOSCH, J ; SVAHNBERG, M: On the notion of variability in software product lines. In: *Proceedings Working IEEEIFIP Conference on Software Architecture* 01 (2001), Nr. 02, 45–54. http://dx.doi.org/10.1109/WICSA.2001.948406. – DOI 10.1109/WICSA.2001.948406. – ISBN 0769513603 66, 69

[VK02] VAN DEURSEN, Arie ; KLINT, Paul: Domain-Specific Language Design Requires Feature Descriptions. In: *Journal of Computing and Information Technology* 10 (2002), Nr. 1, 1–17. http://dx.doi.org/10.2498/cit.2002.01.01. – DOI 10.2498/cit.2002.01.01. – ISSN 13301136 69

[VSSU06] V. AHO, Alfred ; S. LAM, Monica ; SETHI RAVI ; ULLMAN, Jeffrey D.: *Compilers: Principles, Techniques, and Tools*. 2nd Editio. Addison Wesley, 2006 147

[VTKB03] VAN DER AALST, W. M. P. ; TER HOFSTEDE, A. H. M. ; KIEPUSZEWSKI, B. ; BARROS, A. P.: Workflow Patterns. In: *Distributed and Parallel Databases* 14 (2003), Juli, Nr. 1, 5–51. http://dx.doi.org/10.1023/A:1022883727209. – DOI 10.1023/A:1022883727209. – ISSN 0926–8782 44, 92, 98, 112, 114

[Was06] WASHIZAKI, Hironori: Building Software Process Line Architectures from Bottom Up. Version: 2006. http://dx.doi.org/10.1007/11767718. In: MÜNCH, Jürgen (Hrsg.) ; VIERIMAA, Matias (Hrsg.): *Product-Focused Software Process Improvement* Bd. 4034. Berlin, Heidelberg : Springer Berlin Heidelberg, 2006. – DOI 10.1007/11767718. – ISBN 978–3–540–34682–1, 415–421 263, 274

[WEB+09] WEBER, Sebastian ; EMRICH, Andreas ; BROSCHART, Jörg ; RAS, Eric ; ÜNALAN, Özgür: Supporting Software Development Teams with a Semantic Process- andArtifact-oriented Collaboration Environment. In: *SOFTTEAM'09*. Kaiserslautern, 2009 267, 276

[Wes99] WESTFECHTEL, Bernhard: Models and tools for managing development processes. (1999), Januar. http://portal.acm.org/citation.cfm?id=1744651. ISBN 3–540–66756–3 267

[Wes07] WESKE, Mathias: *Business Process Management - Concepts, Languages, Architectures*. Berlin Heidelberg : Springer Verlag, 2007 41, 42, 45

[WfM99a] WFMC: *Terminology Glossary*. http://www.wfmc.org/standards/docs/TC-1011_term_glossary_v3.pdf. Version: 1999 41

[WfM99b] WFMC: Workflow Management Coalition Terminology & Glossary. In: *Management* 39 (1999), Nr. 3, 1–65. http://dx.doi.org/10.1016/S0019-0578(00)00014-8. – DOI 10.1016/S0019–0578(00)00014–8 84, 113

[WfM05] WFMC, Workflow Management C.: *Process Definition Interface – XML Process Definion Language*. Work-

flow Management Coalition Workflow Standard. http://www.wfmc.org/standards/docs/TC-1025_xpdl_2_2005-10-03.pdf. Version: 2005 45

[Wit00] WITHERS, D H.: Software engineering best practices applied to the modeling process. In: *2000 Winter Simulation Conference Proceedings Cat No00CH37165* Bd. 1, 2000. – ISBN 0780365798, S. 432 –439 vol.1 40

[WK04] WISTRAND, Kai ; KARLSSON, Frederik: Method Components rationale revealed. In: *Advanced Information Systems Engineering*, Springer Berlin Heidelberg, 2004, 189–201 58, 262

[WL99] WEISS, David M. ; LAI, Chi Tau R.: *Software Product-Line Engineering: A Family-Based Software Development Process*. Addison-Wesley Professional, 1999. – 448 S. http://www.citeulike.org/group/1374/article/3945027. – ISBN 0201694387 65

[Xu05] XU, Peng: Knowledge Support in Software Process Tailoring. In: *Proceedings of the 38th Annual Hawaii International Conference on System Sciences* 00 (2005), Nr. C, 87c–87c. http://dx.doi.org/10.1109/HICSS.2005.380. – DOI 10.1109/HICSS.2005.380. – ISBN 0769522688 262

[YLZX10] YONGCHAREON, Sira ; LIU, Chengfei ; ZHAO, Xiaohui ; XU, Jiajie: An artifact-centric approach to generating web-based business process driven user interfaces. (2010), Dezember, 419–427. http://dl.acm.org/citation.cfm?id=1991336.1991383. ISBN 3–642–17615–1, 978–3–642–17615–9 266

[Zam01] ZAMLI, Kamal Z.: Process Modeling Languages: A Literature Review. In: *Malaysian Journal of Computer Science* 14 (2001), Nr. 2, S. 26–37 63

[ZCO07] ZHAO, Gansen ; CHADWICK, David ; OTENKO, Sassa: Obligation for Role Based Access Control. (2007). http://kar.kent.ac.uk/23994/ 167

[ZHSF05] ZOWGHI, Didar ; HENDERSON-SELLERS, Brian ; FIRESMITH, Donald G.: Using the OPEN Process Framework to Produce a Situation-Specific Requirements Engineering Method. In: *Proceedings of SREP* (2005). http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.68.1141 52

[Zhu05] ZHU, Hong: *Software Design Methodology*. 1 edition. Elsevier, 2005. – 368 S. 231, 235

[ZL01] Zamli, K Z. ; Lee, P A.: Taxonomy of Process Modeling Languages. In: *Proceedings ACSIEEE International Conference on Computer Systems and Applications* (2001), 435–437. http://dx.doi.org/10.1109/AICCSA.2001.934035. – DOI 10.1109/AICCSA.2001.934035. ISBN 0769511651 266

[ZZHG] Zhu, Hong ; Zhang, Yanlong ; Huo, Qingning ; Greenwood, Sue: Application of hazard analysis to software quality modelling. In: *Proceedings 26th Annual International Computer Software and Applications*, IEEE Comput. Soc. – ISBN 0–7695–1727–7, 139–144 235, 324

List of Figures

1.1 Objectives overview 18
1.2 Overview about the chapters of this thesis 21

2.1 AUTOSAR refined layered software architecture from [AUT12] 25
2.2 Basic AUTOSAR Approach from [AUT12] 26
2.3 AUTOSAR: Package Overview from [AUT12] 28
2.4 Autosar Methodology: Overview from [AUT12] 29
2.5 EAST-ADL Overview from [MAE12] 30
2.6 EAST-ADL Methodology Overview from [MAE12] 32
2.7 Basic TADL2 elements, from [tim09] 33
2.8 Meta modeling layers, from [Lid11] 35
2.9 Aspect-oriented base model 38
2.10 Lifecycle of Business Process Management 42
2.11 Workflow Reference Model - Components & Interfaces from [Hol95] . . . 45
2.12 Structure of the SPEM 2.0 Meta-Model from [OMG08a] 48
2.13 Separation of Method Content and Processes from [Hau05] 49
2.14 SPEM 2.0's conceptual usage framework from [OMG08a] 49
2.15 Overview: Meta model for software development processes 55
2.16 An overall high-level model of the SME approach [HSR10] 55
2.17 A Method meta model according to [RR01a] 59
2.18 Method reengineering process model from [RR01a] 61
2.19 General architecture of CAME environments from [NR08] 62
2.20 Variation point, variant, and the variability dependency in the variability meta model according to [PBV05] 67
2.21 The software product line engineering framework from [PBV05] 70

3.1 Objectives overview 73
3.2 Overview: Software Process Line Engineering 77
3.3 Software Process Line Engineering: Process Family Engineering 78
3.4 Process Line Engineering Assets 80
3.5 Overview: Process Family Definition Phase 82
3.6 Design Level Correspondence 86
3.7 Characteristics framework 89
3.8 Software Process Line Engineering: Feature Model Generation 90
3.9 Feature Model Generation Rules 91
3.10 Software Process Line Engineering: Situational Process Derivation 93

3.11 Basic Adaption Rules 97
3.12 Example: Advanced Adaption Rule 101
3.13 Concept of the resource-oriented analysis of processes. 102
3.14 Requirements Engineering Feature Model 105

4.1 Objectives overview 107
4.2 Aspect-oriented implementation for semantically enriched process models 111
4.3 Overview: Relationship between Meta Model View and Process 118
4.4 Meta model associations: Relationship between CAssoc and AssocProp . 127
4.5 Method chunk-specific data annotation 132
4.6 Artifact Assignment Propagation Obstacle 135
4.7 Artifact Assignment Combination Obstacle 136
4.8 Artifact Assignment Abstration Obstacle 137
4.9 Master-Detail Meta Model Re-arrangement 139
4.10 Guideline Facets 147
4.11 Guideline Structure 152
4.12 Guideline Elements 155
4.13 Statement Meta Model 156
4.14 Statement Modeling I: association Transition 158
4.15 Statement Modeling II: instance Transition 159
4.16 Guideline Navigation 162
4.17 Instance Identification 163
4.18 Attribute Property 164
4.19 Statement navigation via associations 164
4.20 Multiple instance variables and iterators 165
4.21 Instance attribute relations 166
4.22 Case Study: Meta model 168
4.23 CME Case Study Process 169
4.24 Meta Model View Designer 170
4.25 MMV of the Artifact Functional_Requirements 171
4.26 MMV of the Artifact Combined_Requirements 171
4.27 MMV of the Artifact FA_refined_by_timing 172
4.28 MMV of the Artifact SW_Architecture 172
4.29 Case Study Guideline: Functional requirements must have a name, which corresponds to a particular naming convention 176
4.30 Case Study Pattern: The number functions must be equal to the number of functional requirements 177
4.31 Case Study Guideline: a function's timing value must not be lower than 100 178

5.1 Objectives overview 179
5.2 Example of a Generated Master-Detail Editor 182
5.3 Artifact Observer Mechanism 188
5.4 Artifact Observation 192
5.5 Abstract Syntax Kernel Meta Model for OCL Types from [OMG06a] 197
5.6 The basic structure of the abstract syntax kernel metamodel for Expressions from [OMG06a] 198

5.7 Example Meta Model 200
5.8 Example OCL Abstract Syntax Tree 200
5.9 Instance Identification 201
5.10 Attribute Property Calls 201
5.11 Association Property Calls 202
5.12 Iterator expressions 203
5.13 Conversion to prenex normal form 204
5.14 Crossing attribute relations 205
5.15 Validation Report 209
5.16 Consistency Production Line 211
5.17 Simple Workflow Example 214
5.18 Generated Editor to support the MC: CreateNonFunctionalRequirements 221
5.19 Generated Editor to support the MC: Complete Requirements 223
5.20 Generated Editor to support the MC: Define SW Architecture 224
5.21 Execution of “Create Functional Requirements” 227
5.22 Validation of “Create Functional Requirements” 228
5.23 Execution of “Create Non-Functional Requirements” 229
5.24 Execution of “Complete requirements” 229

6.1 Conceptual Architecture 232
6.2 Plugin structure of the SME4PCL approach 233
6.3 Notation of HASARD quality models according to [ZZHG] 235
6.4 Hazards belonging to SPLE components 237
6.5 Hazards belonging to CME components 237
6.6 Hazards belonging to Guidance components 238
6.7 Evaluation Results for different process models 244
6.8 Feature Model for the M3 Process Line 255
6.9 SOA-specific Variant: Initiation Phase 257
6.10 Artifact-specific Meta Model Views for SOA Initiation 258
6.11 MMV for Requirements Artifact created during the SOA Initiation phase . 259

C.1 Application of HASARD to our architecture 354

E.1 Methodology for Service-Oriented Architectures 357
E.2 Methodology for Enterprise Java Beans 358
E.3 Methodology for Embedded Systems 359
E.4 Methodology for M3 Family 360
E.5 M3 Initiation Variant for EJB Development 361
E.6 M3 Initiation Variant for EE Development 362
E.7 M3 Realize Software Construction Variant for EE Development 363
E.8 SOA Initiation Guidance: Part I 364
E.9 SOA Initiation Guidance: Part II 365
E.10 SOA Initiation Guidance: Part III 366
E.11 SOA Initiation Guidance: Part IV 367

Listings

3.1 Generate workflow semantics 98

5.1 Master Detail Definition 183
5.2 ListViewer Template 184
5.3 Example: Completed ListViewer Template 184
5.4 ReferenceAdd Handler Template 185
5.5 Example: Completed ReferenceAdd Handler 186
5.6 Artifact Influence Table 215
5.7 Handle node for the influence table creation 215
5.8 OCL statement: attribute operation 225
5.9 OCL statement: regular expression 225
5.10 OCL statement: iterator expression 1a 225
5.11 OCL statement: iterator expression 1b 225
5.12 OCL statement: operation call expression 225

A Acronyms

ABPMP Association of Business Process Management Professionals

AD Attribute Detail

ADL Architecture Description Language

AHP Analytical Hierarchy Process

AI Artificial Intelligence

AN Attribute Node

APM Artifact Propagation Mechanism

ARIS Architecture of Integrated Information Systems

ASIL Automotive Safety Integrity Level

ATAM Architecture Tradeoff Analysis Method

ATL ATLAS Transformation Language

AUTOSAR AUTomotive Open System ARchitecture

AUTOSIG Automotive Special Interest Group

BNF Backus Naur Form

BPEL Business Process Execution Language

BPM Business Process Management

BPMS Business Process Management System

BPMN Business Process Modeling Notation

BSE Basic Structural Element

CAME Computer-aided Method Engineering

CASE Computer-aided Software Engineering

CCF Characteristic Concepts Function

CDT Complex Data Type

CFF Characteristic Feature Function

CME Computational Method Engineering

CMMI Capability Maturity Model Integration

COTS Commercial off-the-shelf

DSL Domain-specific Language

E/E Electric/Electronic

EABPM European Association of Business Process Management

EAST-ADL Advancing Traffic Efficiency and Safety through Software Technology

ECU Electronic Control Unit

EMF Eclipse Modeling Framework

EPF Eclipse Process Framework

EPGM Executable Process Guidance Model

EUSA Element Usage Scenario Attribute

EVL Epsilon Validation Language

FAA Function Analysis Architecture

FD Feature Detail

FODA Feature-Oriented Domain Analysis

FPC Flexible-managed Process Component

FSE Functional Structural Elements

GMF Graphical Modeling Framework

HASARD Hazard Analysis of Software ARchitectural Designs

HAZOP Hazard and Operability Studies

HTN Hierarchical Task Network

IEC International Electrotechnical Commission

IEEE Institute of Electrical and Electronics Engineers

ISO International Organization for Standardization

ITIL IT Infrastructure Library

JET Java Emitter Template

jPDL jBPM Process Definition Language

JWT Java Workflow Tooling

KPI Key Performance Indicator

GE Grouping Element

M2M Model-to-Model

M2T Model-to-Text

MAENAD Model-based Analysis & Engineering of Novel Architectures for Dependable Electric Vehicles

MC Method Chunk

ME Method Engineering

MDA Model Driven Architecture

MDE Model Driven Engineering

MDSD Model Driven Software Development

MMTS Meta-modeling Technical Space

MMV Meta Model View

MMVT Meta Model View Tree

MF Method Fragment

MOA Method-oriented Architecture

MOF Meta Object Facility

MUD Model Under Development

OASIS Organization for the Advancement of Structured Information Standards

OCL Object Constraint Language

OEM Original Equipment Manufacturer

OM Object Master

OMG Object Management Group

OTS Ontological Technical Space

OWL Web Ontology Language

PE Process Engineering

PDDL Planning Domain Definition Language

PDL Process Definition Language

PDT Primitive Data Type

PLE Product Line Engineering

PMod Process Model

PPS Predefined Property Set

PSEE Process-Centered Software Engineering Environments

QVT Query View Transformation

RD Reference Detail

RDF Resource Description Framework

RE Requirements Engineering

RF Resource Fragment

RMC Rationale Method Composer

ROnt Resource Ontology

RP Reference Process

RTE Runtime Environment

RUP Rational Unified Process

S-AHP Stratified Analytical Hierarchy Process

SAAM Software Architecture Analysis Method

SAFE Safe Automotive soFtware architEcture

SEI Software Engineering Institute

SHOP2 Simple Hierarchical Ordered Planner 2

SMCG Situational Method-centric Guideline

SME Situational Method Engineering

SME4PCL Situational Method Engineering for Process-Centric Languages

SOA Service-oriented Architecture

SPC Strict-managed Process Component

SPEM Software Process Engineering Metamodel

SPICE Software Process Improvement and Capability Determination

SPLE Software Process Line Engineering

SWT Standard Widget Toolkit

SysML Systems Modeling Language

TADL Timing Augmented Description Language

TIMMO TIMing MOdel - TOols

TS Technical Space

UMA Unified Method Architecture

UML Unified Modeling Language

VDA German Association of the Automotive Industry

VFB Virtual Functional Bus

VFM Vehicle Feature Model

VP Variation Point

VSE Visual Structural Element

WfMC Workflow Management Coalition

WSBPEL Web Service Business process Execution Language

XPDL XML Process Definition Language

XMI XML Metadata Interchange

XML Extensible Markup Language

XWT XML Widget Toolkit

B Translational Statement Semantics

```
1 transformation NewTransformation(in gl : GL, out oclout: OCL);
2 main() {
3   //the root element, i.e. self must have only one outgoing edge
4   gl.rootObjects()[GL::PosClass].outReferences->forEach(ref){
5     ref.oclAsType(GL::InstanceTransition).
6     handleInstanceTransition(null, null);
7   };
8 }
9
10 --Helper function to pass through only
11 helper GL::InstanceTransition::handleInstanceTransition(InputSrc:
12 OCL::expressions::OCLExpression, inputVars:OrderedSet(String)):
13 OCL::expressions::OCLExpression{
14     var srcTMP:=InputSrc.deepclone().oclAsType(OCL::expressions::
15     OCLExpression);
16     --InstanceTransition
17     return self._end.oclAsType(GL::NegClass).handleNegClass
18     (srcTMP,inputVars);
19 }
20
21 --Handle association from individual to navigation concept
22 helper GL::AssociationTransition::handleAssociationTransition
23 (InputSrc:OCL::expressions::OCLExpression,
24 inputVars:OrderedSet(String)): OCL::expressions::OCLExpression{
25     --Handle set asssociations->Must be transformed to an
26     --IteratorExpression
27     if(self.oclAsType(GL::AssociationTransition).objectRef.upperBound
28     <> 1)then{
29     --Case: Association makes iterator expression over set of
30     --objects with forAll, exists, reject, collect, or select
31       if(self.iterationType.repr() = "forAll" or
32       self.iterationType.repr() = "exists" or
33       self.iterationType.repr() = "select" or
34       self.iterationType.repr()='collect'or
35       self.iterationType.repr()='reject')then{
36         --Make the resulting IteratorExpression
37         var res :OCL::expressions::IteratorExp =
38         new OCL::expressions::IteratorExp();
39         --Make all iterator variables, i.e. for all
40         --associated individuals
41         var iterators :OrderedSet(OCL::expressions::Variable);
```

```
42        self._end.outReferences->forEach(iterator|
43        iterator.oclIsTypeOf(GL::InstanceTransition)){
44           iterators += iterator._end.makeVariable();
45        };
46        --Set body expression
47        res.body := self._end.oclAsType(GL::NavigationNode).
48        handleNavigationNodeForSet(null,inputVars);
49        --Set source expression
50        if(InputSrc= null or InputSrc.oclIsKindOf
51        (OCL::expressions::NullLiteralExp)) then{
52          res.source := self.reference2PropCallExp();
53        }
54        else{
55          var srcTMP:=InputSrc.deepclone().
56          oclAsType(OCL::expressions::OCLExpression);
57        }endif;
58
59        --Set iterator variables
60        iterators->forEach(iter){
61        res.iterator += iter;
62        };
63        //Set name of the iterator, i.e. the type of iteration
64        --(forAll, exists, collect, select,..)
65        res.setName(self.iterationType.repr());
66        --Determine whether iterator expression result will be set or
67        -- Boolean value
68        if(self.iterationType.repr()='select'or
69          self.iterationType.repr()='collect'or
70          self.iterationType.repr()='reject')then{
71          //Iterator result is a set
72          res.setType(makeSet());
73          }
74        else {
75          //Iterator result is a boolean value
76          res.setType(makeBoolean());
77        }endif;
78        return res;
79        }
80        --Case: Association makes iterator expression over set of
81        --objects with includes.
82        --For this a specific element  must be selected from the set
83        else if(self.iterationType.repr() = "includes" or
84        self.iterationType.repr() = "includesAll")then{
85        --An OperationCall expression must be used  in
86        --OCL for includes and includesAll
87        --The operation is used the actual context as source and
88        -- an iterator expression to "select" the element(s)
89        --which should be checked
90        var res :OCL::expressions::OperationCallExp =
91        new OCL::expressions::OperationCallExp();
92
```

```
93  --Make the Selection iterator expression
94  var argument :OCL::expressions::IteratorExp =
95  new OCL::expressions::IteratorExp();
96  --Make all iterator variables, i.e.
97  --for all associated individuals
98  var iterators :OrderedSet(OCL::expressions::Variable);
99  self._end.outReferences->forEach(iterator|
100 iterator.oclIsTypeOf(GL::InstanceTransition)){
101   iterators += iterator._end.makeVariable();
102 };
103 --Set iterator variables
104 iterators->forEach(iter){
105 argument.iterator += iter;
106 };
107 --Set body expression for the selection part of
108 --the includes operation
109 var expressions :Sequence(OCL::expressions::OCLExpression);
110 expressions += self._end.oclAsType(GL::NavigationNode).
111 handleNavigationNodeForSet(null,inputVars);
112 --Combine the expression with a type check
113 expressions += self._end.outReferences->first()._end.
114 oclAsType(GL::NegClass).maketypeValidationExpression();
115 var combinedExpressions := expressions->
116 makeCompositeOpCallAND();
117 argument.body := combinedExpressions;
118 --Set source expression
119  if(InputSrc= null or InputSrc.oclIsKindOf
120  (OCL::expressions::NullLiteralExp))then{
121   argument.source := self.reference2PropCallExp();
122  }
123  else{
124   var srcTMP:=InputSrc.deepclone().
125   oclAsType(OCL::expressions::OCLExpression);
126   argument.source := self.reference2PropCallExp(srcTMP);
127  }endif;
128  --Set name of the iterator expression,
129  --i.e. the selection part of the includes operation
130  argument.setName("select");
131  //Iterator result is a set
132  argument.setType(makeSet());
133  --argument expression looks like the following:
134  --context->select(var|typeCheck AND elementConstraints)
135 
136   --combine the argument with the result expression to get
137   --context->includes(All)(context->select(var|typeCheck AND
138   --elementConstraints)(->first()))
139   if(self.iterationType.repr() = "includes")then{
140     var firstFromSelect :OCL::expressions::OperationCallExp =
141     new OCL::expressions::OperationCallExp();
142     firstFromSelect.referredOperation :=
143     makeOperation("OrderedSet(T)_Class", "first");
```

```
144            firstFromSelect.source:= argument;
145            res.argument += firstFromSelect;}
146        --Case: for includesAll not only the first element is needed
147          else{
148           res.argument += argument;
149          }
150          endif;
151          res.referredOperation :=
152          makeOperation("OrderedSet(T)_Class", "includes");
153          res.setType(Boolean);
154          var src:= argument.source.deepclone().
155          oclAsType(OCL::expressions::OCLExpression);
156          src.setType(makeSet());
157          res.source :=src;
158          return res;
159          }
160          --Case: Association makes boolean expression over the
161          --set with notEmpty or isEmpty
162          else if(self.iterationType.repr() = "isEmpty" or
163           self.iterationType.repr() = "notEmpty") then
164          {
165           var res :OCL::expressions::OperationCallExp =
166           new OCL::expressions::OperationCallExp();
167           --check whether the context is calculated
168           --or is alkready avalaible
169           if(InputSrc= null or InputSrc.
170           oclIsKindOf(OCL::expressions::NullLiteralExp)) then{
171            res.source := self.reference2PropCallExp();
172           }
173           else{
174             var srcTMP:=InputSrc.deepclone().
175             oclAsType(OCL::expressions::OCLExpression);
176             res.source := self.reference2PropCallExp(srcTMP);
177            }endif;
178            res.source.setType(makeSet());
179            --make the operation of the result expression
180            if(self.iterationType.repr() = "isEmpty") then{
181             res.referredOperation :=
182             makeOperation("OrderedSet(T)_Class", "isEmpty");
183            }
184            else if(self.iterationType.repr() = "notEmpty") then{
185              res.referredOperation :=
186              makeOperation("OrderedSet(T)_Class", "notEmpty");
187            }endif endif;
188            res.setType(Boolean);
189            return res;
190            }
191            --Case: Association makes boolean expression
192            --over the set with size
193          else if(self.iterationType.repr() = "size") then{
194            if(self._end.outReferences->isEmpty() or
```

```
195             self._end.outReferences = null)then{
196             --For this kind of expression the size
197             --of a set is compared with another size.
198             --as a comparison operator is the encompassing
199             --operation an OperationCall expression is used
200             var res :OCL::expressions::OperationCallExp =
201             new OCL::expressions::OperationCallExp();
202             --Create the expression to determine the size of
203             --a set of objects
204             var src :OCL::expressions::OperationCallExp =
205             new OCL::expressions::OperationCallExp();
206 
207             if(InputSrc= null or InputSrc.
208             oclIsKindOf(OCL::expressions::NullLiteralExp))then{
209               src.source := self.reference2PropCallExp();
210             }
211             else{
212               var srcTMP:=InputSrc.deepclone().
213               oclAsType(OCL::expressions::OCLExpression);
214               src.source := self.reference2PropCallExp(srcTMP);
215             }endif;
216             src.source.setType(makeSet());
217             src.referredOperation :=
218             makeOperation("OrderedSet(T)_Class", "size");
219             src.setType(Integer);
220 
221             --compare the result with a value
222             res.source := src;
223             res.referredOperation := makeOperation("OclAny", "=");
224             --set value with which size should be compared
225             var argument := self._end.expression.makeVariable();
226             argument.setType(Integer);
227             res.argument += argument;
228             res.setType(Boolean);
229 
230             return res;
231             }
232             --Special Case: Association makes boolean expression
233             --over a CONSTRAINT set with size
234             else{
235             var sizeResult :OCL::expressions::OperationCallExp =
236             new OCL::expressions::OperationCallExp();
237             --Make the resulting IteratorExpression
238             var res :OCL::expressions::IteratorExp =
239             new OCL::expressions::IteratorExp();
240             --Make all iterator variables, i.e. for
241             --all associated individuals
242             var iterators :OrderedSet(OCL::expressions::Variable);
243             self._end.outReferences->forEach
244             (iterator|iterator.oclIsTypeOf(GL::InstanceTransition)){
245               iterators += iterator._end.makeVariable();
```

```
246                  };
247                  --Set iterator variables
248                  iterators->forEach(iter){
249                    res.iterator += iter;
250                  };
251                  --Set body expression
252                  res.body :=self._end.oclAsType(GL::NavigationNode).
253                  handleNavigationNodeForSet(null,inputVars);
254                  --Set source expression
255                  if(InputSrc= null or InputSrc.oclIsKindOf
256                  (OCL::expressions::NullLiteralExp))then{
257                    res.source := self.reference2PropCallExp();
258                  }
259                  else{
260                    var srcTMP:=InputSrc.deepclone().oclAsType
261                    (OCL::expressions::OCLExpression);
262                    res.source := self.reference2PropCallExp(srcTMP);
263                  }endif;
264                  --Set name of the iterator, i.e. the type of iteration
265                  --(forAll, exists, collect, select,..)
266                  res.setName("select");
267                  //Iterator result is a set
268                  res.setType(makeSet());
269                  sizeResult.source:=res;
270                  sizeResult.source.setName("select");
271                  res.iterator->forEach(iter){
272                    sizeResult.source.oclAsType
273                    (OCL::expressions::IteratorExp).iterator += iter;
274                  };
275                  sizeResult.source.setType(makeSet());
276                  sizeResult.referredOperation :=
277                  makeOperation("OrderedSet(T)_Class", "size");
278                  sizeResult.setType(Integer);
279                  return   sizeResult;
280                }endif;
281              }
282            endif endif endif endif;
283     }
284     --Handle object associations, i.e. upper bound is 1->
285     --Must be transformed to an CallOperation
286     else{
287       var src:= self.reference2PropCallExp();
288       var expressions :Sequence(OCL::expressions::OCLExpression);
289       expressions += self._end.oclAsType(GL::NavigationNode).
290       handleNavigationNodeForField(src,inputVars);
291       var res:= expressions->makeCompositeOpCallOR();
292       return res;
293     }endif;
294 }
295
296 --Handles a concept node which is referenced as set
```

```
297 helper GL::NavigationNode::handleNavigationNodeForSet
298 (InputSrc:OCL::expressions::OCLExpression,inputVars:OrderedSet(String)):
299 OCL::expressions::OCLExpression{
300       var expressions :Sequence(OCL::expressions::OCLExpression) ;
301       var vars :        OrderedSet(String);
302       --calculate the sub-expressions for each outgoing association
303       self.outReferences->forEach(outgoing){
304         var srcTMP:=InputSrc.deepclone().oclAsType
305         (OCL::expressions::OCLExpression);
306         var iteratorRefExp=outgoing.oclAsType(GL::InstanceTransition).
307         handleInstanceTransition(srcTMP,vars);
308       expressions += iteratorRefExp;
309       iteratorRefExp.oclAsType(ocl::expressions::IteratorExp).
310       iterator->forEach(v){vars += v.getName()};
311       };
312       --if there are more than one variable it must be
313       --ensured that the compared one are different
314       var impliesClause :OCL::expressions::OCLExpression ;
315       if(self->outReferences->size()>1 and
316       self.inReferences->first().oclAsType
317       (GL::AssociationTransition).objectRef.upperBound <>1)then{
318         var variables : Sequence(String);
319         self.outReferences->forEach(outgoing){
320           variables += outgoing._end.name.substringBefore("_");
321         };
322         impliesClause := variables->makeImpliesSource();
323       }
324       endif;
325       --combine expression results from outgoing
326       --individual associations
327       var combinedExpressions := expressions->makeCompositeOpCallOR();
328       --if only one individuals are referenced a
329       --implies clause is necessary
330       if(impliesClause=null)then{
331         return combinedExpressions;
332       }
333       --if more than one individuals are referenced a
334       --implies clause is necessary
335       else{
336         combinedExpressions:=
337         impliesClause.makeImplies(combinedExpressions);
338         return combinedExpressions;
339       }endif;
340 }
341
342 --Handles a concept node which is referenced as single object
343 helper GL::NavigationNode::handleNavigationNodeForField
344 (InputSrc:OCL::expressions::OCLExpression,inputVars:OrderedSet(String)):
345 Sequence(OCL::expressions::OCLExpression){
346       --Handle outgoing individual associations
347       var expressions :Sequence(OCL::expressions::OCLExpression) ;
```

```
348         var vars :          OrderedSet(String);
349         --calculate the sub-expressions for each outgoing association
350         self.outReferences->forEach(outgoing){
351           var srcTMP:=InputSrc.deepclone().oclAsType
352           (OCL::expressions::OCLExpression);
353           var iteratorRefExp:=outgoing.oclAsType(GL::InstanceTransition).
354           handleInstanceTransition(srcTMP,vars);
355         expressions += iteratorRefExp;
356         iteratorRefExp.oclAsType(ocl::expressions::IteratorExp).
357         iterator->forEach(v){vars += v.getName()};
358         };
359         return expressions;
360 }
361
362
363
364 --Handles an individual node be aggregating all of its property
365 --constraints in one Boolean expression
366 helper GL::instanceNode::handleinstanceNode
367 (InputSrc:OCL::expressions::OCLExpression,inputVars:OrderedSet(String)):
368 OCL::expressions::OCLExpression{
369       var expressions :Sequence(OCL::expressions::OCLExpression) ;
370       --only for individual relation between objects, e.g.
371       --o1 <>02 in contrast to o1.name <> o2.name
372       self.individualOuts->forEach(individualRelationOUT|
373       (individualRelationOUT.firstParameter= null and
374       individualRelationOUT.secondParameter=null)or
375       (individualRelationOUT.firstParameter= "" and
376       individualRelationOUT.secondParameter="")){
377         var relationOut :=
378         individualRelationOUT.handleIndividualRelationTarget(inputVars);
379         if(relationOut != null)then{
380           expressions += relationOut;
381         }
382         endif;
383       };
384
385       --only for individual relation between objects, e.g.
386       --o1 <>02 in contrast to o1.name <> o2.name
387       self.individualIns->forEach(individualRelationIN|
388       (individualRelationIN.firstParameter= null and
389       individualRelationIN.secondParameter=null)or
390       (individualRelationIN.firstParameter= "" and
391       individualRelationIN.secondParameter="")){
392         var relationIn :=
393         individualRelationIN.handleIndividualRelationSource(inputVars);
394           if(relationIn != null)then{
395             expressions += relationIn;
396           }endif;
397         };
398
```

```
399   --Handle all attributes
400   self.attributes->forEach(ar){
401       var srcTMP:=
402       InputSrc.deepclone().oclAsType(OCL::expressions::OCLExpression);
403       var attr = ar.handleAttribute(srcTMP,inputVars);
404       if(attr != null)then{
405         expressions += attr;
406       }
407       endif;
408     };
409
410 --Handle the expression field to refer the object itself
411   if(self.expression <> null and self.expression <> "" and
412   not(self.isInSetContext()))then{
413     var srcTMP:=
414     InputSrc.deepclone().oclAsType(OCL::expressions::OCLExpression);
415     var exprTMP :=
416     self.makeExpressionFromString(srcTMP,self.expression);
417       if(exprTMP<>null)then{
418         expressions+= exprTMP;
419       }endif;
420   }endif;
421
422   --Handle outgoing navigating associations
423   var vars :    OrderedSet(String);
424   var refs :Sequence(OCL::expressions::OCLExpression) ;
425   self.outReferences->forEach(outgoing){
426     var srcTMP:=
427     InputSrc.deepclone().oclAsType(OCL::expressions::OCLExpression);
428     var iteratorRefExp = outgoing.oclAsType(GL::AssociationTransition).
429     handleAssociationTransition(srcTMP,vars);
430   if(iteratorRefExp.oclIsTypeOf(OCL::expressions::IteratorExp))then{
431     refs += iteratorRefExp;
432   }
433   else{
434     expressions += iteratorRefExp;
435   }endif;
436   iteratorRefExp.oclAsType(ocl::expressions::IteratorExp).
437   iterator->forEach(v){vars += v.getName()};
438   };
439   --combine different Iterator Expressions
440   --received from different paths
441   var combinedIteratorExp : OCL::expressions::OCLExpression;
442   if(refs <> null or refs->size()>0)then{
443     if(refs->size()>1)then{
444       combinedIteratorExp:= refs->combineExpressions();
445       expressions+= combinedIteratorExp;
446     }
447     else{
448      if(refs->first()!= null)then{
449       expressions +=     refs->first();
```

```
450        }
451        endif;
452      }endif;
453    }endif;
454 var combinedExpressions:= expressions->makeCompositeOpCallAND();
455 return combinedExpressions;
456 }
457
458
459 --Helper function to combine a sequence of OCL expression
460 --Distinguishes different types of expressions and is responsible
461 --to create prenex form from a set of iterator expressions
462 helper Sequence(OCL::expressions::OCLExpression)::combineExpressions( ):
463  OCL::expressions::OCLExpression{
464
465   var resultExpression: OCL::expressions::OperationCallExp :=
466   new OCL::expressions::OperationCallExp();
467   var booleans :Sequence(OCL::expressions::IteratorExp);
468   var sets   :List(OCL::expressions::IteratorExp);
469   var bodyParameter  :List(OCL::expressions::OCLExpression);
470
471   //group the iterator expressions by type
472   self->forEach(e){
473     if(e.oclIsTypeOf(OCL::expressions::IteratorExp))then{
474       if(e.oclAsType(OCL::expressions::IteratorExp).getName()="forAll")
475       then{
476         booleans += e.oclAsType(OCL::expressions::IteratorExp);
477       }
478       else
479       if(e.oclAsType(OCL::expressions::IteratorExp).getName()="exists")
480       then{
481         booleans += e.oclAsType(OCL::expressions::IteratorExp);
482       }
483       else
484       if(e.oclAsType(OCL::expressions::IteratorExp).getName()="isUnique")
485       then{
486         bodyParameter += e.oclAsType(OCL::expressions::IteratorExp);
487       }
488       else
489       if(e.oclAsType(OCL::expressions::IteratorExp).getName()="collect")
490       then{
491         sets += e.oclAsType(OCL::expressions::IteratorExp);
492       }
493       else
494       if(e.oclAsType(OCL::expressions::IteratorExp).getName()="select")
495       then{
496         sets += e.oclAsType(OCL::expressions::IteratorExp);
497       }
498       else {
499         bodyParameter += e.oclAsType(OCL::expressions::OCLExpression);
500       }
```

```
501         endif endif endif endif endif;
502         }
503         else{
504           bodyParameter += e.oclAsType(OCL::expressions::OCLExpression);
505         } endif;
506       };
507       var actualExpRes : OCL::expressions::IteratorExp;
508       if(booleans->size()>1)then{
509       --combine the bodies of all iterator expressions
510       var bodies : Sequence(OCL::expressions::OCLExpression);
511       booleans->forEach(b|b.body <>null){
512         bodies += b.body;
513       };
514       var bodyInput : OCL::expressions::OCLExpression;
515       if(bodies->size()=1)then{
516         bodyInput := bodies->first();
517       }
518       else {
519         bodyInput := bodies->combineBodiesAND();
520       }endif;
521       --build prenex form
522       actualExpRes =booleans->combineBooleans(bodyInput);
523       }else{
524         actualExpRes := booleans->first();
525       }
526       endif;
527       return actualExpRes;
528 }
529 
530 --Helper function to combine a sequence of expressions
531 --with the boolean connector "AND" in one OperationCall expression
532 helper Sequence(OCL::expressions::OCLExpression)::combineBodiesAND():
533  OCL::expressions::OperationCallExp{
534 
535   var res : OCL::expressions::OperationCallExp =
536   new OCL::expressions::OperationCallExp();
537   res.source := self->first();
538   res.setType(Boolean);
539   res.referredOperation:= makeOperation("Boolean", "and");
540   self->forEach(arg|arg<>self->first()){
541     res.argument += arg;
542   };
543 return res;
544 }
545 
546 --Helper function to combine iterator expressions in prenex form.
547 --Finally, the body input paramter is set
548 helper Sequence(OCL::expressions::IteratorExp)::
549 combineBooleans(bodyInput:OCL::expressions::OCLExpression):
550 OCL::expressions::IteratorExp{
551   var res :ocl::expressions::IteratorExp =
```

```
552     new ocl::expressions::IteratorExp();
553     res.source := self->first().source;
554     res.setType(self->first().getType());
555     res.setName(self->first().getName());
556     self->first()->iterator->forEach(iter){
557       var newIter := iter;
558       res.iterator += newIter;
559     };
560     if(self->size()>1)then{
561       self->reject(a|a=self->first())->
562       combineBooleansRecursive(res,bodyInput);
563     }
564     else{
565     res.body := bodyInput;
566     }
567     endif;
568 return res;
569 }
570
571 --Helper function to combine iterator expressions in prenex form.
572 --Finally, the body input paramter is set
573 helper Sequence(OCL::expressions::IteratorExp)::
574 combineBooleansRecursive(inout input :
575 OCL::expressions::IteratorExp ,
576 in bodyInput:OCL::expressions::OCLExpression){
577     var res :ocl::expressions::IteratorExp =
578     new ocl::expressions::IteratorExp();
579     res.source := self->first().source;
580     res.setType(self->first().getType());
581     res.setName(self->first().getName());
582     self->first()->iterator->forEach(iter){
583     var newIter := iter;
584       res.iterator += newIter;
585     };
586     input.body := res;
587     if(self->size()>1)then{
588       self->reject(a|a=self->first())->
589       combineBooleansRecursive(res,bodyInput);
590     }
591     else{
592       res.body := bodyInput;
593     }
594     endif;
595 }
596 --Return whether an individual is used in the
597 --context of a set of objects(containment/composite)
598 helper GL::instanceNode::isInSetContext():Boolean{
599     if(self.inReferences->first().start.inReferences->
600     first().objectRef.upperBound<>1)then{
601       return true;
602     }
```

```
603   else{
604     return false;
605   }endif;
606 }
607
608 --Map a Feature attribute which consists of a value,
609 --to its corresponding condition which consists of
610 --a property call expression(source) and a value to compare
611 helper GL::FeatureAttribute::handleAttribute
612 (InputSrc:OCL::expressions::OCLExpression,
613 inputVars:OrderedSet(String)) : OCL::expressions::OperationCallExp {
614   --Attribute is target of an individual relation condition, i.e
615   --not to process
616   if((self.value = null or self.value = '') and
617   self.isParameterAttributeCall() = null)then{
618     logme("unneccessary",self);
619   }
620   --Attribute must be handled
621   else{
622     //Attribute property value is related with a null object value
623     if(self.value = 'null')then{
624       var res : OCL::expressions::OperationCallExp :=
625       new OCL::expressions::OperationCallExp();
626       //Source
627       res.source := self.attribute2PropCallExp(InputSrc,inputVars);
628       res.setType(Boolean);
629       //Operation
630       res.referredOperation:= makeOperation("String", self.expression);
631       //Argument
632       res.argument += self.variable2NullLiteral();
633       return res;
634     }
635     //Attribute integer value is related with a Integer value
636     else if(isInteger(self.objectRef))then{
637       var res : OCL::expressions::OperationCallExp :=
638       new OCL::expressions::OperationCallExp();
639       //Source
640       res.source := self.attribute2PropCallExp(InputSrc,inputVars);
641       res.setType(Boolean);
642       //Operation
643       res.referredOperation:= makeOperation("String", self.expression);
644       //Argument
645       res.argument += self.variable2IntLiteral();
646       return res;
647       }
648     --Attribute property value is related with another
649     --attribute property value
650     else if((self.value = null or self.value = '' or self.value = '␣')
651     and self.isParameterAttributeCall()!=null)then{
652       var res : OCL::expressions::OperationCallExp :=
653       new OCL::expressions::OperationCallExp();
```

```
654         //Source
655         res.source := self.attribute2PropCallExp(InputSrc,inputVars);
656         res.setType(Boolean);
657         var rel : GL::IndividualRelation :=
658         self.isParameterAttributeCall();
659         res.referredOperation:=
660         makeOperation("String", rel.relationType);
661         var attr : GL::FeatureAttribute;
662         if(rel.firstParameter = self)then{
663           attr:= rel.secondParameter;
664         }
665         else{
666           attr:= rel.firstParameter;
667         }
668         endif;
669         //Argument
670         res.argument += attr.attribute2PropCallExp(null,inputVars);
671         return res;
672       }
673       --Attribute String property must be matched with a
674       --regular expression
675       else if(isString(self.objectRef) and
676       self.expression.repr() = 'regexMatch')then{
677         var res : OCL::expressions::OperationCallExp =
678         new OCL::expressions::OperationCallExp();
679         //Source
680         res.source := self.regexAttribute2OpCallExp(InputSrc,inputVars);
681         res.setType(Boolean);
682         //Operation
683         res.referredOperation:= makeOperation("String", '<>');
684         //Argument
685         res.argument += self.variable2NullLiteral();
686         return res;
687       }
688       //Attribute String value is related with a String value
689       else if(isString(self.objectRef))then{
690         var res : OCL::expressions::OperationCallExp =
691         new OCL::expressions::OperationCallExp();
692         //Source
693         res.source := self.attribute2PropCallExp(InputSrc,inputVars);
694         res.setType(Boolean);
695         //Operation
696         res.referredOperation:= makeOperation("String", self.expression);
697                           //Argument
698         res.argument += self.variable2StringLiteral();
699         return res;
700       }
701       else{
702         return null;
703       }endif endif endif endif endif;
704     }
```

```
705 endif;
706 }
707
708 --Makes a variable if no individuals of a concept
709 --are provided, normally 'self'
710 helper GL::Class::makeVariable() :OCL::expressions::Variable{
711   var res : OCL::expressions::Variable :=
712   new OCL::expressions::Variable();
713   res.setName(self.name.substringBefore("_"));
714   return res;
715 }
716
717 --Makes a variable if no individuals of a concept
718 -- are provided, normally 'self'
719 helper String::makeVariable() :OCL::expressions::VariableExp{
720   var res : OCL::expressions::VariableExp =
721   new OCL::expressions::VariableExp();
722   res.setName(self);
723   var variable : OCL::expressions::Variable :=
724   new OCL::expressions::Variable();
725   variable.setName(self);
726   res.referredVariable := variable;
727   return res;
728 }
729
730 //Makes an Null Literal Expression, i.e. null
731 helper GL::FeatureAttribute::variable2NullLiteral() :
732 OCL::expressions::NullLiteralExp {
733   var res : OCL::expressions::NullLiteralExp:=
734   new OCL::expressions::NullLiteralExp();
735   res.setName("null");
736   return res;
737 }
738
739 //Makes an String Literal Expression, i.e. '123'
740 helper GL::FeatureAttribute::variable2StringLiteral() :
741 OCL::expressions::StringLiteralExp {
742   var res : OCL::expressions::StringLiteralExp :=
743   new OCL::expressions::StringLiteralExp();
744   res.stringSymbol:= ''+ self.value;
745   return res;
746 }
747
748 //Makes an Integer Literal Expression, i.e. 123
749 helper GL::FeatureAttribute::variable2IntLiteral() :
750 OCL::expressions::IntegerLiteralExp {
751   var res : OCL::expressions::IntegerLiteralExp :=
752   new OCL::expressions::IntegerLiteralExp();
753   res.integerSymbol:=toIntegerObject(self.value);
754   return res;
755 }
```

```
756
757 //Calls an regex match operation on String attributes
758 helper GL::FeatureAttribute::regexAttribute2OpCallExp
759 (InputSrc:OCL::expressions::OCLExpression,
760 inputVars:OrderedSet(String)) : OCL::expressions::OperationCallExp{
761   var res : OCL::expressions::OperationCallExp =
762   new OCL::expressions::OperationCallExp();
763   res.argument+= self.variable2StringLiteral();
764   res.referredOperation:= makeOperation("RegEx", 'regexMatch');
765   res.source := self.attribute2PropCallExp(InputSrc,inputVars);
766   return res;
767 }
768
769 //Calls the property of an individual in the case of an attribute
770 helper GL::FeatureAttribute::attribute2PropCallExp
771 (InputSrc:OCL::expressions::OCLExpression,
772 inputVars:OrderedSet(String)) : OCL::expressions::PropertyCallExp {
773   var res :OCL::expressions::PropertyCallExp :=
774   new OCL::expressions::PropertyCallExp();
775   res.setType(self.objectRef.eType);
776   if(InputSrc = null or
777   InputSrc.oclIsKindOf(OCL::expressions::NullLiteralExp))then{
778     res.source := self.attribute2CastedVariableExp();
779   }else{
780     res.source := InputSrc;
781   }endif;
782   res.referredProperty:= makeEObject(self.objectRef);
783   return res;
784 }
785
786 --Helper function to create a casted variable for a specific attribute
787 helper GL::FeatureAttribute::attribute2CastedVariableExp() :
788 OCL::expressions::OperationCallExp {
789   var res : OCL::expressions::OperationCallExp =
790   new OCL::expressions::OperationCallExp();
791   res.setType(self.objectRef.eContainingClass);
792   res.referredOperation := makeOperation("OclAny", 'oclAsType');
793   res.argument += self.attribute2CastingTypeExp();
794   if(self.container().oclAsType(GL::instanceNode).isForSet())then{
795     res.source := self.attribute2VariableExp();
796   }
797   else{
798     res.source := self.container().oclAsType
799     (GL::instanceNode).makeBackwardSource();
800   }endif;
801 return res;
802 }
803
804 --Helper function to decide whether an individual is referenced
805 --with a set or as single object
806 --This influences the usage the actual context which must be
```

```
807 --used to call its properties
808 helper GL::instanceNode::isForSet() : Boolean{
809         if(self.inReferences->first().oclAsType(GL::FeatureAssociation).
810         start.inReferences->first().oclAsType(GL::FeatureAssociation).
811         objectRef.upperBound<>1)then{
812     return true;
813   }
814   else{
815     return false;
816   }endif;
817 }
818 
819 helper GL::instanceNode::makeBackwardSource() :
820 OCL::expressions::OCLExpression{
821   var res : OCL::expressions::PropertyCallExp :=
822   new OCL::expressions::PropertyCallExp();
823   --search for the association property which bases the individual
824 var assoc:=self.inReferences->first().oclAsType(GL::FeatureAssociation).
825 start.inReferences->first().oclAsType(GL::FeatureAssociation);
826   --determine the source of the association property
827   var clazz:= assoc.start;
828   --Determine the context of the expression to call
829   --the association property
830   res.source := clazz.individual2VariableExp();
831   --call the referred associationproperty
832   res.referredProperty := makeEObject(assoc.objectRef);
833   return res;
834 }
835 
836 //Makes a Type exporession for casting operation
837 helper GL::FeatureAttribute::attribute2CastingTypeExp() :
838 OCL::expressions::TypeExp {
839   var res : OCL::expressions::TypeExp =
840   new OCL::expressions::TypeExp();
841   res.referredType := getClassForAttribute(self);
842   return res;
843 }
844 
845 --Helper function to create a variable for a specific attribute
846 helper GL::FeatureAttribute::attribute2VariableExp() :
847 OCL::expressions::VariableExp {
848   var res : OCL::expressions::VariableExp =
849   new OCL::expressions::VariableExp();
850   res.setName(self.name);
851   res.setType(self.objectRef.eContainingClass);
852   res.referredVariable := self.makeVariableFromAttribute();
853   return res;
854 }
855 
856 //makes a variable, eg. 'x2'
857 helper GL::FeatureAttribute::makeVariableFromAttribute() :
```

```
858 OCL::expressions::Variable{
859   var res : OCL::expressions::Variable:=
860   new OCL::expressions::Variable();
861   var container := self.container().oclAsType(GL::instanceNode);
862   var containerName := container.name.substringBefore("_");
863   res.setName(containerName);
864   res.setType(self.objectRef.eContainingClass);
865   return res;
866 }
867
868 -- checks whether a attribute is compared with another
869 --attribute in contrast to a primitive value,
870 -- i.e self.name = '123' vs. self.name = self.name
871 query GL::FeatureAttribute::isParameterAttributeCall() :
872 GL::IndividualRelation {
873   self.container().oclAsType(GL::instanceNode).individualOuts->
874   forEach(individualRel){
875     if(individualRel.firstParameter = self )then{
876       return individualRel;
877     }
878     endif;
879   }
880 }
881
882 --Helper function to parse a String expression and create a operation
883 helper GL::instanceNode::makeExpressionFromString
884 (InputSrc:OCL::expressions::OCLExpression,input:String) :
885 OCL::expressions::OperationCallExp{
886   var res : OCL::expressions::OperationCallExp =
887   new OCL::expressions::OperationCallExp();
888   if(InputSrc = null)then{
889     var variable:= self.individual2VariableExp();
890     res.source := variable;
891   }else{
892     res.source := InputSrc;
893   }endif;
894
895   //Type
896   res.setType(Boolean);
897   //Operation
898   if(input.startsWith("<>") )then{
899     res.referredOperation := makeOperation("OclAny", "<>");
900     var varName :=input.substringAfter("=");
901   }else if(input.startsWith("="))then{
902     res.referredOperation := makeOperation("OclAny", "=");
903     var varName :=input.substringAfter("=");
904   }
905   else{
906     return null;
907   }endif endif;
908   var nil : OCL::expressions::NullLiteralExp:=
```

```
909   new OCL::expressions::NullLiteralExp();
910   nil.setName("null");
911   res.argument += nil;
912   return res;
913 }
914
915          //Makes a variable expression
916 helper GL::Class::individual2VariableExp() :
917 OCL::expressions::VariableExp {
918   var res : OCL::expressions::VariableExp =
919   new OCL::expressions::VariableExp();
920   res.setName(self.name);
921   res.setType(self.objectRef);
922   res.referredVariable := self.makeVariable();
923   return res;
924 }
925
926 --Combines a sequence of expressions with the
927 --boolean connector "AND"
928 helper Sequence(OCL::expressions::OCLExpression)::
929 makeCompositeOpCallAND(): OCL::expressions::OCLExpression{
930   var i :=0;
931   var complexAttr: OCL::expressions::OCLExpression;
932   self->forEach(attr|attr<>null){
933     if(i=0)then{
934       complexAttr:= attr;
935       i:= i+1;
936     }
937     else{
938       var res : OCL::expressions::OperationCallExp =
939       new OCL::expressions::OperationCallExp();
940       res.source := complexAttr;
941       res.setType(Boolean);
942       res.referredOperation:= makeOperation("Boolean", "and");
943       res.argument += attr;
944       complexAttr := res;
945     }
946     endif;
947   };
948 return complexAttr;
949 }
950
951
952
953 //Composites a sequence of expressions with the boolean connector "OR"
954 helper Sequence(OCL::expressions::OCLExpression)::
955 makeCompositeOpCallOR(): OCL::expressions::OCLExpression{
956   var firstAttr:= self->first();
957   var i :=0;
958   var complexAttr: OCL::expressions::OCLExpression;
959   self->forEach(attr| attr <>null){
```

```
960       if(i=0)then{
961         complexAttr:= attr;
962         i:= i+1;
963       }
964       else{
965         var res : OCL::expressions::OperationCallExp =
966         new OCL::expressions::OperationCallExp();
967         res.source := complexAttr;
968         res.setType(Boolean);
969         res.referredOperation:= makeOperation("Boolean", "or");
970         res.argument += attr;
971         complexAttr := res;
972       }
973       endif;
974     };
975 return complexAttr;
976 }
977
978 //Calls the property of an individual in the case of an association
979 helper GL::AssociationTransition::reference2PropCallExp() :
980 OCL::expressions::PropertyCallExp {
981   var res :OCL::expressions::PropertyCallExp :=
982   new OCL::expressions::PropertyCallExp();
983   if(self.objectRef.upperBound<>1)then{
984     res.setType(makeSet());
985   }else{
986     res.setType(self.objectRef.eType);
987   }endif;
988   res.source := self.start.reference2VariableExp();
989   res.referredProperty:= makeEObject(self.objectRef);
990   return res
991 }
992
993 //Calls the property of an individual in the case of an association
994 helper GL::AssociationTransition::reference2PropCallExp
995 (input:OCL::expressions::OCLExpression) :
996 OCL::expressions::PropertyCallExp {
997   var res :OCL::expressions::PropertyCallExp :=
998   new OCL::expressions::PropertyCallExp();
999   res.setType(self.objectRef.eType);
1000  res.source := input;
1001  res.referredProperty:= makeEObject(self.objectRef);
1002  return res
1003 }
1004
1005 //Makes a variable expression
1006 helper GL::Class::reference2VariableExp() :
1007 OCL::expressions::VariableExp {
1008   var res : OCL::expressions::VariableExp =
1009   new OCL::expressions::VariableExp();
1010   res.setName(self.name);
```

```
1011     res.setType(self.objectRef);
1012     res.referredVariable := self.makeVariable();
1013     return res;
1014 }
1015 
1016 helper GL::IndividualRelation::handleIndividualRelationTarget
1017 (input:OrderedSet(String)) : OCL::expressions::OperationCallExp {
1018     if((input->includes
1019     (self.individualTarget.name.substringBefore("_")))) then{
1020       var res : OCL::expressions::OperationCallExp =
1021       new OCL::expressions::OperationCallExp();
1022       res.source :=self.individualSource.individual2VariableExp();
1023       //Type
1024       res.setType(Boolean);
1025       //Operation
1026       if(self.relationType.=(GL::RelationType::neq) )then{
1027         res.referredOperation := makeOperation("OclAny", "<>");
1028       }else if(self.relationType.=(GL::RelationType::eq))then{
1029         res.referredOperation := makeOperation("OclAny", "=");
1030       }endif endif;
1031     res.argument += self.individualTarget.individual2VariableExp();
1032     return res;
1033     }endif;
1034 }
1035 
1036 helper GL::IndividualRelation::handleIndividualRelationSource
1037 (input:OrderedSet(String)) : OCL::expressions::OperationCallExp {
1038     if((input->includes(self.individualSource.name.substringBefore("_"))))
1039     then{
1040       var res : OCL::expressions::OperationCallExp =
1041       new OCL::expressions::OperationCallExp();
1042     res.source :=self.individualSource.individual2VariableExp();
1043     //Type
1044     res.setType(Boolean);
1045     //Operation
1046     if(self.relationType.=(GL::RelationType::neq) )then{
1047       res.referredOperation := makeOperation("OclAny", "<>");
1048     }else if(self.relationType.=(GL::RelationType::eq))then{
1049       res.referredOperation := makeOperation("OclAny", "=");
1050     }endif endif;
1051     res.argument += self.individualTarget.individual2VariableExp();
1052     return res;
1053     }endif;
1054 }
1055 --make source of an implies expression for navigation classes
1056 --with more than one outgoing individual association
1057 --Works only with two arguments at the moment
1058 helper Sequence(String)::makeImpliesSource():
1059 OCL::expressions::OCLExpression{
1060     var res :OCL::expressions::OperationCallExp :=
1061     new OCL::expressions::OperationCallExp();
```

```
1062     var i:= 0;
1063     self->forEach(v|v<>null){
1064        if(i = 0)then{
1065           res.source:= v.makeVariable();
1066           i:=i+1;
1067        }else{
1068           res.argument += v.makeVariable();
1069        }endif;
1070     };
1071     res.referredOperation:=makeOperation("OclAny", "<>");
1072     res.setType(makeBoolean());
1073     return res;
1074  }
1075
1076  --Make an implies expression from two expressions
1077  helper OCL::expressions::OCLExpression::makeImplies
1078  (target:OCL::expressions::OCLExpression):
1079  OCL::expressions::OCLExpression{
1080     var res :OCL::expressions::OperationCallExp :=
1081     new OCL::expressions::OperationCallExp();
1082     res.source:= self;
1083     res.referredOperation:=makeOperation("Boolean", "implies");
1084     res.argument += target;
1085     res.setType(makeBoolean());
1086     return res;
1087  }
1088
1089  --Make an implies expression from two expressions
1090  helper GL::instanceNode::maketypeValidationExpression():
1091  OCL::expressions::OCLExpression{
1092     var res :OCL::expressions::OperationCallExp :=
1093     new OCL::expressions::OperationCallExp();
1094     res.source:= self.individual2VariableExp();
1095     res.referredOperation:= makeOperation("OclAny", "oclIsTypeOf");
1096     var type : OCL::expressions::TypeExp =
1097     new OCL::expressions::TypeExp();
1098     type.referredType := self.objectRef;
1099     res.argument += type;
1100     return res;
1101  }
```

C HAZARD Analysis

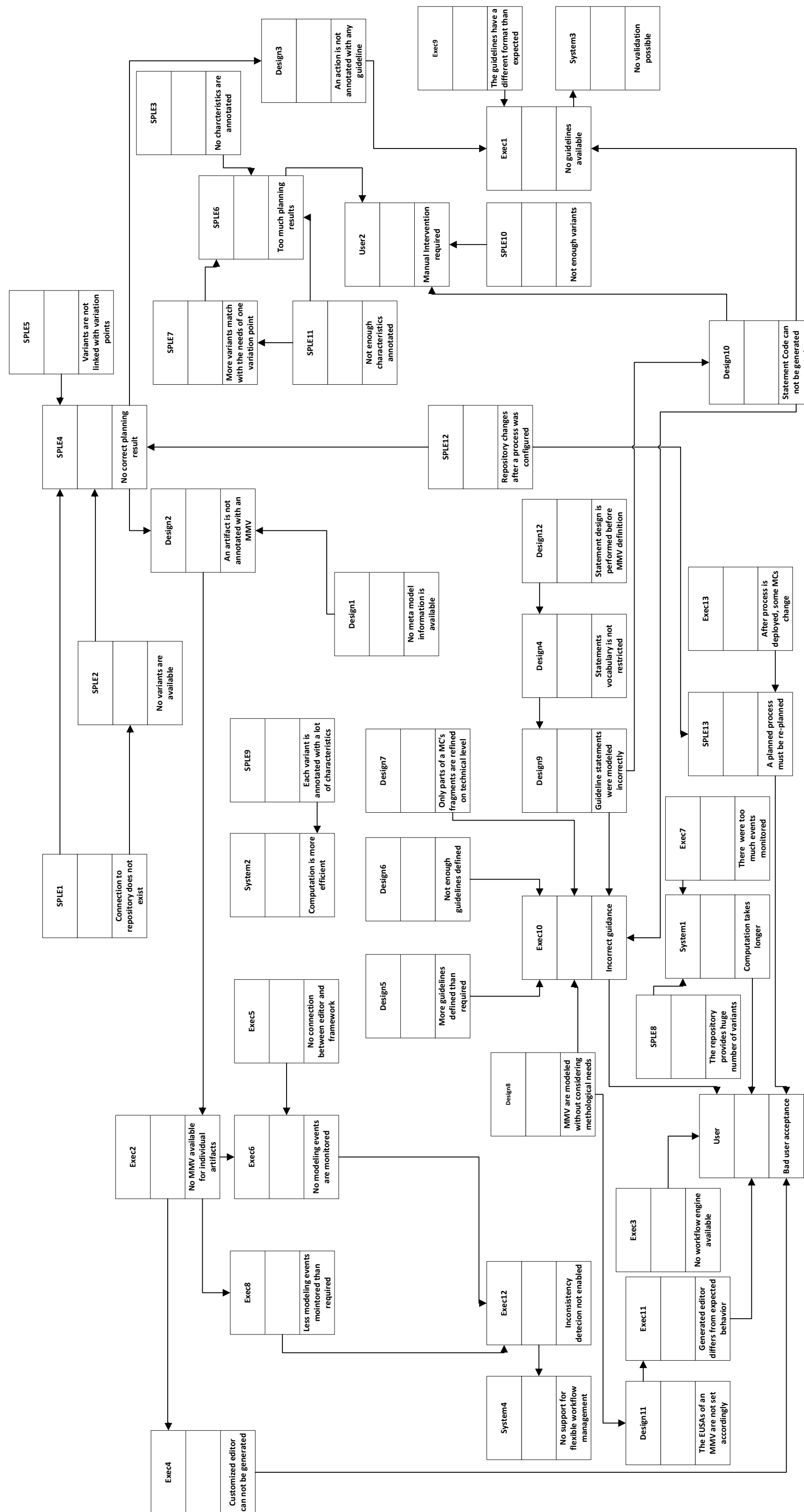


Figure C.1: Application of HASARD to our architecture

D CMMI enabled Process Areas

Process Area	Category	Level	GPs	SPs
Configuration Management (CM)	Support	2		
Measurement and Analysis (MA)	Support	2	GP 2.3;GP 2.6; GP 2.9	
Project Monitoring and Control (PMC)	Project Management	2	GP 2.2;GP 2.3; GP 2.9;GP 3.2	
Project Planning (PP)	Project Management	2	GP 2.2;GP 2.6; GP 2.9	
Process and Product Quality Assurance (PPQA)	Support	2	GP 2.6;GP 2.9	
Requirements Management (REQM)	Project Management	2	GP 2.3	
Supplier Agreement Management (SAM)	Project Management	2	GP 2.3	
Decision Analysis and Resolution (DAR)	Support	3		
Integrated Project Management (IPM)	Project Management	3	GP 2.2;GP 2.5; GP 2.6;GP 2.8; GP 2.9;GP 3.2	SP 1.1;SP 1.2 SP 1.3;SP 1.7
Organizational Process Definition (OPD)	Process Management	3	GP 2.2;GP 2.3; GP 2.6;GP 2.9; GP 3.2	SP 1.1;SP 1.2; SP 1.3;SP 1.5 SP 1.6;SP 1.7
Organizational Process Focus (OPF)	Process Management	3	GP 2.2;GP 2.6; GP 3.2	SP 3.2;SP 3.3; SP 3.4
Organizational Training (OT)	Process Management	3		
Product Integration (PI)	Engineering	3		
Requirements Development (RD)	Engineering	3	GP 2.3	
Risk Management (RSKM)	Project Management	3		
Technical Solution (TS)	Engineering	3	GP 2.2;GP 2.3; GP 2.4;GP 2.5; GP 2.6;GP 2.8; GP 2.9;GP 3.1; GP 3.2	SP 2.1
Validation (VAL)	Engineering	3	GP 2.6;GP 2.9 GP 3.2	SP 1.2;SP 1.3; SP 2.1
Verification (VER)	Engineering	3	GP 2.6;GP 2.9 GP 3.2	SP 1.2;SP 1.3; SP 3.1
Organizational Process Performance (OPP)	Process Management	4	GP 2.2; GP 2.3	
Quantitative Project Management (QPM)	Project Management	4	GP 2.6;GP 2.9	
Causal Analysis and Resolution (CAR)	Support	5	GP 2.3	
Organizational Performance Management (OPM)	Process Management	5	GP 2.3;GP 2.6	

Table D.1: CMMI-DEV process areas, associated categories, maturity levels, promoted generic practices and specific practices

E Case Study

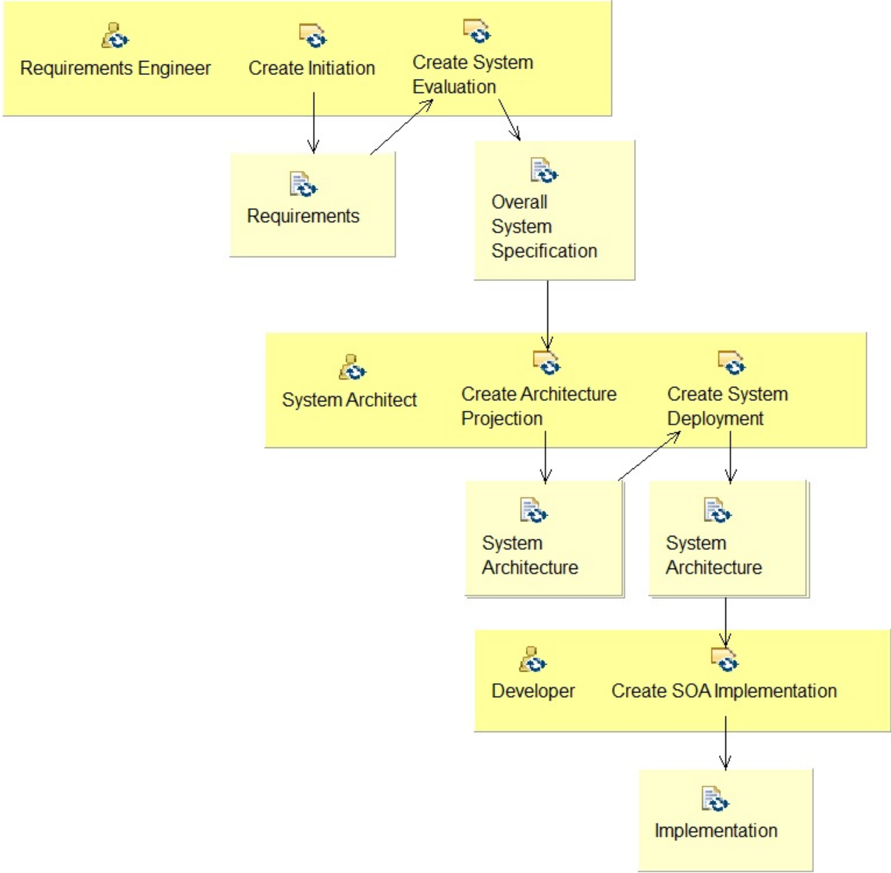


Figure E.1: Methodology for Service-Oriented Architectures

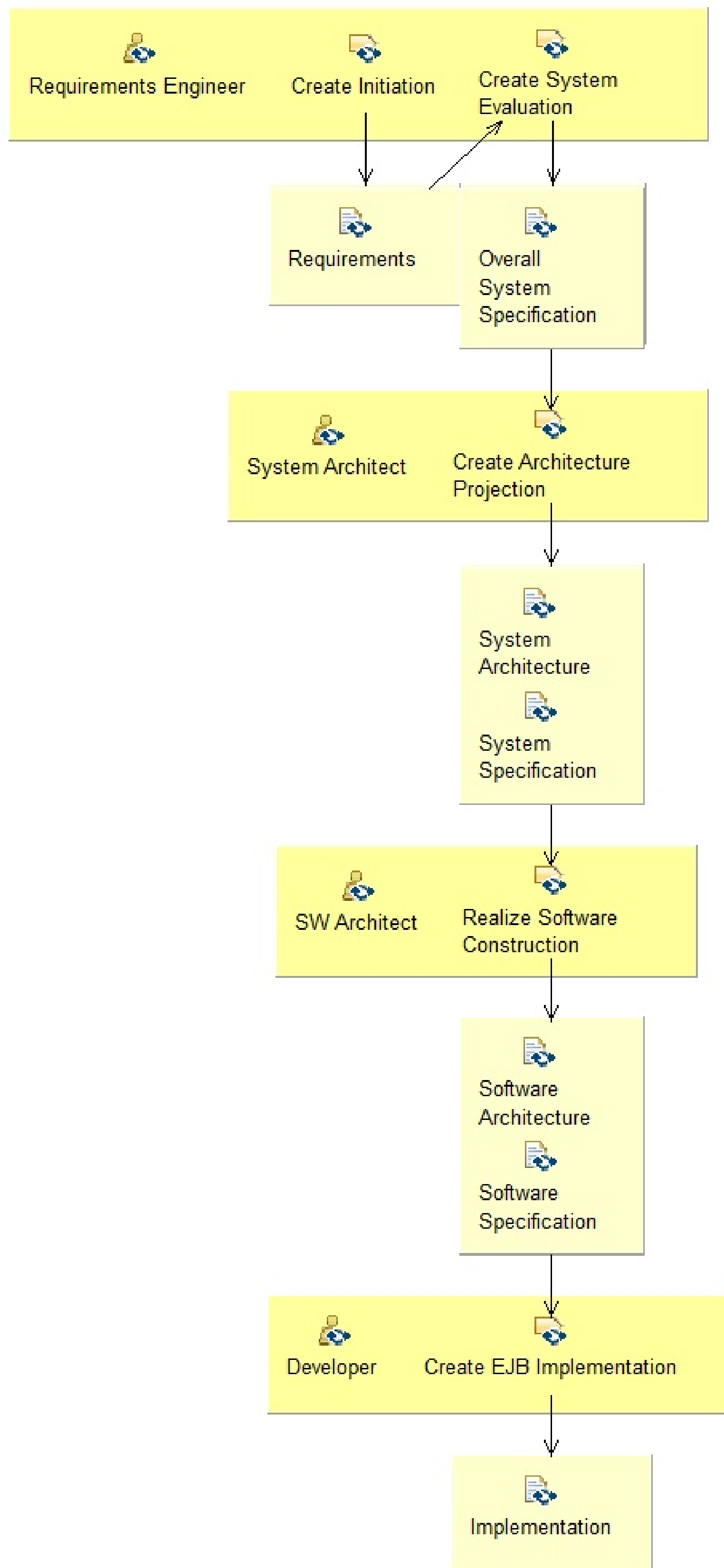


Figure E.2: Methodology for Enterprise Java Beans

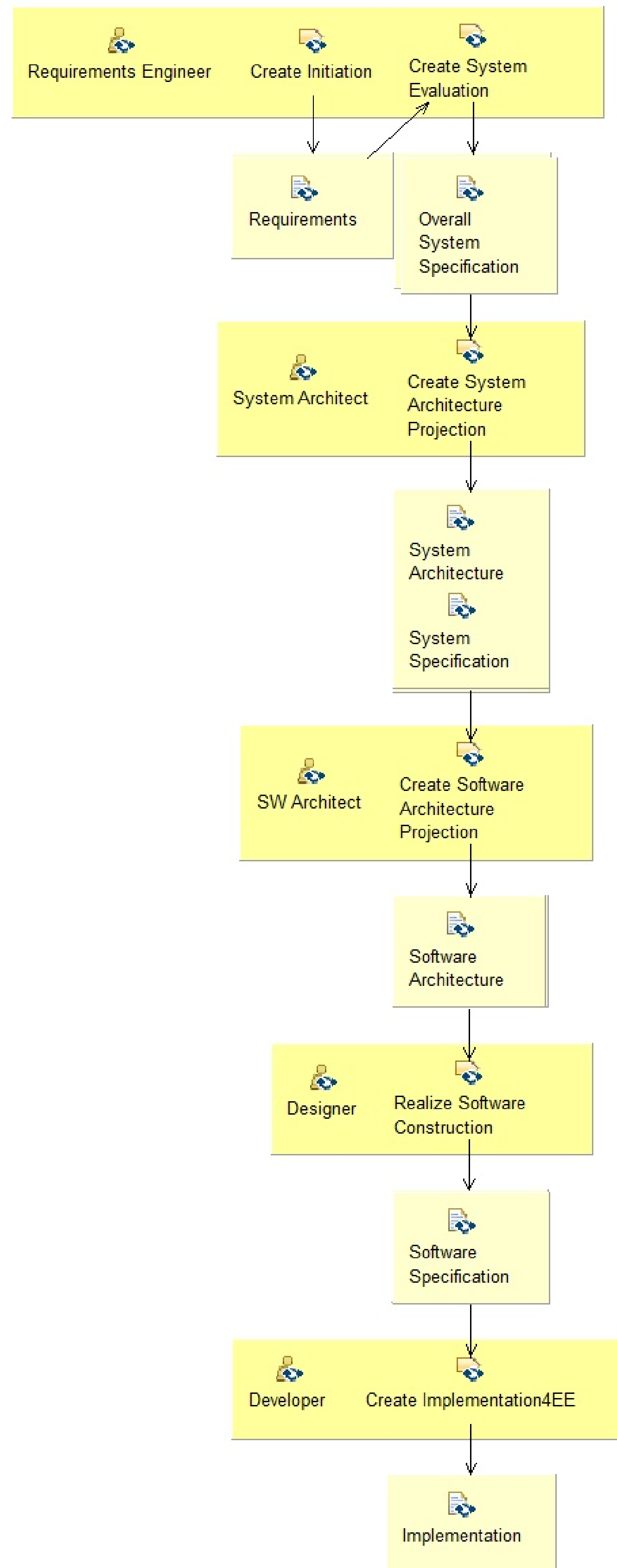


Figure E.3: Methodology for Embedded Systems

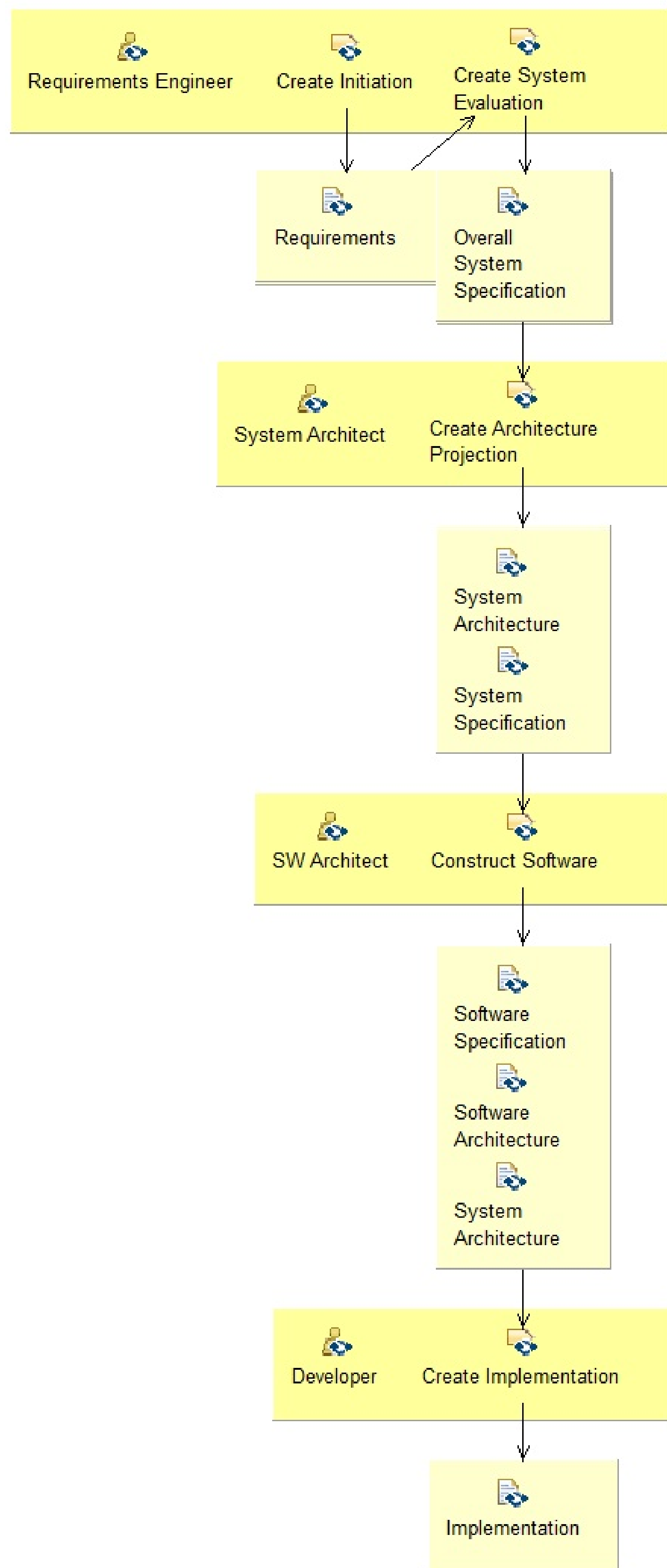


Figure E.4: Methodology for M3 Family

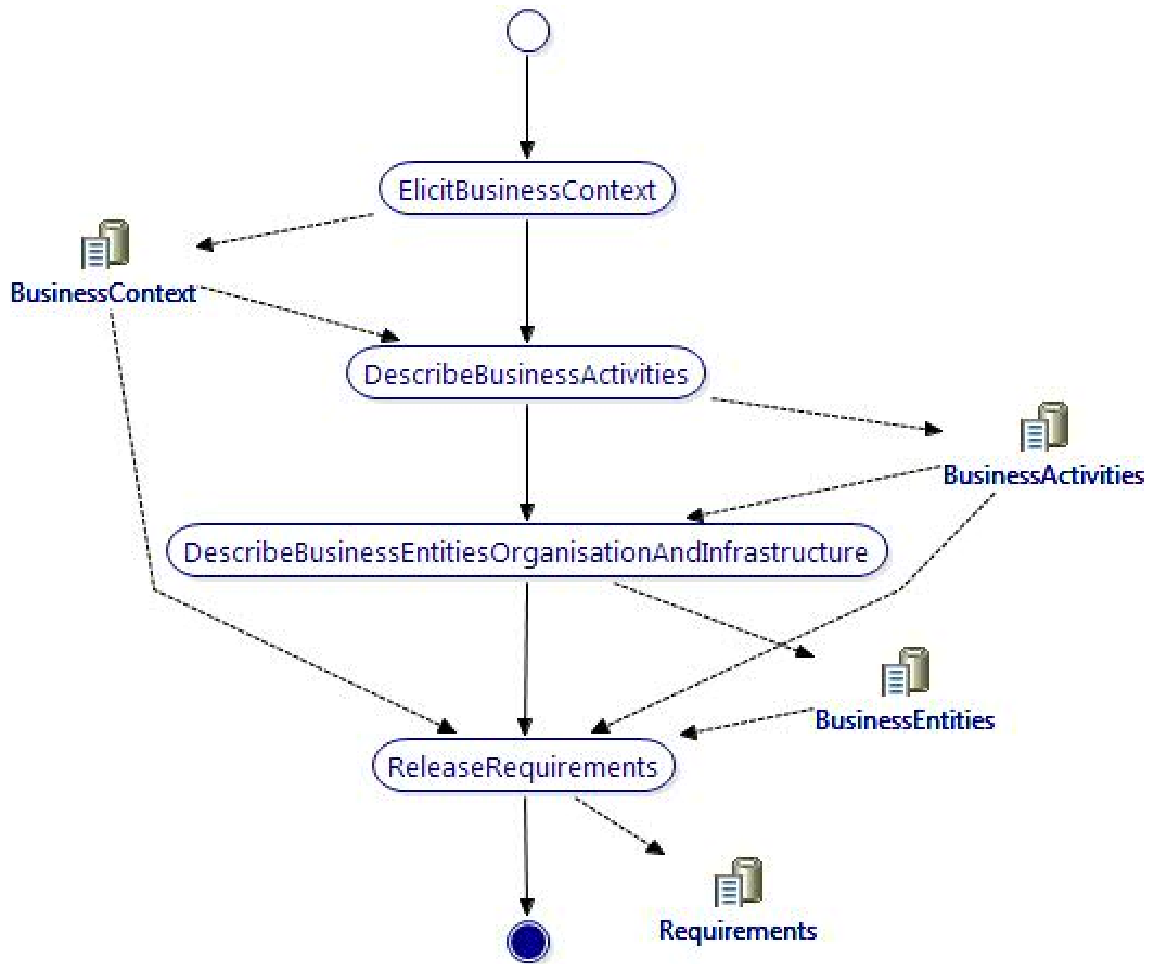


Figure E.5: M3 Initiation Variant for EJB Development

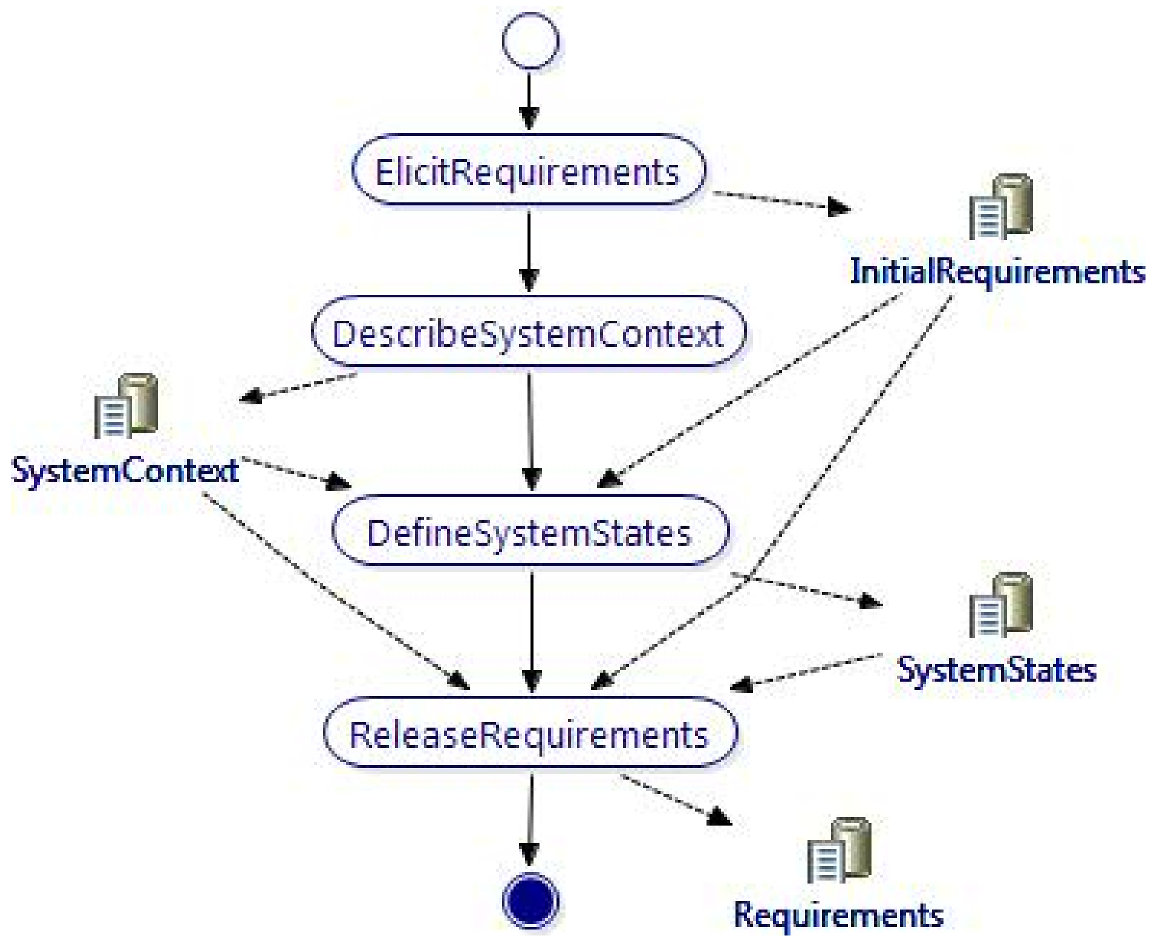


Figure E.6: M3 Initiation Variant for EE Development

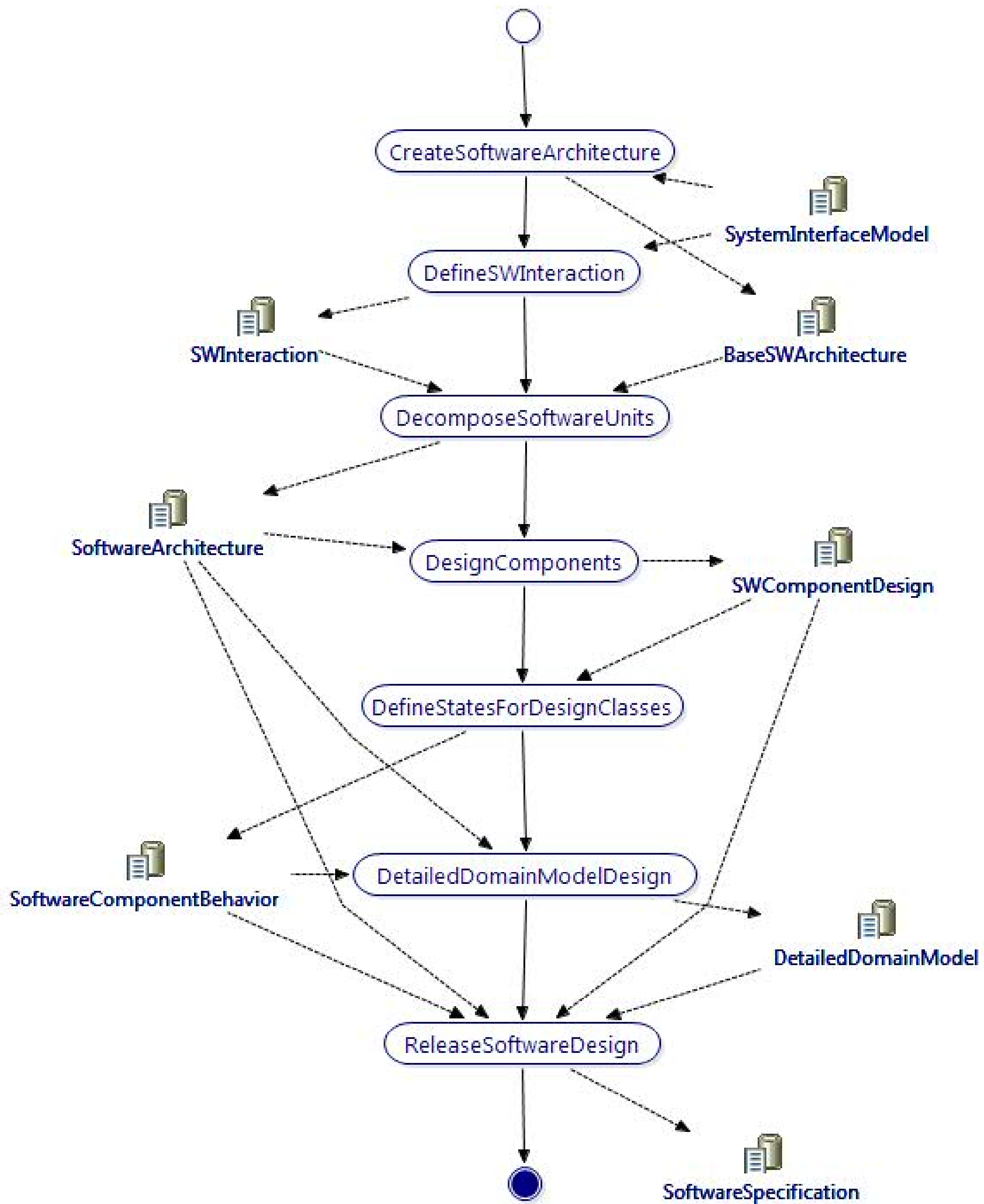


Figure E.7: M3 Realize Software Construction Variant for EE Development

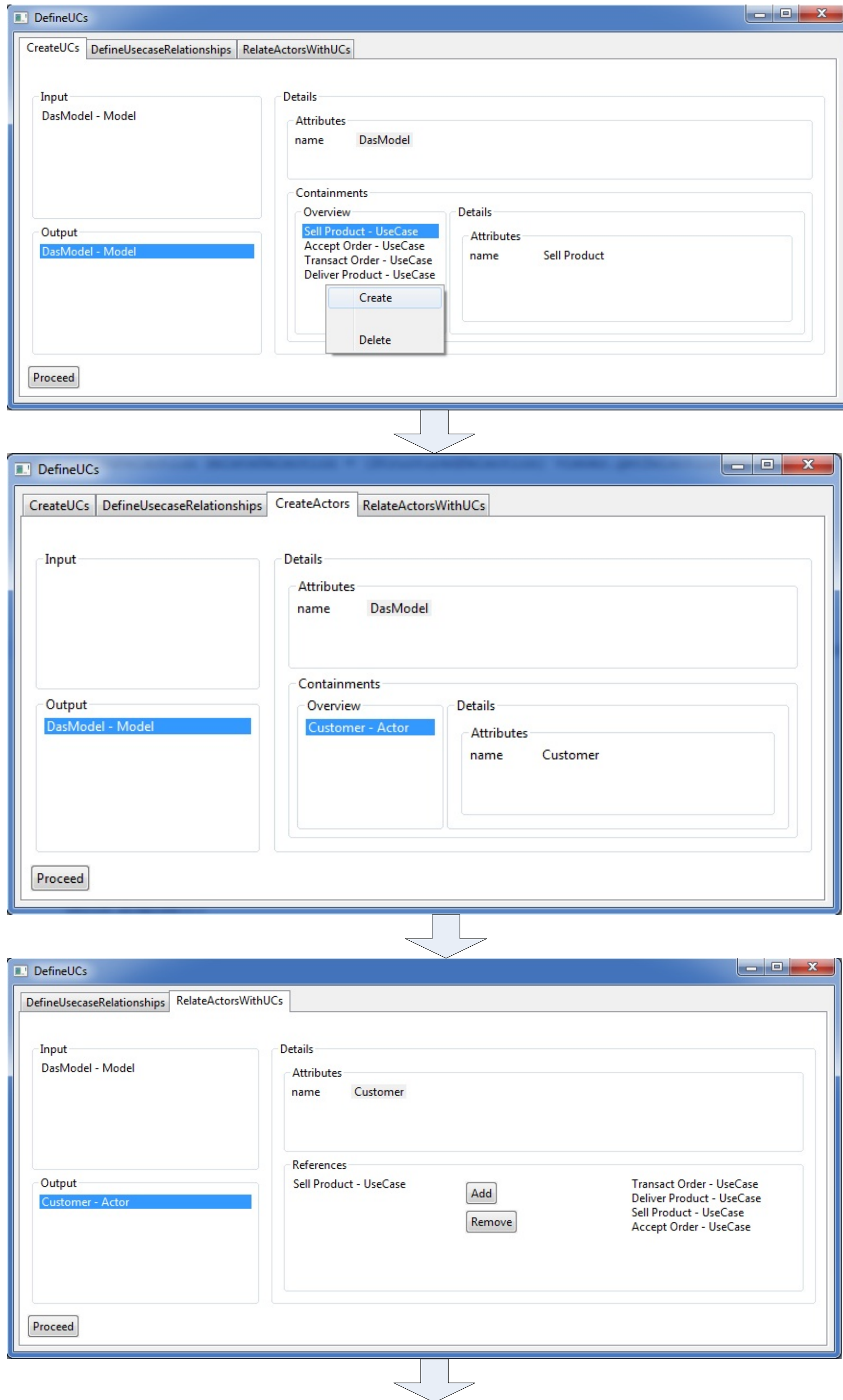


Figure E.8: SOA Initiation Guidance: Part I

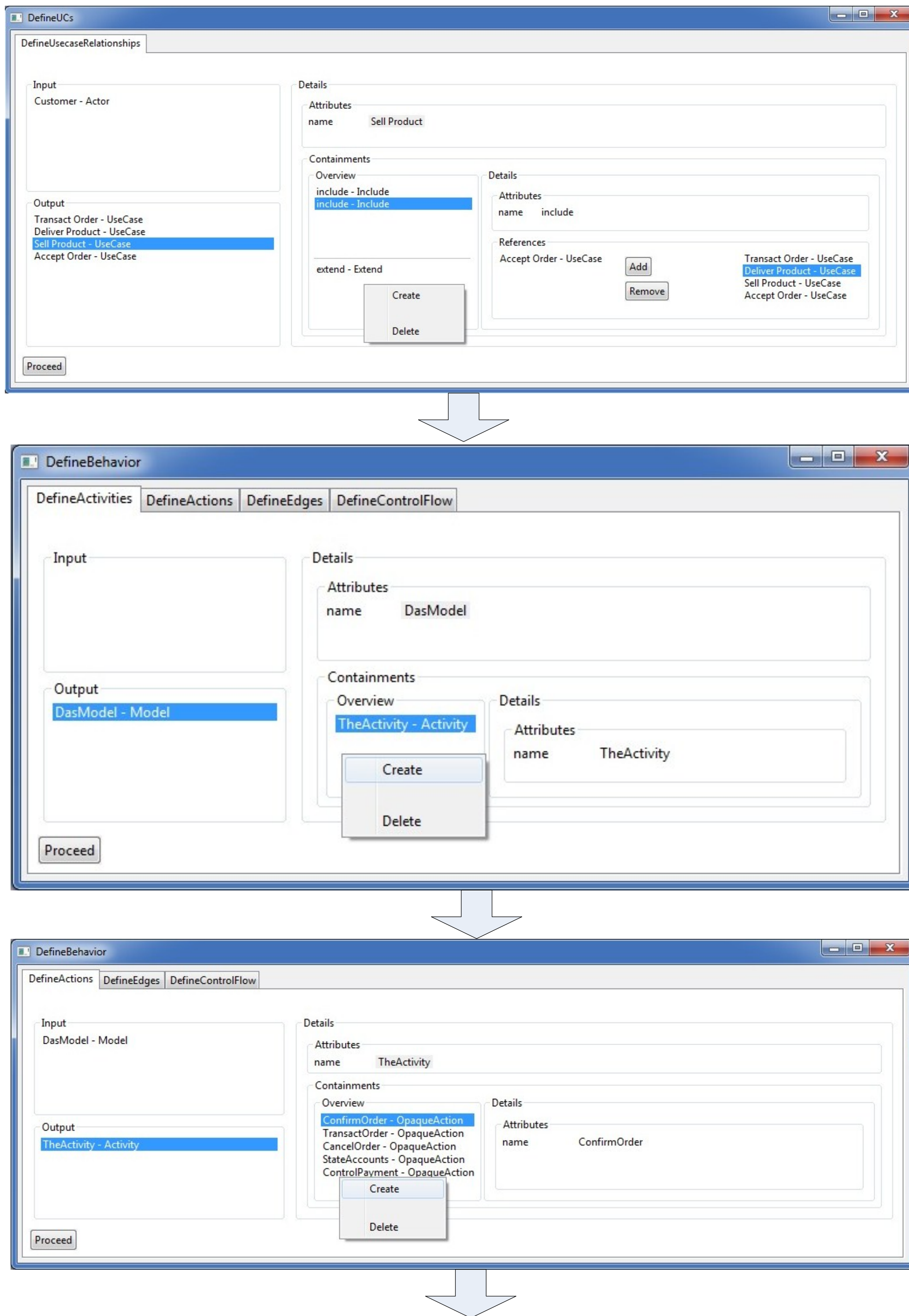


Figure E.9: SOA Initiation Guidance: Part II

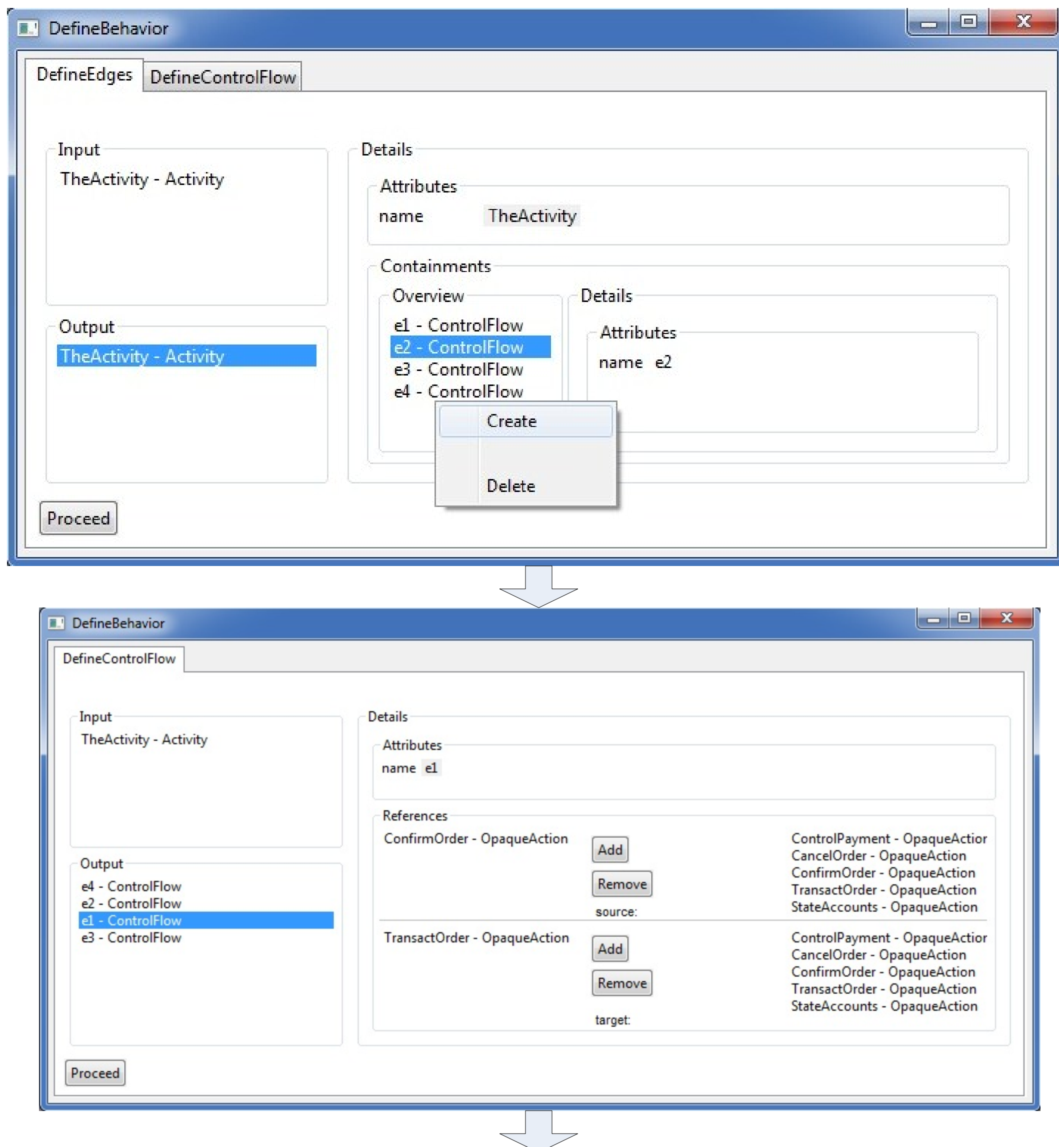
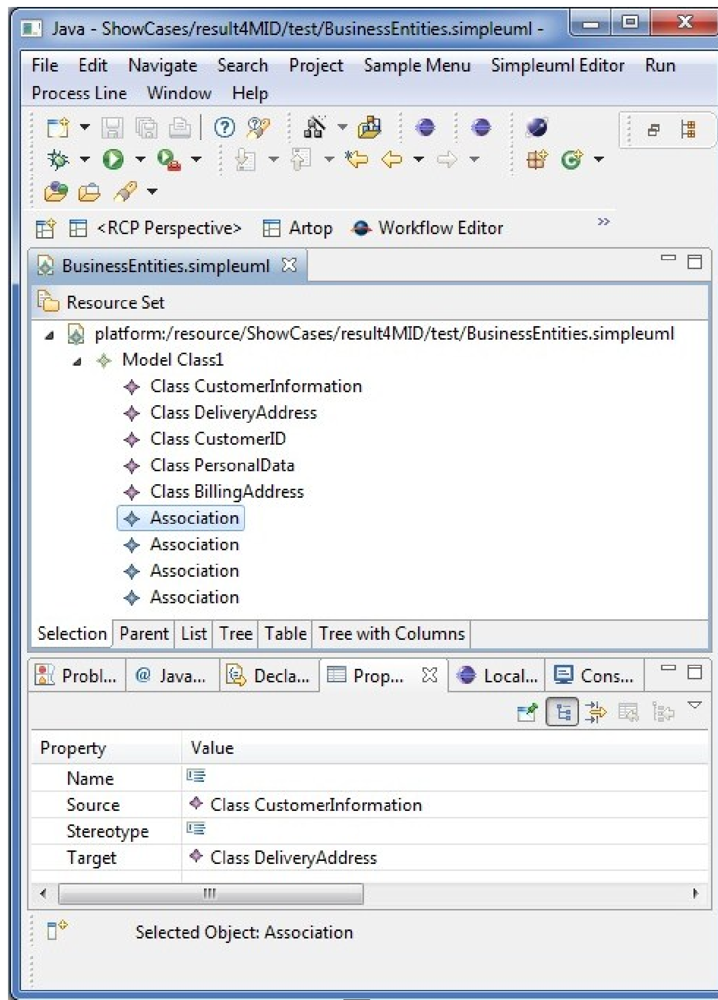


Figure E.10: SOA Initiation Guidance: Part III



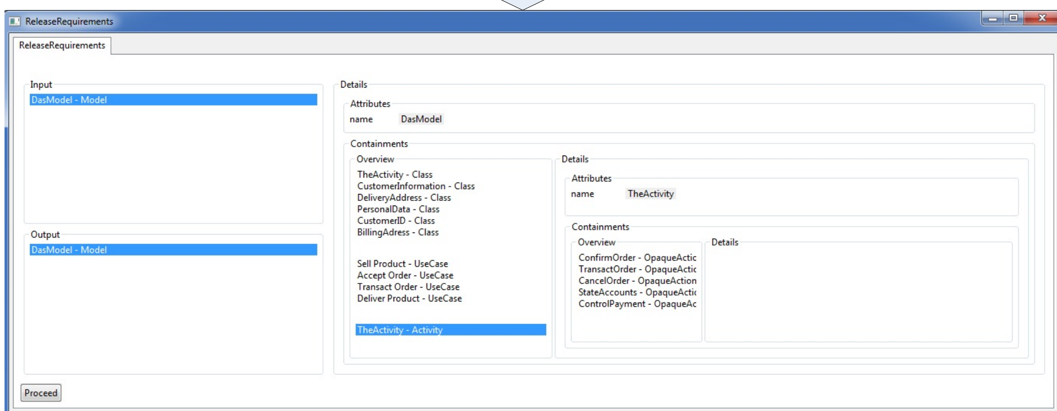


Figure E.11: SOA Initiation Guidance: Part IV

F Curriculum Vitae

Benjamin Honke
born: November 5th, 1982 in Augsburg, Germany

Education:

since 2008:	Ph.D. studies at the Software Methodologies for Distributed Systems lab of the University of Augsburg.
2003 to 2008:	Studies of Applied Computer Science at the University of Augsburg (degree: Diplom-Informatiker (Dipl.Inf.))
2002 to 2003:	Studies of Law at the University of Augsburg
1989 to 2002:	School education (Leaving certificate: Abitur)

Career:

since 2008:	Research assistant at the University of Augsburg, Germany
10/2008-07/2011:	Ph.D. student at Continental Automotive GmbH
10/2006-09/2008:	Student worker at the University of Augsburg
08/2005-09/2005:	Internship at DCM Deutsche Capital Management AG, Munich

Augsburg, April 2013