RWTH Aachen

Department of Computer Science

Technical Report

Model-Based Construction of Embedded & Real-Time Software - A Methodology for Small Devices

Alexander Nyßen

ISSN 0935–3232 · Aachener Informatik Berichte · AIB-2009-03

RWTH Aachen · Department of Computer Science · February 2009

The publications of the Department of Computer Science of RWTH Aachen (*Aachen University of Technology*) are in general accessible through the World Wide Web.

`http://aib.informatik.rwth-aachen.de/`

Model-Based Construction of Embedded & Real-Time Software - A Methodology for Small Devices

Von der Fakultät für Mathematik, Informatik und Naturwissenschaften der RWTH Aachen University zur Erlangung des akademischen Grades eines Doktors der Naturwissenschaften genehmigte Dissertation

vorgelegt von

Dipl.-Inform. Alexander Nyßen
aus Heinsberg (Rhld.)

Berichter: Prof. Dr. rer. nat. Horst Lichter
Prof. Dr.-Ing. Stefan Kowalewski

Tag der mündlichen Prüfung: 28. Januar 2009

Diese Dissertation ist auf den Internetseiten der Hochschulbibliothek online verfügbar.

Alexander Nyßen
Research Group Software Construction
any@swc.rwth-aachen.de

Aachener Informatik Bericht AIB-2009-03

Editor: Department of Computer Science
RWTH Aachen University
Ahornstr. 55
52074 Aachen
GERMANY

ISSN 0935-3232

Abstract

While model-based software engineering - due to its increased abstraction and its advantages in terms of traceability and analyzability - seems to be the adequate means to deal with the increased complexity of software that one faces today, it does not seem to have penetrated all domains yet, in particular not the one of small embedded & real-time systems. Seeing this problem caused by the fact that current model-based approaches do not pay sufficient attention to the rather special technical, organizational, and economical constraints in the respective domain, this work presents an approach that explicitly takes these constraints into account.

MeDUSA, a model-based software construction method for small embedded & real-time systems, is a principal item of the presented solution. To face the strong technical constraints it was especially designed as an instance-driven method, not incorporating any object-oriented concepts, but forcing a class-based design that can be seamlessly transferred into a procedural implementation, which is still state-of-the-art in the regarded domain. To guarantee such a seamless transition MeDUSA was furthermore designed to be a software construction rather than a mere design method, explicitly also addressing the implementation activities, and especially the transition from detailed design into source code. Being organized around the use case concept, the method excels at being very systematic and - inter alia by facilitating a continuous real-time analysis - also at being especially aware about the stringent real-time constraints that have to be faced in the domain of embedded & real-time systems.

ViPER, the supporting tool prototype, forms the second essential part of the solution. It offers generic support for MeDUSA's modeling activities by providing a graphical UML modeling environment, as well as special support for the specification and simulation of narrative, textual use case details. It furthermore demonstrates dedicated methodical support by embedding a hypertext documentation of MeDUSA's definition, by providing implementations of the MeDUSA UML profiles and model constraints, and by offering dedicated wizards to support the execution of certain MeDUSA tasks.

Together with their underlying languages, the Unified Modeling Language as well as the ANSI-C programming language, MeDUSA and ViPER thus form an integrated methodology, which is founded on shared concepts and principles. Especially developed to address above quoted constraints, the methodology is applicable to a domain, which has pretty much been elided so far.

Kurzfassung

Obwohl die modellbasierte Erstellung von Software - aufgrund ihrer höheren Abstraktion und ihrer Vorzüge in Bezug auf Nachverfolgbarkeit und Analysierbarkeit - ein adäquates Mittel zu sein scheint, um die heutige Komplexität von Software zu beherrschen, hat sie bislang nicht alle Anwendungsdomänen durchdrungen, insbesondere nicht die kleiner eingebetteter Echtzeitsysteme. Dies zurückführend auf die Tatsache, dass heutige modellbasierte Ansätze den besonderen technologischen, organisatorischen und ökonomischen Rahmenbedingungen dieser Domäne nicht in ausreichendem Maße Beachtung schenken, stellt diese Arbeit einen Ansatz vor, der diese Rahmenbedingungen explizit berücksichtigt.

MeDUSA, eine modellbasierte Software-Konstruktionsmethode für kleine eingebettete Echtzeitsysteme, bildet den ersten zentralen Bestandteil der vorgestellten Lösung. Um den starken technologischen Einschränkungen gerecht zu werden, wurde sie als instanzgetriebene Methode konzipiert, die keine objektorientierten Konzepte einsetzt und so einen klassenbasierten Entwurf forciert, der nahtlos in eine prozedurale Implementierung, welche in der betrachteten Domäne nach wie vor den Stand der Technik repräsentiert, überführt werden kann. Um einen solch nahtlosen Übergang gewährleisten zu können wurde MeDUSA als Software-Konstruktionsmethode und nicht als bloße Designmethode konzipiert, so dass sie auch die Implementierungstätigkeiten explizit adressiert, und hierbei insbesondere den Übergang vom Entwurf in den Quellcode. Orientiert am Use Case-Konzept zeichnet sich die Methode durch ihre Systematik und durch eine stetig durchgeführte Echtzeitanalyse aus, um die harten Echtzeitanforderungen, denen die Software-Entwicklung in der Anwendungsdomäne Rechnung zu tragen hat, explizit zu berücksichtigen.

ViPER, der unterstützende Werkzeugprototyp, bildet den zweiten wesentlichen Bestandteil der Lösung. Das Werkzeug bietet generische Unterstützung für die Modellierungsaktivitäten der MeDUSA-Methode in Form einer graphischen UML-Modellierungsumgebung sowie einer spezifischen Unterstützung zur Spezifikation und Simulation natürlichsprachiger Anwendungsfallbeschreibungen. ViPER beinhaltet zudem eine spezifische methodische Unterstützung für MeDUSA, in dem es sowohl eine elektronische Dokumentation der Methode zur Verfügung stellt, als auch Implementierungen der von MeDUSA eingesetzten UML-Profile, Prüfungsmöglichkeiten für die im Rahmen der Methode eingesetzten UML-Modelle, sowie spezifische Assistenten für bestimmte, im Rahmen der Methode definierte Aufgaben.

Zusammen mit den zugrunde liegenden Sprachen, der Unified Modeling Language sowie der ANSI-C Programmiersprache, bilden MeDUSA und ViPER somit eine integrierte Methodologie, die auf gemeinsamen Konzepten und Prinzipien begründet ist. Ausdrücklich entworfen um den oben genannten Rahmenbedingungen gerecht zu werden, ist diese Methodologie damit in einer Domäne einsetzbar, für die bislang weitgehend methodische Ansätze fehlten.

Acknowledgments

While its cover sheets only depicts a single author's name, this thesis is the result of a process, in which several persons were involved, who should not remain unmentioned.

First and foremost Professor Lichter has to be named, who has given me the chance to prepare this thesis at the Research Group Software Construction. He has always been a benevolent adviser and has pretty much contributed to this work with his comments and thought-provoking impulses. I want to take the opportunity to express my explicit thanks for that, as well as for his ongoing support and encouragement. I also want to thank Professor Kowalewski for taking the role of the secondary adviser.

I further want to express my thanks to all involved current and former employees of the German ABB Corporate Research Center in Ladenburg, our cooperation partner. First and foremost Jan Suchotzki has to be named in this context, who - as a committed discussion partner and active contributor - has supported me very much in the conception of the initial MeDUSA method. I also want to thank Peter Müller, Lukas Kurmann, Dr. Dirk John and Dr. Detlef Streitferdt for their professional contributions and the always uncomplicated and friendly collaboration. My thanks are also dedicated to Tilo Merlin and Klaus Pose from the ABB Automation Products GmbH, who have delivered valuable input for the evaluation of the MeDUSA method.

For their participation in the development of the ViPER tool I also want to thank my colleagues Veit Hoffmann and Holger Schackmann, as well as all involved student workers and diploma and master thesis students. Mathias Funk, Andreas Walter, Zahid Sadal, Tauseef Ikram, Özgür Kevinç, Marcel Hermanns, Lars Grammel, Supaporn Simcharoen, Huy Do, Philip Ritzkopf, Mark Lehmacher, and Daniel Watermeyer have to be named in this context. They all have - through their dedication - strongly pushed the development of the tool.

I also want to explicitly thank my current and former colleagues Dr. Moritz Schnizler, Dr. Thomas von der Maßen, Thomas Weiler, Holger Schackmann, Veit Hoffmann, Andreas Ganser, and Malek Obaid for the always very cooperative and friendly atmosphere at the research group. Also Bärbel Kronewetter may not be left unmentioned. She has not only carried out a lot of organizational work for me in the last years, but has always also pampered me with cake and candy of all kind.

Last, I want to emphatically thank my parents, who have always and to every extent supported and encouraged me, and whom I have actually thanked far to seldom for that. I know, I have to count myself lucky.

Danksagungen

Wenngleich auf ihrem Deckblatt nur ein einzelner Autor vermerkt ist, so waren in den Entstehungsprozess dieser Arbeit doch eine ganze Reihe von Personen involviert, die nicht unerwähnt bleiben sollen.

Zu allererst sei hier Professor Lichter genannt, der mir die Möglichkeit gegeben hat, diese Arbeit am Lehr- und Forschungsgebiet Software-Konstruktion anzufertigen. Er war ein immer wohlwollender Betreuer und hat durch seine Anmerkungen und Denkanstöße zum Gelingen dieser Arbeit entscheidend beigetragen. Hierfür und für seine Unterstützung und Förderung möchte ich mich an dieser Stelle einmal ausdrücklich bedanken. Ebenfalls danken möchte ich Professor Kowalewski für die Übernahme des Zweitgutachtens dieser Arbeit.

Mein Dank gilt insbesondere auch den involvierten aktiven und ehemaligen Mitarbeitern des deutschen ABB Forschungszentrums in Ladenburg, unseres Kooperationspartners. Namentlich erwähnt sei hier - allen voran - Jan Suchotzki, der mich als engagierter Diskussionspartner und aktiver Mitgestalter bei der Konzeption der initialen MeDUSA Methode sehr unterstützt hat. Auch Peter Müller, Lukas Kurmann, Dr. Dirk John und Dr. Detlef Streitferdt möchte ich danken, für ihre fachlichen Eingaben und für die immer sehr unkomplizierte und freundschaftliche Art der Zusammenarbeit. Ebenso gilt mein Dank Tilo Merlin und Klaus Pose von der ABB Automation Products GmbH, die wichtigen Input für die Evaluierung der MeDUSA Methode geliefert haben.

Für ihr Mitwirken an der Entwicklung des ViPER Werkzeugs möchte ich zudem meinen Kollegen Veit Hoffmann und Holger Schackmann sowie den beteiligten studentischen Hilfskräften, Diplomanden und Master-Studenten ausdrücklich danken. Mathias Funk, Andreas Walter, Zahid Sadal, Tauseef Ikram, Özgür Kevinç, Marcel Hermanns, Lars Grammel, Supaporn Simcharoen, Huy Do, Philip Ritzkopf, Mark Lehmacher und Daniel Watermeyer sind hier zu nennen. Sie alle haben durch ihr Engagement die Entwicklung des Werkzeugs stark vorangetrieben.

Bei meinen jetzigen und ehemaligen Kollegen Dr. Moritz Schnizler, Dr. Thomas von der Maßen, Thomas Weiler, Holger Schackmann, Veit Hoffmann, Andreas Ganser und Malek Obaid möchte ich mich zudem für die immer angenehme und kollegiale Atmosphäre am Lehr- und Forschungsgebiet bedanken. Auch Bärbel Kronewetter darf in diesem Zusammenhang nicht unerwähnt bleiben. Sie hat mir im Laufe der Jahre nicht nur viele organisatorische Arbeiten abgenommen, sondern mich auch stets mit Kuchen und diversen anderen Süßigkeiten verwöhnt.

Schließlich gilt mein ganz besonderer Dank meinen Eltern, die mich immer und in jeder Hinsicht unterstützt und gefördert haben, und denen ich dafür eigentlich viel zu selten gedankt habe. Ich weiß, ich kann mich glücklich schätzen.

Contents

I Foundations **1**

1 Introduction **3**

1.1 Context 3

1.2 Outline 5

2 Terms & Definitions **7**

2.1 Embedded & Real-Time Computer Systems 7

2.1.1 Disambiguation 7

2.1.2 Classification 9

2.2 Embedded & Real-Time Software 11

2.3 Engineering Embedded & Real-Time Software 12

2.3.1 Software Engineering, Construction, and Development 12

2.3.2 Process, Method, and Methodology 14

2.3.3 Model, Modeling Language, Meta-Model 15

2.3.4 Model-Based and Model-Driven Software Engineering 16

3 Embedded & Real-Time Software Construction - An Inventory **19**

3.1 Historical Overview 19

3.1.1 Structured Analysis and Design (From the late 1970's up to the mid 1980's) 20

3.1.2 Object-Oriented Analysis & Design (From the mid 1980's to the late 1990's) . . . 22

3.1.3 A Decade of Great Diversity - CBSE, PLSE, MBSE (from the late 1990's to the early 2000's) . . . 24

3.2 State of the Art - The current situation in academia and industrial research 26

3.2.1 General-purpose Approaches . . . 27

3.2.2 Domain-specific Approaches . . . 29

3.3 State of the Practice - The current situation in industry . . . 34

3.4 Observations & Conclusions . . . 36

4 Definition of Scope - Problems, Challenges & Goals 39

4.1 Defining the Domain Scope . . . 39

4.1.1 Measurement Devices in Industrial Process Instrumentation . 40

4.2 Problems & Challenges . . . 45

4.2.1 Constraint-Inadequate Model-Based Software Construction . 45

4.2.2 Methodological Incompleteness and Discontinuity . . . 46

4.3 Definition of Goals . . . 47

II The MeDUSA-ViPER Methodology 49

5 A Model-Based Methodology 51

5.1 Sketching A Solution . . . 51

5.2 Related Work . . . 52

5.2.1 Methods . . . 53

5.2.2 Tools . . . 60

6 Method - MeDUSA 65

6.1 MeDUSA-Lifecycle . . . 65

6.2 MeDUSA-Definition . . . 67

6.2.1 SPEM 2.0 67
6.2.2 MeDUSA Method Content 70
6.2.3 MeDUSA Method Operations 102
6.3 Reflective Characterization 108

7 Languages - UML & ANSI-C **111**
7.1 MeDUSA UML Models 112
7.1.1 Requirements UML Model 113
7.1.2 Analysis UML Model 115
7.1.3 Design UML Model 118
7.2 MeDUSA UML Profiles 125
7.2.1 Requirements UML Profile 126
7.2.2 Analysis UML Profile 128
7.2.3 Design UML Profile 129
7.3 MeDUSA UML-to-ANSI-C Code Generation Schema 130
7.3.1 General Transformation Strategy 131
7.3.2 Classifiers within a MeDUSA Design UML Model 132
7.3.3 Generating Folders and Translation Units 135
7.3.4 Generation of Syntactic Elements 136

8 Tool - ViPER **159**
8.1 The ViPER Integrated Development Environment 159
8.2 ViPER MetiS- MeDUSA Methodical Support 166
8.2.1 ViPER MetiS Plug-in Architecture 167
8.2.2 ViPER MetiS Definition 168
8.2.3 ViPER MetiS Task Wizards 169
8.2.4 ViPER MetiS Cheatsheets Plug-In 176

8.2.5 Adoptions & Innovations 177

III Evaluation & Conclusion 179

9 Evaluation of MeDUSA 181

9.1 Continuous Evaluation - A Living Method 181

9.1.1 Initial (Pre-Published) Edition - 2005/2006 181

9.1.2 First (Published) Edition - 2007 183

9.1.3 Second (Published) Edition - 2008 185

9.2 Practical Evaluation Results of the Second Edition 187

10 Assessment of ViPER 191

10.1 Software Structure & Complexity Evaluation 192

10.2 Development Process & Infrastructure Characterization 197

11 An Appraisal of Achieved Results 201

11.1 Goal Attainment 201

11.2 Conclusion & Outlook 203

List of Figures

2.1 Embedded and Real-Time Computer Systems 9

2.2 Classification of Embedded & Real-Time Computer Systems 10

2.3 Classification of Embedded Real-Time Computer Systems 10

2.4 Disciplines of Software Engineering 13

2.5 Phases of Software Engineering 13

2.6 The System Triangle Metaphor (cf. [LL07]) 15

2.7 The ”Meta-Step” Pattern of Mega-Modeling (cf. [Fav06]) 16

2.8 Model-Based vs. Model-Driven Software Engineering 17

3.1 Historic Outline of Software Construction Approaches 21

4.1 Domain Scope Definition 40

4.2 Typical Mechanics/Hardware Architecture of a Measurement Device . 41

4.3 Measurement Principle of an Electromagnetic Flow Meter (cf. [GHH+04]) 44

5.1 The ViPER-MeDUSA Methodology 52

5.2 COMET Object-Oriented Lifecycle Model (cf. [Gom00]) 54

5.3 COMET Object/Class Structuring Criteria (cf. [Gom00]) 55

5.4 ROOM Hierarchical Modeling Example - System (cf. [SGW94]) . . . 57

5.5 ROOM Hierarchical Modeling Example - Subsystem (cf. [SGW94]) . 58

5.6 ROOM Behavioral Modeling Example (cf. [SGW94]) 59

5.7 IBM Rational Software Development Platform - Process Browser . . 61

5.8 IBM Rational Software Development Platform - Process Advisor . . . 62

5.9 Jaczone Waypointer - Activity Window 63

5.10 Jaczone Waypointer - Wizard Example 64

5.11 Jaczone Waypointer - Artifact Agent Example 64

6.1 The MeDUSA Life Cycle 66

6.2 Terminology of SPEM 2.0 (cf. [OMG08]) 68

6.3 SPEM 2.0 Meta-Model - Top-level Hierarchy 68

6.4 SPEM 2.0 Meta-Model - Method Content 69

6.5 SPEM 2.0 Meta-Model - Process 69

6.6 The MeDUSA Actor Taxonomy 73

6.7 MeDUSA Example Use Case Diagram 74

6.8 MeDUSA Example Global System States Diagram 74

6.9 MeDUSA Example Use Case Description 75

6.10 MeDUSA Example Use Case Details Diagram 76

6.11 The MeDUSA Object Taxonomy 77

6.12 The MeDUSA Trigger & Interface Object Taxonomies 77

(a) MeDUSA Trigger Object Taxonomy 77

(b) MeDUSA Interface Object Taxonomy 77

6.13 MeDUSA Example Context Diagram 78

6.14 MeDUSA Example Information Diagram 79

6.15 The MeDUSA Control Object Taxonomy 79

6.16 MeDUSA Example Inter-Object Collaboration Diagram (Communication) 80

6.17 MeDUSA Example Inter-Object Collaboration Diagram (Sequence) . 81

6.18 MeDUSA Example Intra-Object Behavior Diagram 82

6.19 MeDUSA Example Initial Structural Subsystem Design Diagram . . . 85

6.20 MeDUSA Example Initial Structural Subsystem Interface Design Diagram . . . 86

6.21 MeDUSA Example Initial Behavioral Subsystem Interface Design Diagram . . . 86

6.22 MeDUSA Example Initial Behavioral Subsystem Design Diagram . . 87

6.23 MeDUSA Example Consolidated Structural Subsystem Design Diagram 88

6.24 MeDUSA Example Consolidated Behavioral Subsystem Design Diagram . . . 89

6.25 MeDUSA Example Consolidated Structural Subsystem Interface Design Diagram . . . 90

6.26 MeDUSA Example Consolidated Behavioral Subsystem Interface Design Diagram . . . 91

6.27 MeDUSA Example Structural System Architecture Diagram 92

6.28 MeDUSA Example Behavioral System Architecture Diagram 93

6.29 MeDUSA Example Structural Detailed Design Diagram 95

6.30 MeDUSA Example Behavioral Detailed Design Diagram (State Machine) . . . 96

6.31 MeDUSA Example Initial Task Report (excerpt) 99

6.32 MeDUSA Example Initial Schedulability Report (excerpt) 100

6.33 The MeDUSA Requirements Workflow Pattern 102

6.34 The MeDUSA Analysis Workflow Pattern 103

6.35 The MeDUSA Architectural Design Workflow Pattern 104

6.36 The MeDUSA Detailed Design Workflow Pattern 105

6.37 The MeDUSA Implementation Workflow Pattern 106

6.38 The MeDUSA Workflow 107

6.39 The MeDUSA Architectural Design Phase 107

7.1 Examples of Key Artifacts contained in a MeDUSA Requirements UML Model . . . 114

7.2 Examples of Key Artifacts contained in a MeDUSA Analysis UML Model 116

7.3 Examples of Key Artifacts contained in the Subsystem fragment of a MeDUSA Design UML Model 121

7.4 Examples of Key Artifacts contained in the System fragment of a MeDUSA Design UML Model 124

7.5 MeDUSA Requirements Profile 127

7.6 MeDUSA Analysis Profile 128

7.7 MeDUSA Design Profile 129

7.8 UML Classifiers Hierarchy, according to [OMG07d] 133

7.9 General Transformation of Classifiers into Translation Units 135

7.10 ANSI-C Primitive Types UML-Library 137

7.11 Example - Transformation of Enumeration 137

(a) Instance Specification: Enumeration with Owned Literals . . . 137

(b) Generated Code: Corresponding Struct Declaration 138

7.12 Example - Transformation of Data Type and (Non-Structured) Class . 138

(a) Instance Specification: Simple Class with Attributes and Operations 138

(b) Generated Code: Corresponding Struct and Constructor & Destructor Function Declarations 139

(c) Generated Code: Member and Selector Function Declarations, corresponding to Owned Attributes 140

(d) Generated Code: Function Declarations for Owned Operations 140

(e) Generated Code: Macro Facade as Public Classifier Interface . 141

7.13 Example - Transformation of (Non-Structured) Class typing Part . . . 142

(a) Instance Specification: A Class typing a Part (Subsystem Decomposition) 142

(b) Generated Code: Member Declarations corresponding to Association Ends 142

(a) Instance Specification: Class typing Port (Provided Interface) . 144

(b) Generated Code: Function Declarations corresponding to Provided Interfaces 144

(a) Instance Specification: Instance Specification: Class typing Port (Required Interface) 145

(b) Generated Code: Function Pointer and Struct Pointer Member Declarations corresponding to Required Interfaces 145

7.16 Example - Transformation of subsystem Component 147

(a) Instance Specification: Subsystem Component 147

(b) Generated Code: Struct Member Declarations for Parts & Ports, Wiring 147

7.17 Example - Transformation of system Component 148

(a) Instance Specification: System Component 148

(b) Generated Code: Struct Member Declarations for Parts, Wiring of Parts within Constructor Implementation 149

7.18 Example - Transformation of a State Machine 150

(a) Instance Specification: State Machine 150

(b) Generated Code: Macro Definitions corresponding to States and Events, Corresponding Struct Declaration 151

(c) Generated Code: Function Implementations corresponding to Guards, Effects, and Transitions, Declaration of State-Transition Table as Function Pointer Array and Declaration of State Transition Function 152

7.19 Example - Transformation of a State Machine as Internal Behavior Specification 153

(a) Instance Specification: State Machine as Internal Behavior . . 153

(b) Generated Code: Struct Member Declaration and Function Implementation within Owning Behaviored Classifier 154

7.20 Example - Transformation of a State Machine as Interaction Protocol Specification 155

(a) Instance Specification: Protocol State Machine as Interaction Protocol 155

(b) Generated Code: Struct Member Declaration and Function Implementations within Port's Type 156

8.1 Eclipse SDK Architecture (cf. [DFK+04]) 160

8.2 ViPER IDE Architecture 161

8.3 ViPER IDE Platform Plug-in Architecture 162

8.4 ViPER IDE UML2 Plug-in Architecture 163

8.5 ViPER UML2 VME (Screenshot) 164

8.6 ViPER UML2 CodeGen (Screenshot) 164

(a) Code Generation Context Menu 164

(b) Code Generation Wizard 164

8.7 ViPER IDE NaUTiluS Plug-in Architecture 165

8.8 ViPER NaUTiluS (Screenshot) 166

8.9 ViPER MetiS Plug-in Architecture 167

8.10 ViPER MetiS Definition (Screenshot) - Integration into Help System . 168

8.11 ViPER MetiS Definition - Help *toc* File (generated) 169

8.12 ViPER MetiS Code Generation Wizard (Screenshots) 170

(a) Task Wizard Description Page 170

(b) Task Wizard (Precondition) Validation Page 170

(c) Task Wizard Generation Sources And Options Selection Page . 170

(d) Task Wizard Generation Processing Page 170

8.13 Detailed Class Design of the ViPER MetiS Task Wizard `core` Package 172

8.14 Detailed Class Design of ViPER MetiS Task Wizard - Generation & Transformation Wizard 174

8.15 ViPER MetiS TaskWizards Traceability Meta-Model 175

8.16 ViPER MetiS Cheatsheets (Screenshot) 176

8.17 ViPER MetiS CheatSheets - MeDUSA Workflow Composite Cheatsheet 177

9.1 MeDUSA Pre-Published Edition - Workflow 182

9.2 MeDUSA First Edition - Workflow Patterns 184

(a) Requirements Modeling 184

(b) Analysis Modeling 184

(c) Architectural Design Modeling 184

(d) Detailed Design Modeling 184

10.1 Quality Characteristics according to IEC/ISO 9126 (cf. [IEC01]) . . . 191

10.2 ViPER IDE Architecture Definition within SonarJ 194

10.3 Abstraction-Instability Graph for ViPER IDE Features 196

10.4 ViPER Download Site (Screenshot) 198

List of Tables

7.1 Usage of Classifiers within a MeDUSA Design UML Model 134

10.1 ViPER IDE - Size Metrics Evaluation Results 193

10.2 ViPER IDE - Coupling & Cohesion Metrics Evaluation Results 195

Part I

Foundations

Chapter 1

Introduction

1.1 Context

The work presented in this thesis was conceived in an industrially funded research project, being conducted at the Research Group Software Construction of the RWTH Aachen University, in close cooperation with the German ABB Corporate Research Center and selected subsidiaries of the ABB Business Unit Instrumentation, located in Germany, Italy, and the United Kingdom.

The research venture was initiated in 2003 with the goal to introduce state-of-the-art model-based software engineering techniques and technology to the domain of small embedded & real-time systems, targeting first and foremost measurement field devices within the industrial automation application area, as they are produced within the ABB Business Unit Instrumentation. Due to the organizational constraints being faced, identification of an appropriate development method, being commonly applicable throughout ABB Business Unit Instrumentation was regarded to be of the highest importance in order to gain a common terminology and procedure across all involved subsidiaries, which up to then - due to the historic course of the ABB Business Unit Instrumentation - had more or less worked in an unrelated, stand-alone manner.

MeDUSA, the model-based construction method, which forms an integral part of the herein presented approach, was developed under this premise. The initial stimulus for its development originated from the evaluation of several pilot projects, being jointly conducted in 2003 and 2004 at the ABB Corporate Research Center and ABB Automation Products GmbH to gather experiences about the practical applicability of the *Concurrent Object Modeling and Architectural Design Method (COMET)* [Gom00], MeDUSA's direct predecessor, to the development of field device software. The evaluation revealed that while *COMET* can be regarded as a promising method, it shows some noticeable shortcomings and is thus not capable to meet the very special characteristics, being faced in the industrial automation application area [NMSL04].

The need for a customized method, being capable to meet the very special characteristics, was thus identified, resulting in the definition of the MeDUSA method. Being first only internally documented and evaluated, the *First Edition* of the method was published in 2007 [NL07a]. The *Second Edition*, a complete revision of the *First Edition*, was published in 2008 [NL08] and incorporates many additional experiences gathered from the application of MeDUSA within a pilot project conducted within ABB Business Unit Instrumentation in 2007 and 2008. What is published herein basically corresponds to the *Second Edition*, while some recent research experiences, which are related to the modeling of timing and concurrency constraints within use case models as well as to the real-time analysis based on use case models, have been incorporated (cf. [Rit08] for details).

Being aware that a model-based methodical approach would not be practically applicable without adequate supporting tools, investigation was started to identify such tools already in late 2004. This was done in terms of several ABB-internal tool evaluations related to UML modeling and code generation tools, which were guided by the Research Group Software Construction. In this context, the need for an own tool prototype, which could be used to quickly built-in and demonstrate new research ideas and experiences, not already built into market available tools, was identified.

The development of ViPER was thus initiated. Having started in late 2004 as a simple UML state machine editor to evaluate the applicability of some frameworks provided by the Eclipse Foundation [Eclipse], the UML modeling related capabilities of ViPER were continuously extended in the following years [Wal07][Ikr08][Do08], adding in particular also support for the integrated modeling of narrative textual use case description [Wal07]. The tool was also equipped with flexible and customizable UML-to-ANSI-C code generators [Fun06][Kev07][FNL08], and finally with dedicated methodical support for the MeDUSA method [Her07][Rit08]. While ViPER was never meant to be practically applied within ABB Business Unit Instrumentation, the tool has thus been very useful to demonstrate how the application of the MeDUSA method could be adequately supported.

It is a central statement of this thesis work that model-based software engineering, due to its increased systematics and traceability, bears a great potential, especially within the domain of embedded & real-time systems. In particular within this domain, the increased abstraction and analyzability that is inherent to models, which then serve as the central development artifacts, offer great advantages compared to traditional engineering approaches. However, these benefits can - as it is the second central statement of this thesis work - only be completely unleashed if an integrated methodology, formed by a systematic and concise method, adequate underlying languages, and appropriate supporting tools is provided. The presentation of an integrated methodology is thus the central concern of this thesis work.

1.2 Outline

This thesis work is split into three main parts. The first part, entitled *"Foundations"* provides the necessary background to enable understanding and facilitate access to the herein presented methodological approach. It is initiated in Chapter 2 with the definition of basic terms and the explanation of fundamental concepts and principles, referred to throughout the work. An inventory of approaches related to the construction of software for embedded & real-time systems - in the past and present - is attached in Chapter 3, emphasizing on methodical approaches, but as well mentioning important contributions related to languages and tools. It includes an explicit investigation about the current state-of-the-art and state-of-the-practice of software construction within the domain of embedded & real-time systems, as well as an in-depth discussion on the discrepancy between the two. Based on this, the scope of the herein presented work is then explicitly defined within Chapter 4, in terms of what systems are actually covered and what technical and organizational constraints are to be faced in the targeted application domain. Problems and challenges, which arise within the defined domain scope are then quoted, resulting in the formulation of goals for the herein presented methodological approach.

The main part, entitled *"The MeDUSA-ViPER Methodology"*, is then concerned with the demonstration of the developed solution, which is formed by the software construction method MeDUSA, its underlying languages UML and ANSI-C, and the supporting tool ViPER. It is initiated within Chapter 5 by a draft of the presented solution, and a quotation of closely related work with respect to methods and tools, before in Chapter 6 the definition of the MeDUSA method is then provided in detail. An overview of the UML language subset used within MeDUSA's UML models, and a definition of the required UML extensions, defined by means of MeDUSA specific UML profiles, is provided in Chapter 7, together with a detailed specification on how the employed language constructs can be adequately transferred into a procedural implementation within the ANSI-C programming language. The main part is concluded by Chapter 8, which introduces the ViPER tool, the third integral part of the presented methodology.

The concluding part, being concerned with *"Evaluation & Conclusion"*, is initiated with an individual evaluation of both, the MeDUSA method as well as the ViPER tool within Chapters 9 and 10 respectively. As far as MeDUSA is concerned, the course of the method is first outlined, as it reflects the results a continuous evaluation of the method in the context of several pilot projects, conducted within ABB Business Unit Instrumentation. A presentation and interpretation of gathered evaluation results is then attached. Following this, the ViPER tool is evaluated in terms of aspects of product quality. Here, the structure and complexity of the tool, as well as the process, which is used for its development, are quoted, as both provide essential indications on its *inner* quality characteristics. The thesis is concluded in Chapter 11 with a detailed reflection on the actual contributions of this work, an assessment related to the attainment of the previously formulated goals, as well as a conclusive summary and an outlook on future work.

Chapter 2

Terms & Definitions

2.1 Embedded & Real-Time Computer Systems

From a broad perspective, there is quite an intuitive understanding on the notion of an *embedded computer system* and a *real-time computer system*. However, when trying to define precisely what an embedded or real-time system is, definitions vary in manifold ways. This may be caused by the fact that the domain of embedded & real-time systems is a quite broad one, and embedded as well as real-time systems can be found in various application areas, like automotive, aerospace & defense, healthcare, telecommunications, consumer electronics, or industrial automation, all leading to very different system characteristics.

Nevertheless, to define the scope of this thesis work, a disambiguation of the central terms *embedded computer system* and *real-time computer system* is essentially necessary, and is thus performed subsequently. A classification of those systems matching the provided definitions is presented afterwards. This will then support the definition of the scope of those systems, the approach presented herein, is applicable to.

2.1.1 Disambiguation

The IEEE Computer Society defines an ***Embedded Computer System*** as *"a computer system that is part of a larger system and performs some of the requirements of that system [...]"* [IEEE91]. According to Michael Barr's Embedded Systems Glossary [Bar], an embedded computer system is furthermore defined to be *"a combination of computer hardware and software, and perhaps additional mechanical or other parts [...]"*.

The combination of these two rather general definitions seems to be a usable definition, as they stress on two important aspects of an embedded computer system: the embeddedness into a larger - the embedding - system, and its hybridity in terms of

hardware, software, and probably mechanics. An embedded computer system can be further characterized by the following properties (cf. [Mar03]):

- *Dedication* - Embedded computer systems are usually *dedicated* to a certain application or function. That is, in contrast to general-purpose computer systems, the functionality is usually tied up into the hard- and software in an unchangeable manner.
- *Efficiency* - Embedded computer systems have to be *efficient*. This might involve weight and cost, it almost always includes efficiency in terms of resource consumption at run-time, which can be denoted in terms of energy consumption, memory consumption, and computation time.
- *Dependability* - Embedded computer systems often have to be *dependable*, as they are used in safety-critical environments. According to Marwedel [Mar03], dependability can be expressed in terms of reliability, maintainability, availability, safety, and security.

Complementary, a ***Real-Time Computer System*** may be defined according to Kopetz [Kop97] as *"a computer system in which correctness of the system behavior depends not only on the logical results of the computations, but also on the physical instant at which these results are produced"*.

A real-time computer system is - as an embedded computer system - always part of a larger system - the so called real-time system - which can in turn be decomposed into different clusters, namely the real-time computer system, the controlled object, and a (human) operator. As this implies, most *real-time computer systems* have the task to control an object in their environment, what means to influence the state of the respective controlled object. The simple observation of an object in the environment without taking influence may also be a legal task for a real-time system (i.e. for a real-time measurement system), so it is reasonable to broaden the definition to that extend.

Besides the essential property of *dependability*, which can as well be attributed to real-time computer systems, they may be further characterized by the following properties (cf. [Kop97]):

- *Reactivity* - A real-time computer system has to be *reactive*. In order to perform its respective control or operation task, it has to react to stimuli from its environment (operator or controlled object) with computed results. Those results usually have to be produced within a certain time interval, prescribed by the environment of the real-time computer system. Dependent on the urgency of the result, real-time systems may be classified into *hard real-time* and *soft real-time* systems. While soft real-time systems may accept the result even if a given deadline has been passed, hard real-time systems have to produce the respective result at the correct instant, possibly causing harm if the expected result is not computed within the defined deadline.

- *Responsiveness* - A real-time computer system has to be *responsive*. Results to environmental stimuli have to be computed with a given certainty, including exceptional situations of high demand or even fault. According to the predictability of the responsiveness of a real-time computer system, it may be characterized as being *guaranteed response* or *best-effort*, depending on whether adequate reasoning about the design can be done based on a specified fault- and load-hypothesis or not.
- *(Resource-)Adequacy* - A real-time computer system has to *adequately* handle the provision of its *resources*. That is, even in situations of high load or in fault scenarios, the availability of sufficient resources has to be ensured. Regarding the appropriateness of resource handling, real-time computer systems may thus be characterized as being either *resource-adequate* or *resource-inadequate*.

Dependent on whether a real-time system can be best described to start its processing and communication activities either in case of the occurrence of an environmental *event* (excluding regular clock ticks) or at predetermined points in *time*, it can be further characterized as being either *event-triggered* or *time-triggered*[1].

2.1.2 Classification

As it can be easily concluded from the preceding definitions, the intersection between the set of embedded computer systems and the set of real-time computer systems is actually not empty. In fact, as indicated by Figure 2.1, most of the embedded software systems are probably also real-time computer systems, and most of the real-time computer systems are probably also embedded computer systems.

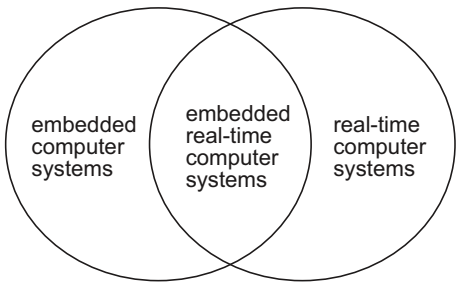


Figure 2.1: Embedded and Real-Time Computer Systems

Besides those systems falling into both categories, which will be referred to as ***Embedded Real-Time Computer Systems*** in the following, there are also a couple of embedded systems, which do not have real-time properties, and there are also real-time

[1] Note that a real-time system is not - by nature - *event-triggered* or *time-triggered*, but that this is determined from the control paradigm chosen to describe the reactive behavior of the system. In this context, a real-time computer system may as well be regarded as being *hybrid*, if it can neither be *represented and analyzed with sufficient precision either by the methods of the continuous systems theory or by the methods of the discrete systems theory* (cf. [Lun02])

systems, being not embedded. As denoted by Figure 2.2, embedded computing systems like automated tellers or set-top boxes, embedded communication & networking systems like print servers or fax devices may be named as application examples for non-real-time embedded computer systems. The group of non-embedded real-time systems is probably larger. According to [Kop97], it may be classified into plant automation systems and multimedia systems.

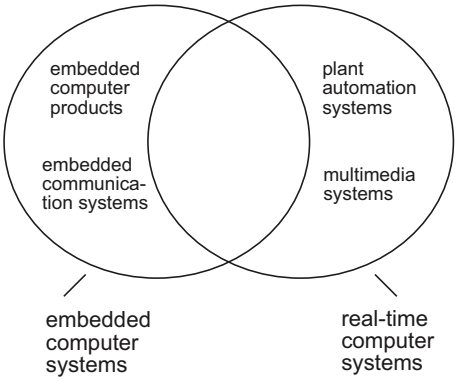


Figure 2.2: Classification of Embedded & Real-Time Computer Systems

However, the borders are getting more and more fuzzy between real-time and non-real time embedded systems, and most embedded systems that can be found today, do indeed show at least a portion of real-time behavior, so that the group of computer systems that actually apply to both categories is probably also the largest group of systems. Such systems are primarily used to measure or to control. They are needed in various application areas like industrial automation, automotive, consumer electronics, or even aerospace & defense. Besides being probably the quantitatively largest group, measurement & control systems also have the largest historical background, as they were traditionally used as replacements for electronic hardware approaches in control and feedback control engineering (cf. [Sch04]).

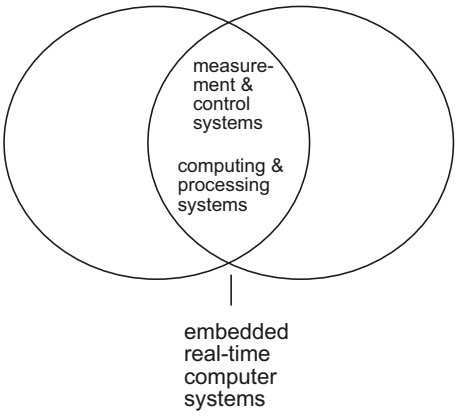


Figure 2.3: Classification of Embedded Real-Time Computer Systems

Besides them embedded real-time computing & signal processing systems may be identified, whose primary task is not to measure or to control (cf. Figure 2.3) but to process signals or perform other arbitrary computation tasks. Those systems can traditionally be found in the telecommunications industry, but are also widely spread through other application areas like consumer electronics, medicine, or robotics.

It has to be mentioned that besides the classification schema applied here, which pretty much takes into account the purpose of the respective systems and thus classifies embedded & real-time systems along their application areas, several contrasting classification schemata can be found in literature. Most of those schemata however concentrate on certain predominant of the above outlined characteristics, resulting in classification terms like *dedicated system*, *dependable system*, *reactive system*, *continuous system*, or *hybrid system*, which is not regarded to be appropriate here.

2.2 Embedded & Real-Time Software

Having clarified the notion of embedded and real-time computer systems, ***Embedded Software*** as well as ***Real-Time Software*** may be simply defined as the software running on an embedded respectively real-time computer system. The term ***Embedded & Real-Time Software*** is therefore used as the general term to refer to the software running on embedded & real-time systems. This is reasonable, because the characteristics of the respective software running on an embedded & real-time computer system are mostly dependent on the characteristics of the underlying hardware. In particular, dependability and efficiency as well as responsiveness and resource-adequacy of an embedded & real-time computer system have strong influence on the contained software in terms of its non-functional requirements.

According to this, the software of an embedded & real-time computer system may be characterized by a rather strong restrictiveness related to the resources being consumed. That is, compared to other software, embedded & real-time software typically has to be less demanding regarding computation time and memory consumption. Related to the reactivity and responsiveness constraints imposed on an embedded & real-time computer system, the software running on it typically shows some sort of reactive code. The necessity of parallel processing causes that concurrently running parts may usually be identifiable. Dependability further causes that safety-related functionality can be found, related to fault detection and recovery. Closely related, mostly due to the strong non-functional restrictions that are imposed to an embedded & real-time computer system by its environment, as well due to its dedicatedness, a strong commitment to be compliant with industry standards is characteristic for embedded & real-time software.

A classification of embedded & real-time software in terms of its purpose - like it was done in case of the overall system - might lead to a distinction into embedded & real-time *System Software* and *Application Software*, where system software is understood to be responsible of interfacing with hardware and running the necessary services for

user-interfaces and applications, while application software performs productive tasks for users. However, such a classification may be problematic within the domain of embedded & real-time software, as a clear separation of system-related (i.e. hardware-related) and user-related functionality is often impossible. Indeed, most of the software running on an embedded & real-time system will probably show a mixture of both aspects, especially if no underlying operating system can be found and the application software is also responsible of interfacing to the underlying hardware. However, it is reasonable to distinct a bootloader or an operating system from the software realizing the dedicated application (in case such a distinction can be made), so system software and application software will be referred to in the above sense, having in mind that such a clear separation will not always be adequate or possible.

2.3 Engineering Embedded & Real-Time Software

Before defining the scope of the herein presented approach, a disambiguation of the term *Software Engineering* and of closely related terms is regarded to be necessary. This will be done in the succeeding section, followed by a clarification of the terms *Process*, *Method*, and *Methodology*, as well as of terms and concepts related to model-based software engineering.

2.3.1 Software Engineering, Construction, and Development

Friedrich Ludwig Bauer is often nominated as the originator of the term *Software Engineering*. It was on the NATO conference, held in Garmish, Germany, in 1968, where he formulated his vision of *"the establishment and use of sound engineering principles in order to obtain economically software that is reliable and works efficiently on real machines."* [Bau75]. To overcome the so called *Software Crisis*, Bauer regarded the application of profound engineering principles as the essential step to systematize the unsystematic, unreliable and error-prone software development, which could not deal with the ever rising complexity. Today, software engineering has become a mature field in computer science, although the software crisis may still be regarded to not have completely ended, as the problem of increasing complexity is also noticeable today, especially in the field of embedded & real-time systems.

In line with the initial vision of Bauer, the IEEE Computer Society defines ***Software Engineering*** as *"the application of a systematic, disciplined, quantifiable approach to the development, operation, and maintenance of software; that is, the application of engineering to software."* [IEEE91], which is regarded to be a reasonable definition.

As denoted by Figure 2.4, it can be divided into four major disciplines, namely *Software Construction*, *Software Quality Management*, *Software Configuration & Change Management*, and *Software Project Management*.

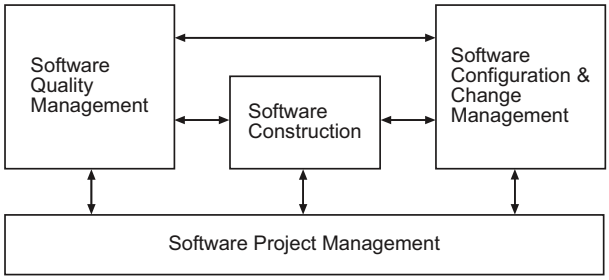


Figure 2.4: Disciplines of Software Engineering

It further covers all phases of the overall *Software Lifecycle*, which can - on a coarse-grained level - be split into *Software Development*, *Software Operation & Maintenance*, and *Software Decommissioning*, as denoted by Figure 2.5.

Figure 2.5: Phases of Software Engineering

In this sense, ***Software Construction*** is defined to comprise all *constructive* activities and techniques related to the development and maintenance of software, including requirements engineering, analysis, design, and implementation[2]. Software quality management, sofware configuration & change management, and software project management are defined accordingly to comprise all management activities of the respective management areas. In contrast to this, ***Software Development*** is defined as *"the period of time that begins with the decision to develop a software product and ends when the software is delivered"* (cf. [IEEE91]).

While *Software Development* thus denotes a phase inside the overall software lifecycle, *Software Construction* refers to one of the covered disciplines. When thus referring to an approach as a *Software Development Approach* within this work, no assumption on whether the approach does only cover constructive activities (related to development) or is indeed a more wholistic approach covering management activities as well, is made. In addition, when referring to a *Software Construction Approach*, it is not further stated which lifecycle phases are indeed covered, but the application of constructive activities and the disregarding of management activities is assumed. This also

[2]Note that - as so often - there is a multiplicity of competing definitions related to software engineering terminology. The IEEE Software Engineering Body of Knowledge (SWEBOK) [IEEE04] for example defines five knowledge areas to capture the *constructive* software engineering activities, namely *Software Requirements*, *Software Design*, *Software Construction*, *Software Testing*, and *Software Maintenance*, out of which *Software Construction* is defined to the *"detailed creation of working, meaningful software through a combination of coding, verification, unit testing, integration testing, and debugging."* The SWEBOK does however not define a term to refer to the entirety of all activities, related to the construction of software, so the term *Software Construction* will instead be used to refer to this, while the set of all implementation level activities will be denoted as *Software Implementation* instead.

implies that speaking of a *Software Engineering Approach*, no consice determination on which respective disciplines and lifecycle phases are covered, is made.

2.3.2 Process, Method, and Methodology

As with several other terms, a clear consensus on the meaning of the term *Software Engineering Method* has not been reached. Often, it is enmeshed with the related term *Software Engineering Process*, which is defined by the IEEE as *"a sequence of steps performed for a given purpose"*. This is a quite general and expressionless definition and does not allow a clear and precise differentiation. Other provided definitions are fuzzy as well. Humphrey for example defines a *Software Engineering Process* as *"the total set of software engineering activities needed to transform a user's requirements into software"* [Hum89], which is similarly imprecise. The same holds for the definition of the term *Method* itself, which may be exemplarily quoted from the American Heritage Dictionay as *"a means or manner of procedure, especially a regular and systematic way of accomplishing something"* [AHD04].

It is clear that a concise definition of the term *Software Engineering Method* thus requires that in turn a clear definition of the term *Software Engineering Process* is provided. Despite the fuzziness and diversity that is apparent in the above definitions, there seems to be a quite common consensus on the fact that a *Software Engineering Process* refers to somewhat more wholistic and less precise, in such a sense that it usually covers all (or most of the) disciplines and lifecycle phases of software engineering, while often not specifying any concrete principles or techniques in detail. A *Software Engineering Method* on the other hand does usually not cover all software engineering disciplines or phases, but indeed provides detailed techniques and principles to reach a certain sub-goal.

Trying to capture this intuitive understanding into a definition, a ***Software Engineering Process*** may be regarded as *a coarse-structural definition of the logical and temporal application of software engineering activities to develop or maintain software*. A ***Software Engineering Method*** may then be further defined as a *practical, systematic, and detailed procedure to accomplish a certain software engineering goal, in line with software engineering principles and by applying software engineering techniques*. Following this, a ***Software Design Method***, can thus be defined as a method that supports the developer with the creation of a software design. Accordingly, a ***Software Construction Method*** may be characterized as a method that covers all software construction actitivies, and whose goal therefore is a systematically developed implementation.

Having defined the notion of a *Software Engineering Method*, a ***Software Engineering Methodology*** may be defined as a *wholistic approach combining an approved method, a workable notation, and appropriate tools, all being tied together by a set of common principles and concepts*[3]. Ludewig and Lichter [LL07] illustrate this quite demonstra-

[3] It may have to be pointed out in this context that the term *methodology* is sometimes also found in

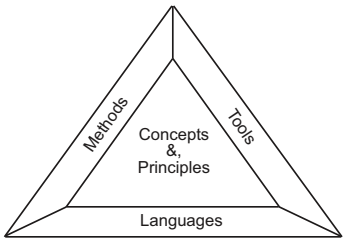


Figure 2.6: The System Triangle Metaphor (cf. [LL07])

tively by means of their *System Triangle* metaphor, which is depicted in Figure 2.6. Only if method, notation, and tool are grounded on common concepts and principles, they form a coherent system, which may then be referred to as a methodology.

2.3.3 Model, Modeling Language, Meta-Model

Engineering has a long tradition in the use of models, which reaches even back into antiquity. It was already in Ancient Greece and Rome, that models were applied to support the construction of buildings and machinery (cf. [Sel03b, pp 1-16]). Then and now, engineers use models for two main purposes, namely to *describe* an already existing system, or to *prescribe* how a not yet existing system has to be constructed (cf. [LL07]). Taking these two purposes, the definition of a ***Model*** as *"a description or specification of a system and its environment for some certain purpose"* [OMG03] seems to be natural and intuitive.

A *good* engineering model, according to Selic, has to fulfil some key requirements in terms of abstraction, intuition, accuracy, and expense (cf. [Sel03b, pp 1-16]). That is, it has to be *abstract* in a sense that it hides all details irrelevant to the domain of concern. It further has to be *intuitive*, so comprehension is easily accomplishable. A good engineering model furthermore has to be *accurate*, in such a sense that it must faithfully represent all interesting aspects of the modeled system and that it allows accurate predictions about interesting properties. Last, it has to be *inexpensive* in terms of its construction effort, compared to the actual system. It thus supports the intuitive understanding and effective reasoning about a complex system by hiding unwanted details and by concentrating on the interesting properties. It further allows to *"understand the interesting aspects of a complex system before going through the expense and effort of actually constructing it"* [Sel03b, pp 1-16].

Like in traditional engineering, models also play a central role in software engineering. In fact most of the artefacts, a software engineer has to deal with, are indeed models - even if they differ in their degree of formality. This holds for a requirements specification as well as for the source code of the resulting program, which is literally taken an implementation model. In such a sense, even the software itself can be regarded

the meaning of *the study of methods*. This is however not the meaning that will applied here.

as a model. The intense use of models in software engineering of course requires a certain degree of formality within the employed models, especially if tools are used. That is, a (formal) engineering model has to be conformant to some (formal) specification. According to *mega modeling* theory [Fav06], it may be equally stated that a (formal) model has to be an element of a ***Modeling Language***, which - inspired by language theory, where *"a language is formalized as a set of sentences"* - can in turn be regarded as *"a set of [valid] models"* [Fav05].

The complete specification of a modeling language, referred to as a ***Modeling Language Specification*** in the following, usually has to cover the definition of its ***abstract*** and ***concrete syntax***, as well as its ***static*** and ***dynamic semantics***. While the abstract, notation independent syntax defines the constructs that build up the modeling language, the static semantics, sometimes also referred to as *well-formedness*, specifies how instances of the defined constructs are related to each other. The actual meaning of a (well-formed) construct is defined by the dynamic semantics, while the (graphical) notation of the covered constructs is in turn defined by the concrete syntax.

While the dynamic semantics of a modeling language is usually defined in terms of natural language, as it is hard to formalize, and while the concrete syntax is normally defined by some informal graphical illustrations, the abstract syntax and the static semantics are usually formally defined, by means of a (formal) model, the so called ***Meta-Model***. According to *mega modeling* theory, such a meta-model may be regarded as *"a model of a modeling language"* [Fav06], or - from another viewpoint - as a model of *"a set of models"*.

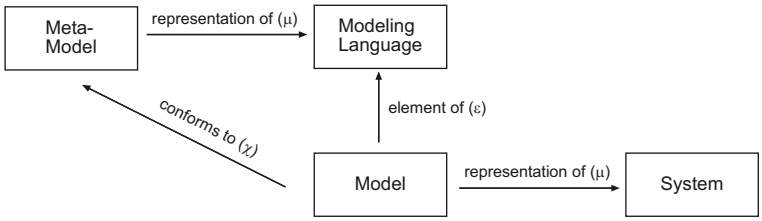


Figure 2.7: The "Meta-Step" Pattern of Mega-Modeling (cf. [Fav06])

Conformancy of a model to a meta-model can is this sense be easily exemplified by the so called *"Meta-Step" or "Z"* pattern of mega-modeling theory, which is demonstrated in Figure 2.7. That is, a model is conformant to a meta-model, if the model is a member of the set of models, which is represented by the meta-model in turn.

2.3.4 Model-Based and Model-Driven Software Engineering

Different to all other engineering disciplines, software engineering has the outstanding property that the developed product, the software, is an immaterial one. The engineering of software - unlike any other engineering discipline - thus offers the exceptional possibility to *"directly evolve models into full-fledged implementations without chang-*

ing the engineering medium, tools, or methods", as Selic points it out [Sel03a]. It thus seems to be a natural and promising approach to found the engineering of software on (formal) engineering models, as it might help to *"raise the level of abstraction of specifications to be closer to the problem domain and further away from the implementation domain by using modeling languages with higher-level and better behaved constructs"* and as it might further help to *"raise the level of automation by using computer technology to bridge the semantic gap between the specification (the model) and the implementation (the generated code)"* [Sel06].

Two main terms have emerged, referring to engineering approaches that employ models in the above sense, namely model-based and model-driven software engineering. As so often, a concise definition and a clear separation of both terms is not commonly agreed on.

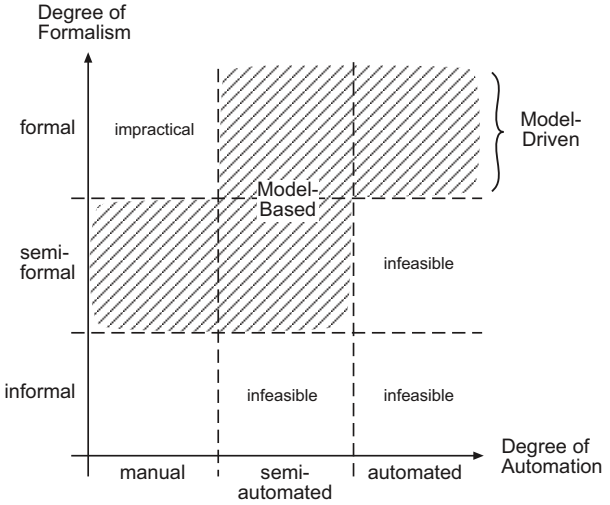


Figure 2.8: Model-Based vs. Model-Driven Software Engineering

However, as indicated by Figure 2.8, it seems that there is a slight subliminal agreement inside the software engineering community to use the term ***model-based*** software engineering in a broad sense to refer to all approaches that rely on (formal) engineering models as primary engineering artefacts, and to use the term ***model-driven*** software engineering exclusively to refer to those model-based approaches, that further achieve a strong automation of the engineering process.

Model-Based Software Engineering (MBSE) can thus be defined as *software engineering, making systematic use of (formal) engineering models as primary engineering artifacts throughout the overall engineering life-cycle*. ***Model-Driven Software Engineering (MDSE)*** accordingly as *model-based software engineering, formalizing and automating the engineering process in terms of (semi-)automated model-to-model transformation and model-to-code generation*.

Chapter 3

Embedded & Real-Time Software Construction - An Inventory

To be able to understand and evaluate the presented embedded & real-time software construction approach, the current situation of software construction in this respective domain has to be sketched. This will be done from two perspectives, an academic *State-of-the-Art* as well as an industrial *State-of-the-Practice* viewpoint, as there is quite a discrepancy between these two.

Before sketching the current situation however, a historic survey on the topic will be provided. This is needed, as the basic principles and concepts underlying current development approaches often have quite a history, and deep understanding, as well as examination and evaluation of such an approach is often not possible without founded historical background.

3.1 Historical Overview

Reflecting the history of embedded and real-time software construction approaches, one has to concern oneself with a multiplicity of methods, languages, and tools. However, even if an approach delivered all of those kinds, forming an integrated methodology, the principles and concepts characterizing it, can be inferred to the greatest extend from the subsumed method and its related notation. As further - at least up to the unification of several notations by the *Unified Modeling Language (UML)* in the late 1990's - each construction method had its own proprietary notation, the following outline will predominantly concentrate on the method adherent to a construction approach, and will in general not regared the associated notation and related tools explicitly. Where however notations are shared between methodological approaches, or where notations of former approaches are reused, or where tools are considerably worth to be mentioned, this will be pointed out explicitly.

While embedded & real-time software has - as already pointed out - its very special characteristics, a lot of the underlying concepts and principles, as well as related notations and methds are applicable to the development of software in other domains as well. In fact, most of the approaches being applied to embedded & real-time software have originated from the domain of classical industrial information systems. This is why the following historic outline will not only concentrate on embedded development approaches but will try to draw a relatively complete picture, enclosing all those approaches, that either directly or indirectly influenced software development in the embedded & real-time domain.

The graphical outline provided by Figure 3.1 might help the reader to reflect the following sections[1]. It shows all approaches being explicitly named in the following in their respective historic context, classifies them according to their predominant development paradigm and their membership to the embedded & real-time or other development domains. It further sketches relationships between those approaches that influenced one another.

3.1.1 Structured Analysis and Design (From the late 1970's up to the mid 1980's)

While systematic approaches to *Structured Programming* have already been developed in the late 1960's and the early 1970's [DDH72], and while basic design principles like *stepwise refinement* [Wir71] or *information hiding* [Par72] have been formulated already during this time as well, it was not before the mid 1970's that systematic software development methods were developed.

Inspired by those *Structured Programming* approaches of the early 1970's, the concepts of procedural programming were adopted to the modeling of the system design in the mid 1970's, leading to what became known as *Structured Design*. *Jackson Structured Programming* [Jac75], *Structured Design* by Yourdon and Constantine [YC79], and the *Warnier/Orr method* [Orr78] are prominent software design methods that appeared during that time. They introduced several notations to capture functional and data decomposition as well as data flow. Having applied programming concepts to modeling the system design, it was a next natural step to transfer those concepts from modeling of the solution domain to modeling of the problem domain as well. With the end of the 1970's the notion of *Structured Analysis & Design* was submitted by the works of DeMarco [DeM79] as well as Gane and Sarson [GS79].

It was as well during the mid/late 1980's that the first design methods especially targeting the real-time domain emerged. Ward and Mellor [WM85] as well as Hatley and Pirbhai [HP88] may be named in this context. Their approaches, being known since as *Real-Time Structured Analysis & Design*, introduced the notation of state diagrams

[1] The Figure was mainly inspired by similar graphical outline on the history of the *Unified Modeling Language*, which can be found in [Oes01]. Valuable input on the history of object-oriented methods, as well as on those approaches especially targeting the embedded and real-time domain was also taken from [Gra91] and [Gom00] respectively.

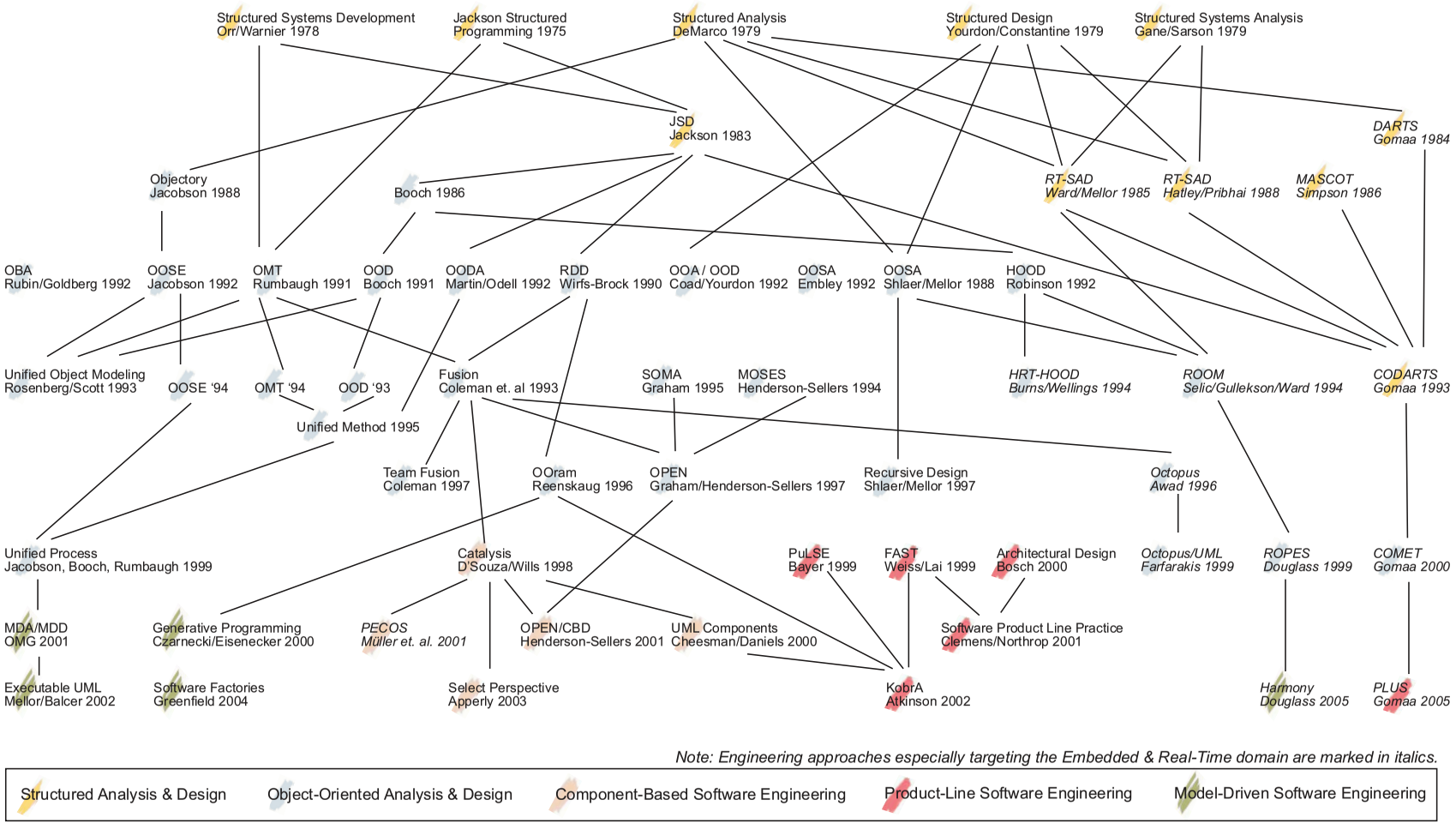


Figure 3.1: Historic Outline of Software Construction Approaches

to be able to specify stateful behavior. In this context, Harel also has to be mentioned, who invented the notation of *State Charts* [Har88]. Other contributions targeting the real-time domain, which appeared during the mid 1980's, having significant impact on later design methods, are the *Modular Approach to Software Construction Operation and Test (MASCOT)* by Simpson [Sim86] and the *Design Approach for Real-Time Systems (DARTS)* by Gomaa [Gom84], who formalized the modeling of concurrent tasks and their respective interfaces, and provided means to structure a real-time system into such concurrent tasks.

3.1.2 Object-Oriented Analysis & Design (From the mid 1980's to the late 1990's)

One of the most significant contributions of the early 1980's is probably *Jackson System Development* [Jac83]. Here, Jackson relieved himself from the concept of functional decomposition that had embossed those *Structured Analysis & Design* methods of the 1970's, and proposed to model real-world entities and events to capture the problem domain, which is why it is sometimes regarded as one of the first object-oriented analysis methods - or at least as their direct predecessor.

Following this new programming paradigm of object-orientation, first truly object-oriented methods were then developed in the late 1980's and early 1990's. Similar to as it was done in terms of their *Structured Design* anchestors, object-oriented programming concepts were transferred to the modeling of system design. Booch's *Object-Oriented Design (OOD)* is probably the first such approach. Having first published his ideas in the object-based world of Ada-based systems [Boo86], Booch extended his approach to a full *Object-Oriented Design* approach in the following years [Boo91] [Boo94]. As with the *Structured Design* approaches a decade before, a shift from modeling of the solution domain to modeling of the problem domain was the natural next step. *Object-Oriented Systems Analysis (OOSA)* by Shlaer and Mellor [SM88], is probably the first approach targeting in this direction.

It was followed by several other *Object-Oriented Analysis & Design* approaches. A list of just the most important ones may contain *OOA/OOD* by Coad and Yourdon [CY91] [CY92], *Responsibility-Driven Design (RDD)* by Wirfs-Brock et. al. [WBWW90], as well as *Object-Oriented System Analysis (OOSA)* by Embley et. al. [EKW92], *Hierarchical Object-Oriented Design (HOOD)* by Robinson [Rob92], *Object-Oriented Analysis & Design (OODA)* by Martin and Odell [MO92] as well as *Object Behaviour Analysis (OBA)* by Rubin and Goldberg [RG92]. Of course the *Object-Modeling Technique (OMT)* by Rumbaugh et. al. [RBP+91] as well as *Object-Oriented Software Engineering (OOSE)* by Jacobson et. al. [JCJv92] have to mentioned in this context as well. *OMT* was another data-oriented approach, like Booch's *OOD* was. Similar to *OOA/OOD* by Coad and Yourdon, it used class models for the analysis. However, its notation, which found broad acceptance, was different to that of all previous approaches. It was also one of the first approaches that employed state-transition diagrams to model the life cycle of instances. *OOSE* is also a rather outstanding approach, as it was the first one proposing the idea of using *Use Cases* to perform a scenario-

based analysis, rather than starting directly with a class model. The same holds for *OBA*, which similarly describes the use of scenarios in the early analysis. *HOOD*, which was developed on behalf of the European Space Agency (ESA), may be also explicitly mentioned in this context, as it introduced the notion of a top-down hierarchically decomposed design that was picked up by various later approaches, especially in the embedded & real-time domain.

It is not astonishing that Oestereich speaks of a *"blossoming of methods"* ([Oes01]), Jacobson even of *"method wars"* [Jac96], when referring to that period of late 1980's and early 1990's. However, this period of diversity was followed by a time of unification. Having already influenced each other quite intensely in the following years, leading to revised versions of their individual approaches Booch and Rumbaugh, unified their *OOD* and *OMT* approaches to the so called *Unified Method* [BR95]. Finally the *Unified Process* [JBR99], a wholistic software development approach that further integrated Jacobson's *OOSE* was developed, combining the approaches of all *three amigos*. Other unifying approaches had as well be undergone. The *Fusion* method by Coleman et. al. ([CAB$^+$93]) may be exemplarily named in this context, as well as the *Unified Modeling Approach* by Rosenberg and Scott [RS99]. Furthermore *Object-oriented Process, Environment and Notation (OPEN)* by Graham, Henderson-Sellers and Younessi [GHSY97] may be named, which started as a unifying approach of Henderson-Sellers' *Methodology for Object-Oriented Software Engineering of Systems (MOSES)* [HSE94] and Graham's *Semantic Object Modelling Approach (SOMA)* [Gra95] and resulted in a unification movement being joined by around 30 methodologists, forming the so called *OPEN Consortium*.

Even if quite a few methods already existed during that time and the unification efforts were quite strong, new approaches were yet published during the mid 1990's. *Recursive Design* by Shlaer and Mellor [SM97] may be mentioned here, as it is one of the first approaches using automated transformation of analysis into design models as well as complete code generation. *OOram* by Reenskaug et. al. [RWL96] is also worth being mentioned, having introduced the notion of role models.

In the embedded & real-time domain *Software Design Methods for Concurrent and Real-Time Systems (CODARTS)* by Gomaa [Gom93] and *Octopus* by Awad, Kuusela, and Ziegler [AKZ96] may be mentioned. While *CODARTS* can be seen as a mature *Structured Analysis and Design* approach, bringing together best practices and experiences of the elapsed decade, *Octopus* is one of the first object-oriented methods targeting the embedded & real-time domain. Other prominent approaches of that time are *Real Time Object Oriented Modeling (ROOM)* by Selic et. al. [SGW94] and *Hard Real-Time Hierarchical Object-Oriented Design (HRT-HOOD)* by Burns and Wellings [BW94]. While *HRT-HOOD* was an extension of the already established *HOOD* method to meet the needs of hard real-time systems, *ROOM* was a rather novel approach. While using hierarchically decomposed models, similar to *HOOD*, *ROOM* introduced the notion of fully encapsulated classifiers, being connected to their environment by ports. *ROOM* was probably also the first approach that defined a clear semantical cohesion between structural and behavioral models.

Two years before a common methodology was defined by the *three amigos* with their *Unified Process*, they comitted themselves to a common notation, the *Unified Modeling Language (UML)* [BJR96]. The *UML* was however not only influenced by Booch, Jacobson, and Rumbaugh, but also incorported aspects of other approaches as well. Harel for example contributed the notation of state charts [Har88], Martin and Odell that of activity diagrams, Embley the idea of composite structures and high-level view, just to name a few. A competetive approach to the *UML*, called *OPEN Modeling Language (OML)*, had been developed by the *OPEN Consortium* during the same time. This notation however has not even slightly gained the impact of the *UML*. At the latest by its adoption as an official *Object Management Group (OMG)* standard in 1997 [OMG97], the *UML* was the first and foremost modeling language. While former approaches almost always had their own notation and tooling, those approaches being published at the end of the 1990's and in the early 2000's were mostly influenced by the *UML*.

The trend towards *UML* did also pertain the embedded and real-time domain, where the *Rapid Optimizing Process for Embedded Systems (ROPES)* by Douglass [Dou99a] and *Concurrent Object Modeling and Architectural Design Method (COMET)* by Gomaa [Gom00] may be mentioned as the most significant developments. An adoption of the *ROOM* methodology to use *UML* notation was also proposed by Selic and Rumbaugh at this time as well. The respective profile extension to the *UML* ([SR98]) became known as *UML-RT* and was the notation being supported by the *Rational Rose RealTime* tool, which was an improvement of the *ObjectTime* tool supporting *ROOM*. *UML-RT* may not be confused with *RT-UML*, which is the colloquial denomination of a competeting proposal by Douglass [Dou99b], which was the notation supported by the *I-Logix Rhapsody* tool, forming also the basis for the *UML Profile for Schedulability, Performance and Time*, which was first published by the Object Management Group (OMG) as a Final Adopted Specification in 2002 [OMG02].

3.1.3 A Decade of Great Diversity - CBSE, PLSE, MBSE (from the late 1990's to the early 2000's)

While the *Object-Oriented* paradigm had clearly dominated the 1990's, the decade of the 2000's (including the late 1990's) can be summarized as a decade of great diversity.

Inspired by the great success of *Object-Oriented Analysis & Design* and being driven by the goal to achieve reuse also on the level of coarse-grained architectural building blocks, adopting component-technology to existing object-oriented methodology lead to the new paradigm of *Component-Based Software Engineering (CBSE)*. *Catalysis* by D'Souza and Wills 1998 [DW98] may be named as one of the first and probably the most outstanding approach targeting this direction. It employs UML notation to specify components with their respective interfaces, as well as connectors denoting the composition of components. Other component-based contributions of that time are for example *UML components* by Cheesman and Daniels [CD00], *KobrA* by Atkinson et. al. [ABB+02], or *Select Perspective* by Apperly et. al. [AHL+03], all being related to the UML as underlying notation as well. An enhanced version of the *OPEN* approach,

called *OPEN/CBD*, was also published at that time [HS01], enriching the original approach by techniques to support the acquisition and integration of components. In the embedded and real-time domain, *Component-Based Software Engineering* did not have the same impact than in the field of industrial information systems. However, some approaches were developed targeting especially this domain. The *Koala* approach [vvKM00] by Ommering et. al. may be exemplarily named in this context, being a component-based approach for embedded systems in the consumer electronics market. The *Pervasive component systems (PECOS)* approach [MSZ01] by Müller et. al. may be mentioned in this context as well, as it was one of the few approaches especially targeting on small embedded real-time systems.

Being closely related to the *CBSE* principle of reusing components is the idea of developing software applications in terms of product families or product lines. Even if Parnas had initiated the idea of developing program families already in the mid 1970's [Par76], and some trend-setting contributions could already be found in the late 1980's and early 1990's, it was not before the early 2000's that this trend gained significant impact. Being inspired by the success story of such approaches in the hardware development, where development in terms of product lines was already common practice at this time, *Product-Line Software Engineering (PLSE)* was first and foremost applied in the closely related domain of embedded & real-time software. Quite a few *PLSE* approaches saw the light of the day during the early 2000's, out of which the most prominent ones are probably *FAST* method developed by Weiss and Lai [WL99], *PuLSE* by Bayer et. al. [BFK+99], and the *Architectural Design Method* by Bosch [Bos00], as well as the *Framework for Software Product Line Practice* by Clements and Northrop [CN02], which is - in contrast to the before mentioned - not a single methodical approach, but a collection of approved practices and recurring problem patterns. The *Product Line UML-Based Software Engineering (PLUS)* approach by Gomaa, an extended version of his *COMET* method to develop software product lines, may be named for the sake of completeness, as well as some approaches related to various European wide research projects like the *ITEA Eureka* projects *CAFÉ* or DESS. All of those did however not have very great significance.

Besides those trends of *Component-Based Software Engineering* and *Product-Line Software Engineering*, what can be observed during the early 2000's is that with the adoption of the UML as an official standard in 1997, a general trend to intensive use of models throughout the development life cycle started, mostly employing the *UML* as underlying notation. The *Model-Driven Architecture*/*Model-Driven Development (MDA/MDD)* approach by the OMG [OMG01] may be named as one of the first approaches, making intense use of models, and it is by far the most popular one, being often named as some sort of progenitor for other model-driven approaches. The basic idea behind it is to employ a *Computation Independent Model (CIM)* analysis model and a *Platform Independent Model (PIM)* design model to capture a software's business logic independent from any underlying technical implementation detail, like middle-ware or database technology. By combining the *PIM* with a model of a concrete target platform, referred to as the *Platform Model (PM)*, a *Platform Specific Model (PSM)* can be obtained, which can then be transformed into the *Platform Specific Implementation (PSI)*.

While the idea behind MDA/MDD is quite specific to the domain of industrial information systems, the general ideas of having models as the central engineering artifacts has been adopted by several succeeding approaches, leading to new paradigm of *Model-Based Software Engineering (MBSE)* or *Model-Driven Software Engineering (MDSE)* respectively. Like with all preceding paradigms, quite a few of such model-based approaches were published beginning in the early 2000's. *Executable UML* by Mellor and Balcer [MB02] may be named as one of the early approaches of that kind, as well as the *Software Factories* approach by Greenfield and Short [GSCK04] at Microsoft and the *Generative Programming* approach by Czarnecki and Eisenecker [CE00], both being based on domain-specific modeling languages. In the embedded & real-time domain, the trend to model-based approaches can as well be observed. However, this trend is rather young, so there are few approaches that have been published before the mid 2000's.

A last trend having emerged during the early 2000's is that toward flexible and lightweight development methods, originating from the ideas of Beck's *Extreme Programming* [Bec00]. This trend can as well be seen as a self-contained development paradigm, namely *Agile Software Development*, and quite a few successful approaches were published related to it. The *Dynamic Systems Development Method (DSDM)* by Stapleton et. al. [Sta97], *Feature Driven Development (FDD)* by Coad et. al. [CLL99], and *Agile Modeling* by Ambler [Amb02] may be exemplarily named in this context, which all have in common that they concentrate on the domain of classical information systems. Although few methodologists tried to apply the idea to the embedded and real-time domain as well, amongst which Hruschka and Rupp [HR02], as well as Grenning et. al. [GPB04] may be named, one may generally draw the conclusion that agile approaches are indeed mostly inapplicable to this domain. Several reasons may be cited to support this thesis, for example the strong relationship between hardware and software development and the hard time and space constraints embedded systems may face, which indeed prevents the application of most agile techniques like planning games, object-oriented refactoring, or test-driven development. The most apparent reason may however be that this topic has already pretty much left the focus of the embedded and agile communities, so that only few contributions can nowadays be found on that topic.

3.2 State of the Art - The current situation in academia and industrial research

As already indicated, component-based, product-line, and model-based software engineering are probably the predominant engineering paradigms of the current decade - at least as far as the academic research is concerned. One may just look at the multiplicity of research projects being undertaken in this direction, as well as at the number of conferences and workshops related to the field, to fortify this thesis. While model-based software engineering is a quite recently upcoming paradigm - at least if one ignores the very early approaches related to OMG's *Model-Driven Architecture/Model-Driven Design(MDA/MDD)* initiative [OMG01], which can also be traced back to the early

2000's - component-based and product-line software engineering may be regarded to already have a certain history. Nevertheless, they still seem to be exposed to increasing interest, especially in the embedded & real-time domain, where due to large technological advances in recent times, component-technology seems to be more and more applicable, and where product-line related questions are always a hot topic due to the multiplicity of product variants that often have to be faced.

However, while both, component-based and product-line software engineering, may be regarded as interesting research fields, they are not in the central focus of this thesis' work and will therefore not be investigated in detail in the following. One may refer to [ABGP05] or [vdLSR07] and [KD06] respectively to find a good survey on current research trends related to both fields. While this does of course not rule out that some of the approaches, being named in the following, may also investigate component-based related questions or cover aspects related to the engineering of product-lines, it is just that the focus will be set to model-based software engineering in the following, as it is regarded to have introduced a rather novel quality due to its central and absolute commitment on models as the primary engineering artifacts.

As already indicated in the historic outline, it was not before the mid 2000's, when the Object Management Group adopted version 2.0 of its *Unified Modeling Language* standard, that model-based software engineering has significantly gained impact. A multiplicity of approaches has however been published since then, and this trend is yet steadily increasing. While a number of those approaches may be regarded as being generally applicable, various approaches can also be found, having their origin in a certain application area, thus accommodating the special characteristics of that respective area.

There is a lot of interference between general purpose and application area specific approaches, in terms of chronological adjacence as well as regarding contents - often general purpose approaches have been customized by application area-specific ones, or general purpose approaches have been generalized from practices and experiences being applied in certain application areas. A clear separation between those approaches being applicable to several and those being restricted to a certain application area is therefore hard to achieve. However, to draw a more systematic picture of the state-of-the-art situation, such a separation is tried in the following, pointing out interferences between differently classified approaches where they are immanent. It should be clear that below those explicitly mentioned interrelationships, influences between the different presented approaches are natural and likely, not least because of the close chronological adjacency of their inceptions.

3.2.1 General-purpose Approaches

Even if OMG's *Model Driven Architecture/Model-Driven Design (MDA/MDD)* initiative may be regarded as a rather special representative within the broad set of model-based engineering approaches because of its strong focus on platform abstraction aspects (cf. Section 3.1.3), it has been generally accepted as the prequel of such ap-

proaches. Indeed, *MDA/MDD* may be regarded as one of the first approaches employing models as the central working artifacts throughout the overall software life-cycle. However, while having great indirect influence in the field, a direct impact of OMG's initiative to the embedded & real-time domain could - at first - not be observed. This is probably caused by the fact that platform abstraction is difficult in a domain where resource constraints are very restrictive and where a lot of interfaces to the underlying hardware are existing. It may also be caused by the fact that OMG's initiative was at first understood to be very much related to its object-oriented middle-ware approach *Common Object Request Broker Architecture Revision 2 (CORBA-2)* [OMG95], which is hard to apply in the embedded & real-time domain.

Having thus laid the early basis for this new engineering paradigm in the beginning of the 2000's, it was probably not before the adoption of OMG's *Unified Modeling Language* standard version 2.0 [OMG05c] in 2005, that model-based software engineering has significantly gained impact in the embedded & real-time domain. This may be explained by the fact that the new standard version of the *Unified Modeling Language* adopted many embedded & real-time concepts, such making the language more applicable to this domain as well. The newly introduced composite structure diagrams may be named in this context, which have been adapted from *ROOM*, as well as a major revision of the already contained sequence diagrams to gain an expressive power comparable to that of *Message Sequence Charts (MSC)* [ITU04] (compare Section 3.2.2 on the situation in the telecommunications application area). Further, a newly defined action semantics to increase the expressive power of behavior diagrams, as well as new concepts especially related to timing aspects, incorporated into the newly introduced timing diagrams, found its way into the language standard.

In the following years quite a few model-based methodological approaches have been published, most of which had their origin in a respective application area (they will be dealt with in the succeeding sections). Nevertheless, a few approaches may also be named that can be regarded as being general purpose, for example I-Logix' *HARMONY* [Dou07], a model-driven successor to the quite popular *ROPES* approach, or *Accord/UML* [TGRT06], an approach demonstrating the applicability of MDA concepts to embedded & real-time systems, being published in the context of the French Commissariat à l'Énergie Atomique (CEA).

At the same time the Object Management Group started to spent more and more efforts on making its approaches better applicable to the embedded & real-time domain. While the current 2.1.2 version of the *Unified Modeling Language* [OMG07d] is intended to be widely applicable and does therefore not subsume additional embedded & real-time specific concepts, great efforts are currently being spent in the definition of a new dedicated add-on for the modeling and analysis of embedded & real-time systems, the so called *Modeling Analysis of Real-time and Embedded systems (MARTE)* profile. Having been published in a first beta version in mid 2007 [OMG07a], *MARTE* is intended to be a replacement of the current *UML Profile for Schedulability, Performance and Time* [OMG05b]. It combines experiences gained with the latter profile and also tries to integrate domain specific approaches like *AADL* or *AUTOSAR* (compare Section 3.2.2 on domain-specific approaches within the automation and aerospace &

defense application domains for details) that have emerged in the meantime. Efforts being spent by the OMG to transfer CORBA to the embedded & real-time domain (CORBA/e) [OMG06a] may be also named in this context, as it is intended to increase the applicability of MDA/MDD approaches in the respective domain.

While one trend of the current decade can thus be identified in terms of model-based engineering becoming the predominant software engineering paradigm (at least from a state-of-the-art perspective), a second trend can as well be observed, namely the integration of related but distinct hardware and software engineering disciplines into a common, integrated one. Indeed, *Systems Engineering*, as it is referred to, has a long tradition. First ideas have been published in the late 1950's and early 1960's [GM57][Hal62][Che67], and *HW/SW co-design*, which is closely related, has been a major topic since the early 1990's. *Object Oriented Systems Engineering Method (OOSEM)*, developed by Friedenthal et. al. [LFM00], or the *Rational Unified Process for Systems Engineering (RUP-SE)* [Can01] may be named as more recent approaches. With the adoption of the current UML standard this trend has then significantly amplified through the definition of a respective *UML Profile for Systems Engineering (SysML)* [OMG07b]. Approaches like *HARMONY-SE* [Hof06] or *SYSMOD* [Wei06] may be quoted as evidence for this.

3.2.2 Domain-specific Approaches

While there are a couple of generally applicable ones, the preponderant majority of current engineering approaches - especially those related to model-based software engineering - may be traced back to a specific application area. Here, aerospace & defense, automotive, as well as telecommunications may be regarded as the key application areas, serving as some sort of catalyst due to their specific regulatory, organizational and economical constraints.

That is, within all three domains, strong regulatory constraints related to reliability and safety can be observed. A distributed development environment in terms of a multiplicity of suppliers and a smaller set of manufacturers is also characteristic. Both contributes to the fact that unifying standards are agreed on. However, the predominant reason probably is that within all three key application areas, a handful of market dominating manufacturers can be found, which have the economic strength to facilitate new approaches. Most domain-specific model-based engineering approaches, having emerged in recent time, thus can - probably due to this exceptional position - be traced back to the aerospace & defense, automotive, or telecommunications application area, as will be pointed out in the succeeding sections.

Aerospace & Defense Aerospace & defense can probably be seen as one of the precursor application areas in terms of model-based software engineering. Already in the mid 1990's first efforts to use graphical specification languages and code generation could be observed. As a result of *DARPA*'s *Domain Specific Software Architecture (DSSA)* program, which was started in 1991 [MG92], early specification languages

like *MetaH* to specify software architectures or *ControlH*, being used for guidance, navigation, and control algorithms were developed, as well as sound modeling tools to support those languages.

A unified modeling standard to specify software architectures and to leverage model-based software engineering was reached with the *Architecture Analysis and Design Language (AADL)* language, being defined by the Aerospace Avionic Systems Division of the Society of Automotive Engineers (SAE) in 2004. Having started as a mere textual specification language, inspired pretty much by the above mentioned *MetaH* language, AADL was subsequently enhanced by graphical notation capabilities and in 2006 also with a concise underlying meta-model. Since 2007 a UML profile for AADL is as well available.

Closely related to AADL is the *COTRE Architecture Description Language*, which was developed by the European *COmposant Temps RÉel (COTRE)* project [FG+04], executed from 2002 until 2004. In contrast to AADL, the COTRE language was designed to not only support specification but also formal verification of systems, thus going beyond the scope of AADL. However, AADL and COTRE also had a lot of interferences, leading finally to the adoption of some of COTRE's core modeling concepts (high level composition and dynamic description of component behavior using an automata language) into the AADL language standard, incorporated into the so called *Behavioural Annex of the AADL* [Sub07].

A multiplicity of non-commercial and commercial tools have been published to support AADL/COTRE, for example SEI's *Open Source AADL Tool Environment (OSATE)*, the *Toolkit In OPen source for Critical Applications & SystEms Development (TOPCASED)* by the CNRT Aeronautic & Space partners, or *Ocarina* by Télécom Paris, all supporting the graphical modeling with AADL as well as the evaluation of AADL specified architectures. Some, as the *ADeS* tool by Axlog Ingénierie, also offer simulation of such architectures.

While there are thus some commonalities with respect to notation and tools, only few common approaches in terms of software engineering methods can be identified. Both, *AADL* as well as *COTRE* are closely related to the notation provided by the *HOOD* respectively *HRT-HOOD* method. Indeed, a lot of modeling concepts incorporated into those languages originate from the HOOD notation, others can be directly mapped to HOOD concepts [Dis04]. Thus, in terms of software engineering methods, HOOD and HRT-HOOD, which have undergone several updates from their first publications in 1992 and 1994 respectively, still have to be regarded as state-of-the-art, even if *AADL* is employed as underlying notation. Apart from them, only few engineering methods can be found that are specific to the aerospace & defense application area, and those that can be found are usually proprietary to a specific company. The *Model-Centric Software Development (MCSD)* [WL06] of Lockheed Martin may be exemplarily named in this context, which is one of the few publicly proclaimed company specific model-based engineering solutions.

Other approaches are rather domain unspecific, often not limited to the engineering of software but indeed addressing overall systems engineering. The aforementioned rather general *OOSEM* and *Harmony-SE* method approaches indeed have their origin in the aerospace & defense application area. The *Department of Defense Architecture Framework (DoDAF)* and the *UK Ministry of Defense Architecture Framework (MODAF)* can be also named in this context. Even if a complete software engineering method is not delivered by them, at least a compilation of guidelines and instructions is provided. The so called *DoDAF deskbook* [Gro03] for example prescribes detailed tasks that have to be performed to create DoDAF models and defines, which deliverables have to be produced.

Automotive In the automotive application area, the situation faced is quite comparable to that of the aerospace & defense area. That is, model-based software engineering of control and feedback control algorithms could be regarded as state-of-the-art in automotive software engineering already in the early 1990's (cf. [Sch04]). Tools like MathWorks' MATLAB Simulink/Stateflow or ETAS' ASCET-SD were used from then on to specify and also simulate control and feedback control engineering models, code generation from those models was already supported. However, those approaches were restricted to the engineering of single car functions.

As a consequence, strong efforts to achieve software engineering solutions to face the integration of different functions and subsystems in terms of an overall software architecture, can thus be observed from the early 2000's. The ITEA *Electronics Architecture and Software Technology - Embedded Electronic Architecture (EAST-EEA)* research project [TEF+03] may be named in this context, which was executed from 2001 to 2004, bringing together major European car manufacturers, as well as suppliers, tool editors and research institutes. Besides the definition of a common middleware platform to abstract from the rather special hardware architecture in terms of distributed and interconnected electronic control units (ECUs), its most important work result is probably the *EAST Architecture Design Language (EAST-ADL)*. EAST-ADL was specified as a UML profile to allow modeling support by standard UML tools. It may be regarded as the first unified notation for software architectures in the automotive application area, supporting modeling on different abstraction levels.

Further, *AutoMoDe*, a research project being executed from 2003 to 2006 at the TU Munich in close cooperation with the German car manufacturer BMW, its supplier Robert Bosch GmbH and the tool vendor ETAS [BBR+05], may be mentioned. Its goal was to provide concepts to better integrate between different abstraction levels and views of an architecture model, something that was identified as a weakness in the EAST-ADL. Its major working outcome was the homonymously named *AutoMoDe* notation allowing to better integrate models of different abstraction levels, and a prototypical tools support by the respectively extended AutoFocus2 tool.

Besides those efforts of the research community, the automotive industry itself pressed ahead by itself in terms of the AUTOSAR (AUTomotive Open System ARchitecture) initiative. That is, in 2003 a consortium was jointly founded by major international car

manufacturers and suppliers to define a standard architecture for automotive software applications. As a result, the first AUTOSAR specification was published in 2005, a current version of the standard, AUTOSAR 2.1, is available since 2007. The initiative adopted various concepts defined by the EAST-EEA project, in particular the specification of an independent middle-ware, which enables the integration of independently developed AUTOSAR software components into an overall software system. The development of AUTOSAR compliant software components is supported by nearly all domain-specific tools, for example ETAS' *ASCET* or dSpace's *SystemDesk*. It has to be pointed out that AUTOSAR does only specify a common software architecture in terms of a common runtime environment and defined hardware abstraction and software communication interfaces and no common notation.

To bring together those efforts, the European Commission founded the *Advancing Traffic Efficiency and Safety through Software Technology (ATTEST)* project in 2006 [ATE07]. Its goal is to define a new automotive architecture description language, based on the experiences with EAST ADL, being furthermore aligned with AUTOSAR and UML/SysML. Special research is currently also going on to deal with integration aspects related to timing and concurrency issues in the context of AUTOSAR. This is investigated by the ITEA *Timing Model (TIMMO)* project [J+07].

However, while a lot of efforts can be observed related to model-based software engineering in terms of notations/languages and tools, similar to the aerospace & defense application area, common software engineering methods seem to be rather rare.

Telecommunications The situation faced in the application area of telecommunications is a twofold one. Mobile devices like cellular phones or PDA's have undergone a fast-paced development. While resource limitations in terms of memory and computation time still require special awareness in the engineering of mobile application software, they do not seem to be the most challenging constraints any more. Instead, aspects relevant to classical industrial information systems like modifiability, run-time extensibility and rapid application development [Mik07] seem to be the most significant drivers.

While the development of software for such devices is still challenging, it is therefore probably more comparable to that of classical information systems rather than that of other embedded & real-time systems (and is therefore slightly out of the scope of this thesis' work). Indeed, system software for such devices can rely on capable operating systems and implementation frameworks; often parts of the required functionality can even be integrated as a complete *System on a Chip (SoC)*. Development of application software can apply modern object-oriented technology like Java or .NET and profound class bibliographies and API's are available.

The situation faced in the engineering of network infrastructure related devices like telecommunication switches or routers seems to be a bit different. Here, indeed most of the embedded & real-time characteristics laid out in Section 2.1 still perfectly hold. Traditionally, those software has been specified using languages like the *Abstract Syntax Notation 1 (ASN.1)* [ITU02], the *Specification and Description Language (SDL)* [ITU99], or *Message Sequence Charts (MSC)* [ITU04]. Those languages have all being standardized by the International Telecommunication Union (ITU) already during the late 1970's and early 1980's, and soon after the adoption of the initial Unified Modeling Language standard in 1997, have been mapped to it in the form of UML profiles.

Having undergone their latest revisions in the late 1990's and early 2000's, those ITU-related languages are nowadays mostly subsumed by the current version of the Unified Modeling Language. This is reflected - not least - by the fact that specific tool support for the respective languages has been replaced with according UML support (for instance compare Telelogic's *Tau Generation 1* versus *Tau Generation 2*). While the Unified Modeling Language thus seems to be accepted as the current state-of-the-art modeling language (at least with respective profile support), the applicability of OMG's MDA approach in the respective application area is a current research topic. The European funded *MOdelDrivenArchitecture in TELecomunnications (MODA-TEL)* [Gav02] executed from 2002 to 2004 may be named in this context.

Miscellaneous Different to aerospace & defense, automotive, and telecommunications, the state-of-the-art situation in other embedded & real-time application areas like industrial automation, consumer electronics, or healthcare seems to be hard to sketch, as - due to different organizational and economical constraints - joint domain-specific research efforts seem to be quite seldom. While some major companies of the respective application areas are also involved in the several research projects attributed to one of the aforementioned key application areas, most domain-specific research efforts in the non-key application areas seem to be proprietary, that is specific to a certain individual company.

There are several reasons for this. First, in contrast to the aforementioned aerospace & defense, automotive, and telecommunications key application areas, the software mostly seems to be of a lower complexity. This allows a centralized, non-distributed development, where one developing organization is in overall control of the development cycle. This is very much in contrast for example to the automotive application area, where the car manufacturers mostly take the role of integrators and the development of software components is realized by suppliers, forcing a distributed engineering of software. Further, as opposed to the aforementioned key domains, the respective markets seems to be not as narrow, showing a greater number of market participants. Also regulatory constraints seem to be less demanding, if one for example compares the situation of consumer electronics to that in the aerospace & defense industry. Last, the trend to standardization seems to be less traditional in those domains, if for example compared to the situation faced in the telecommunications sector, where major joint standards had already been defined in the late 1960's. All those aspects con-

tribute their portion to the effect that a trend to joint research or even to unification and standardization efforts seems to be less distinct in the respective domains.

However, a general trend towards model-based software engineering can also be observed in those application areas. This may be fortified by the fact that - even if the scope and impact of such approaches is limited and the overall number of publications is not as distinct as in the aforementioned key application areas - research experiences related to model-based software engineering approaches are publicly shared at respective conferences or workshops. Also product-line engineering can be regarded as a central point of interest, in particular in application areas like the consumer electronics, where a large number of product variants has to be handled.

3.3 State of the Practice - The current situation in industry

While the state-of-the-art situation reflects that model-based software engineering - besides component-based and product-line engineering - is probably the dominant engineering paradigm of the decade, the current situation that one can face in the industrial practice does - of course - often provide a different picture.

The European *Software Engineering Methodologies for Embedded Systems (MOOSE)* research project [vS02], executed from 2002 to 2004, conducted a survey amongst 19 Finnish companies (Hirvi survey) and 30 European companies (acquired by a web repository) and gathered empirical data from over 100 development projects related to embedded & real-time software (cf. [vS04][TK04]).

Interestingly, the survey showed that indeed 16% of the Hirvi survey projects and even about 40% of the web repository projects were developed without using any design respectively construction method. While 16% (Hirvi survey) to 17% (web repository) of the projects claimed to use *Structured Analysis & Design*, and *Object-Oriented Analysis & Design* based on Rumbaugh notation was used in 13% of the web repository projects (in this case no data is available from the Hirvi survey), 37% (Hirvi) respectively 23% (web repository) of the projects used a design method based on the UML as notation, not providing however any detail about the underlying engineering paradigm.

The interesting question would be how many of those approaches using the UML indeed can be regarded as being model-based or even model-driven. While truly scientific data is not available to investigate this (the MOOSE project did not cover this), an online survey conducted by Ganssle in 2006 amongst 659 respondents of his newsletter *The Embedded Muse* [Gan06] might give an interesting answer: accordingly, only about 5% of the respondents stated to even perform automatic code generation based on UML models[2]. This enforces the impression that model-based software engineering seems to not have found its way into the majority of industrial projects yet. The impression may additionally be fortified by the fact that according to the MOOSE sur-

[2] According to the survey, 4% perform automatic code generation based on UML class diagrams, 3% from UML sequence diagrams, and 7% from UML state machine diagrams

vey, over 50% of the overall projects (Hirvi and web repository) were developed using either no design tools or a general drawing tool.

Another interesting result of the MOOSE project is that the main implementation language used in most development projects related to embedded & real-time projects is still C or even Assembler. The amount of projects using Assembler (in combination with C/C++) as coding language is stated to be above 20%, plain C is used for implementation in more than 30% of all projects, and in combination with C++ in more than 50% of the projects. This seems to be a representative result. Indeed, online surveys performed by ESL Now! [ESL05] or the Electronic Engineering Times [Rom06] performed in 2005 and 2006 yield similar results on the yet outstanding importance of the C and Assembler languages.

In case of those projects using a combination of C and C++ it would of course be interesting to know, to what percentage those projects indeed used the object-oriented capabilities of the C++ language. While the study conducted by the MOOSE project does not give any hints on this, the already quoted online survey by Ganssle might help to come to a concise interpretation: out of the 659 respondents of his survey, 63.7% stated to use no object-oriented implementation language at all and 34.7% answered to use an object-oriented implementation, but only for purposes of encapsulation, not making use of object-oriented features like inheritance or polymorphism. 22.2% of the respondents answered to use an object-oriented language, making use of its inheritance features but not relying on polymorphism. Only 14% of the respondents stated to use an object-oriented language to its overall extend in terms of encapsulation, inheritance, and polymorphism [Gan06].

While this empirical data might help to gather a first impression of the current state-of-the-practice, its representativity may be questioned, due to its limited scope. A concise characterization of the current industrial situation could of course be better achieved by examining the situation in the different application areas in detail, as it was done in case of the state-of-the-art situation. Concerning the situation in the industrial practice however, this is a considerably difficult venture, as empirical data is outstandingly rare.

What can be stated is that in the aerospace & defense application area, still most industrial development projects seem to employ long-term approved methodological approaches. The development efforts related to the Eurofighter fighter plane or the Airbus A-380 passenger plane may be named exemplarily in this context, where the application of *HOOD* and *HRT-HOOD* methods and related *STOOD* and *CP-HOOD* tools is documented, reflecting a state-of-the-art situation of the early 1990's. However, quite a number of industrial pilot projects related to model-based software engineering using AADL and UML respectively SysML are documented as well, so it is likely that there will be ongoing effort in this direction (cf. [Fei07]). The large extend to which aerospace & defense companies are involved in current research efforts related to model-based engineering also confirms this prognosis.

In the automotive application area, model-based engineering of single control and feedback-control related car functions, based on mathematical models, as supported by

tools like Mathwork's *Matlab*, *Simulink* and *Stateflow* or National Instruments' *MATRIXx*, has quite a long tradition, reaching back to early pioneers in the early 1990's and may be regarded as being widely used with the end of the 1990's. Today, model-driven development of single car functions may thus be regarded as being state-of-the-practice. Model-based engineering of the overall vehicle software in terms of a proper integration of different software components (on different ECUs) to an overall software system is not yet state-of-the-practice and is therefore in the very central interest of the automotive industry. One may just look at the strong efforts related to the aforementioned AUTOSAR initiative to confirm this. Here, according to [FBH+06], first car models, having parts of its functionality realized conformant to AUTOSAR, will go into series production in 2008, and all core partners of the AUTOSAR consortium will have implemented AUTOSAR conformant software in their vehicle generations by 2010.

While the general impression that arises from the surveys and polls quoted above sketches a rather dark picture regarding the state-of-the-art situation in the embedded & real-time domain, these two examples show that - at least in some application areas - first ground regarding model-based engineering of software has already been gained. A widespread adoption of model-based engineering in the embedded & real-time domain, or in even a single of its key application areas, is however far from being realized.

3.4 Observations & Conclusions

Reflecting the historic course of software engineering approaches for the embedded & real-time domain it can be observed that this domain has more and more moved into the focus of academic interest. That is, while the overall number of approaches targeting classical information systems may still outweigh the number of embedded & real-time approaches, recent history shows that the domain is significantly gaining impact. A very strong indicator for this may be that several of the recently introduced *product-line*, *component-based*, and *model-based* engineering approaches either explicitly concentrate on the domain of embedded & real-time systems, or at least provide a demonstrating case study being related to the respective domain.

While this may be an indication for the significant increase in its overall (economic) importance, it can as well be taken as an indication that the adversity here is a rather great one. The current state-of-the-practice situation, as it was sketched in the preceding section, may support this thesis. Indeed, the embedded & real-time domain seems to face a number of difficulties. First, strong technical restrictions are typical to most embedded & real-time systems, having strong implications on the software that is running on those systems in terms of dependability, efficiency, or adequacy and thus making the engineering of those software challenging. Second, the organizational, and economical constraints in the embedded & real-time domain are somewhat outstanding compared for example to the domain of classical industrial information systems. In fact, software engineering has - historically seen - played a quite subordinate role

in the engineering of most embedded & real-time systems, which were in the majority originally build up as mere mechatronic devices, having either no or only few parts realized in software.

With the trend to build-in more and more functionality into embedded & real-time devices, and with an ever increasing cost pressure, the amount of software in such devices is steadily increased, leading also to a greater complexity. This increased complexity together with the strong aforementioned technical restrictions that are characteristic for embedded & real-time software are the central challenges, the embedded & real-time domain has to face. As the historic survey unveils, abstraction is the principle to face complexity. Having first begun to transfer programming language constructs and principles to the late detailed design, the application of models - offering an increased level of abstraction - has advanced more and more from the solution space into the problem space. That is, soon after those early approaches using models within detailed design only, further approaches entered the field, applying modeling also to the analysis phase to gain a concise understanding of the problem space.

The central commitment to models as the primary engineering artifacts throughout the overall software life-cycle, as it is proposed by model-based software engineering, thus seems to be an adequate means to face the increased complexity. The large number of model-based approaches, being published related to model-based software engineering in the embedded & real-time domain, and strong related efforts spent at least in some key industrial application areas may be taken as a good indication for this.

However, as the characterization of the industrial state-of-the-practice situation reveals, a broad adoption of current state-of-the-art approaches, in a sense that they are commonly applicable to the overall embedded & real-time domain, does not seem to be the case. Indeed, observing those strong efforts in the aerospace & defense, automotive, and telecommunications industry on the one side, and evaluating the results of current surveys on the other side, leads to the impression of a rather strong diversification within the embedded & real-time domain. While certain key application areas are far ahead - being in the central focus of current research efforts - other application areas like consumer electronics, industrial automation, healthcare and others seem to be way behind, either because of different technical restrictions, but probably also because of different organizational and economical constraints.

Chapter 4

Definition of Scope - Problems, Challenges & Goals

As the comparison of the current situation in the academic research and in the industrial practice clearly unveils, there is quite a broad gap between them. Even if this does not hold for the embedded & real-time domain in particular, but can be observed for other domains as well, here the gap is a rather great one. This seems to hold especially for those non key application areas besides aerospace & defense, automotive, or telecommunications, where the adoption of current state-of-the-art results seems to be hindered by technical, economical, or organizational constraints. It is thus the goal of the herein presented approach to deliver a model-based methodology, which especially targets small embedded & real-time systems in the respective marginal application areas, where software development is - due to the very special constraints - far behind that of others.

It has to be clear that even if limiting the systems, being targeted by the approach, to small embedded & real-time systems, and even if limiting the application areas, being targeted, to those marginal application areas as industrial automation, medicine, or consumer electronics, the domain scope would be too broad as that a single methodological approach could reasonably address it. The domain scope therefore has to be limited. This will be done in the following section. Based on a thoroughly defined domain scope, concrete problems and challenges will then be investigated within the defined scope, so that concrete goals for the herein presented approach can be derived subsequently.

4.1 Defining the Domain Scope

While searching for those embedded & real-time devices, where the adoption of current state-of-the-art approaches seems to be difficult, and where the gap between state-of-the-art and state-of-the-practice thus seems to be rather great, one inevitable crosses

the domain of *small embedded & real-time systems*. While from a general viewpoint those devices may be characterized as rather incomplex, their development is nevertheless rather challenging, as usually strong technical constraints have to be faced. Additionally, the rather marginal application areas, they can be found in, have their own distinct organizational and economical constraints, as already pointed out.

While this characterization of *small embedded & real-time systems* gives a first impression, it leaves a lot of fuzziness about which systems do indeed fall into the respective scope of the herein presented approach, and which do not. To achieve this, a more detailed characterization of the domain scope is of course necessary. However, an exact qualitative and quantitative definition of such a scope evidently seems to be quite difficult, as the applicability of an approach cannot be judged on a yes-or-now scale.

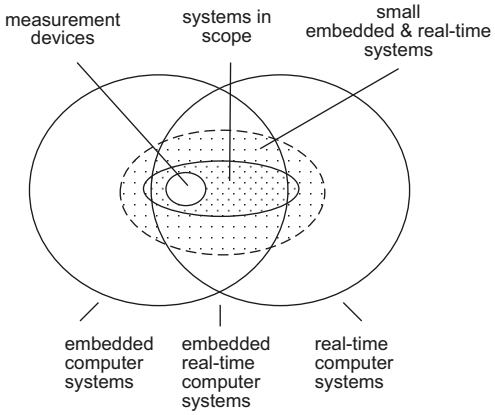


Figure 4.1: Domain Scope Definition

While the approach presented herein may not be regarded as domain or even device specific, its scope may thus - as indicated by Figure 4.1 - be probably best defined by naming a prominent representative, which is covered and shows some typical characteristics. Measurement devices, as they can be found in the industrial automation application area, will be used for this purpose[1]. The somehow broadened scope of the approach may then be easily inferred by comparing the characteristics of a potential target system to that of the representative.

4.1.1 Measurement Devices in Industrial Process Instrumentation

Within the industrial automation application area, measurement devices are counted among that group of devices, which is referred to as *field devices* because - in the context of a larger industrial automation plant - they are employed *in the field*, that is in direct contact to the physical processes being controlled, in contrast to those devices being employed in the process control level. Measurement devices may be classified

[1] Indeed measurement devices are sort of devices, the herein presented approach was initially developed for and where it has been initially evaluated with.

according to the physical process value they measure, which is mainly temperature, pressure, or flow. They occur in a multiplicity of product variants related to the different measurement principles, which are applicable, as well as to the different communication bus systems, being used in the process automation industry. Variants range from low cost mass products, as they are for example used in temperature measurement of nonhazardous liquids, to very upscaled marginal products, as for example used in the flow measurement of explosive gases.

In general, those devices may be characterized by hard resource constraints in terms of memory consumption, power consumption, and computation time, being imposed on them by the industrial environments, they are used in. Strong constraints regarding reliability are always natural for such industrial devices as well. The presence of possibly harsh environments additionally enforces that certain measurement devices, being specially developed for such purposes, also fulfill respective safety requirements, as prescribed by related industry standards ([IEC98], [MIS94]).

From a hardware viewpoint, measurement devices may be characterized as single or simple multi-processor systems (a single main micro-controller and usually not more than 2 peripheral micro-controllers), mostly based on 16-bit, sometimes also on small 32-bit embedded micro-controllers, being usually equipped with physical memory of about 32-512 KByte ROM and 0.5-512 KByte RAM. As already stated, measurement devices have to offer interfaces to different industrial communication bus systems to output the measured process values, and to be configurable and controllable by the process control level.

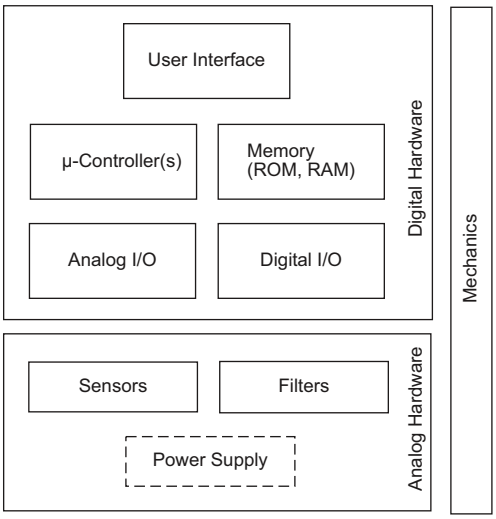


Figure 4.2: Typical Mechanics/Hardware Architecture of a Measurement Device

The typical mechanics/hardware architecture of a measurement device may be characterized as denoted by Figure 4.2. That is, on the analog hardware level they are equipped with sensors and related filters, and - dependent on whether the communication bus systems supplies the device with electrical power or not - with an own

power supply. The digital hardware consists of analog i/o components interfacing to the analog hardware, i.e. analog-digital converters used to collect data samples from the analog sensors, as well as digital-analog converters, used for example to output the measured process values on analog communication busses (this may however also be realized by the main or a peripheral microcontroller). Digital i/o components can also be found to realize communication with digital input and output components. Measurement devices are further equipped with a user interface in terms of a simple display and a keypad to display the currently measured process values and to allow a configuration and control of the device by a local operator.

The software running on measurement devices may in general be characterized as having a rather low complexity. That is, its run-time structure is rather static, in a sense that the overall software structure is initialized at startup and does not dynamically reconfigure during run-time. Even in case of a distributed architecture in terms of a multi-processor system, the software may as well not regarded to be intensely complex, as each peripheral microcontroller unit normally has a separate serial communication interface to the main microcontroller unit, and communication is usually initiated solely from there. Further, all hard real-time related functionality can be unambiguously attributed to the single main measurement task, the device has to fulfill, which is performed in terms of gathering analogue data samples via the device's sensors, processing the gathered sample data in order to calculate measurement values (signal processing), as well as output of the resulting measurement values on communication busses and on the local display. All other functionality offered by a measurement device, which is related to configuration, diagnosis, or maintenance, is usually non-real-time critical.

Further, the software often has to take direct control of the underlying physical hardware, as still most of such measurement devices is not equipped with an embedded real-time operating system. However, at least in case of multi-processor devices, an embedded real-time operating system is nowadays usually used on the main microcontroller unit, and a recent trend towards using an embedded operating system also in the simpler devices is observable. Not only because of the high hardware contiguousness that results from the absense of an operating system, but also because of the very stringent resource consumption constraints, the implementation language used to realize the software is still plain C - in close correspondence to the survey results quoted in Section 3.3.

From an economical and organizational viewpoint, the domain of measurement devices may be regarded as a rather harsh one as well. There is an inherently existing cost pressure that accompanies the development of measurement device software. In those cases where measurement devices are produced in a low overall number of pieces, the overall development costs have a relatively high impact on the single per-product costs, so software development is directly affected by this cost pressure, while the per piece hardware costs are of course also crucial. In case of a large number of produced pieces, the per-product hardware costs are the most significant ones, so that preferably cheap hardware is employed, what then indirectly has an effect on software development as well, as stronger technical restrictions are faced. In both cases hardware development

seems to have precedence, as it is the important cost driver. Consequently the development of the system's hardware is traditionally done in advance or at least with a slight jut towards that of software.

From a regulatory viewpoint, safety standards like IEC 61508 [IEC98] or recommendations like the MiSRA reports [MIS94] have to be named, which are applicable to those devices, being used in hazardeous environments. While those standards do not specify how software development has to be performed in detail, they restrict it by prescribing the application of certain safety related engineering techniques to assure that safety related aspects are covered. Besides, domain-wide software development standards can not be found, and efforts targeting this direction can further not yet be identified. This may be explained by the fact that the organizational constraints are quite different here compared to those key application areas, where a handful of market dominant manufacturers can agree on unified development standards, and further have the market power to dictate them.

A last thing that has to be pointed out is that, as in a considerably amount of other embedded & real-time application areas as well, software development is mostly not performed by software engineers, but by electrical engineers, communication engineers, process engineers, or physicians. While this should not be understood as a discrimation of those professions, it is merely mentioned to point out that due to the minor role software engineering has traditionally played in the embedded & real-time domain, software is developed mostly by domain experts, which do not have profound software engineering skills, as they are not extensively educated in the respective field.

A Typical Measurement Device - The Electromagnetic Flow Meter

To conclude the characterization of the preceding section, it may be reasonable to characterize a typical example device. The example considered here will be an electromagnetic flow meter, which is a measurement device used to measure the flow rate of a (electrically conducting) liquid, floating through a pipe[2].

The physical measurement principle, an electromagnetic flow meter is based upon, is the *Faraday law of electromagnetic inductance*. It states that an electric conductor, being moved through a magnetic field, induces a voltage orthogonal to the direction of the magnetic field and the direction of its movement. The electromagnetic flowmeter makes use of this law, by creating an electromagnetic field through two magnetic coils, which are diametrically fitted on the pipe, through which the to be measured liquid is floating. If the liquid is conducting, it will induce a voltage orthogonal to the electromagnetic field, as denoted by Figure 4.3, which can then be measured by electrodes, which are placed orthogonal to the magnetic coils that induce the field. From the measured induced voltage, the so called *raw flow velocity*, the *flow velocity (in m/s)* and the *flow rate (in l/s))* (both referred to as *process values* in the following) of the liquid, proportional to the latter, can be computed.

[2] The example device was already introduced in [NL07a] and [NL08] as a running example. It will serve the same purpose within this thesis report, accordingly.

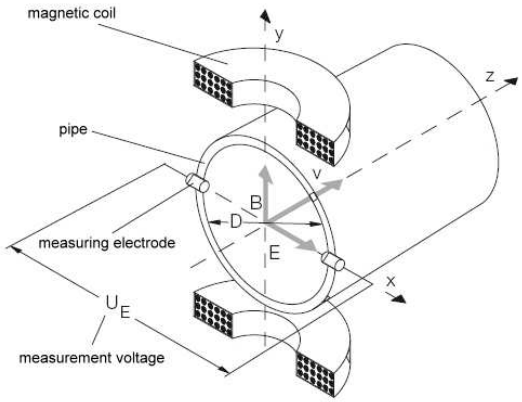


Figure 4.3: Measurement Principle of an Electromagnetic Flow Meter (cf. [GHH+04])

Conformant to the typical hardware architecture of a measurement device, outlined before, the electromagnetic flow meter example device is assumed to the equipped with a human machine interface (HMI), manifested in terms of a small LCD display and a keypad, used to report the process values to a local operator and to allow local configuration of the device. Furthermore, it is assumed to come with two digital and an analog (current) output. Its hardware is further split into two parts, namely a sensor board, which controls the magnetic coils and collects raw data from the measurement electrodes, and a main board, which is responsible of controlling the sensor board, as well as the HMI and the analog and digital outputs[3].

From a software viewpoint the electromagnetic flow meter device may be regarded to contain two distinct embedded software systems, namely the one on the sensor and the one on the main board, both being interconnected via a defined communication protocol. The software running on the sensor board is responsible of interfacing to the sensor hardware, i.e. it drives the magnetic coil by a pulse-width-modulation (PWM) and gathers the sensor raw data via an analog-digital-converter (ADC). It is further responsible of preprocessing the collected sensor raw data, i.e. it performs a sensor related calibration, as well as en error detection and correction. The main board software is responsible of performing the signal processing, i.e. it calibrates and filters the preprocessed raw values, calculates the resulting process values, and outputs those values on the display and the outputs. It is furthermore responsible of handling the interaction with a local operator via the HMI.

[3]The example system presented in [NL08] is assumed to have a dedicated output board as well, used to control the analog (current) output. It may however also be assumed, that the current output is driven by means of a pulse-width-modulation (PWM) through the main board micro-controller itself.

4.2 Problems & Challenges

As pointed out before, model-based software development still faces a lot of problems in the domain of small embedded & real-time systems, especially in those non-key application areas apart from aerospace & defense, automotive, and telecommunications. In order to provide an approach that is applicable within these application areas, the reasons for this inapplicability have to be investigated first, so that distinct goals can be formulated. This will be done subsequently.

4.2.1 Constraint-Inadequate Model-Based Software Construction

As stated before, model-based software engineering seems to be a reasonable means to deal with the increased complexity that has to be faced. It further seems to be promising in order to face the strong technical restrictions that especially small embedded & real-time systems like measurement devices have to face. This is not least due to the increased analysis potential that is gained through the intense use of models. By reasoning on models and by simulating or even executing them, consequences of decisions or the adequacy of possible solutions may be evaluated and assessed very early, even before they are actually determined or realized. It is also because model-based engineering - if applied systematically - empowers enhanced traceability, so that design decisions can thus be tracked. This does of course hold for model-driven engineering in particular, where models are interrelated much closer due to the enhanced formality, and the thereby enabled higher degree of automation.

The interesting question that comes up in this context is, why promising model-based or even model-driven software engineering approaches, which have been adopted in other parts of the domain, are not properly applicable to small embedded & real-time systems, as they can be found in the marginal application areas. The most significant reason for this seems to be that most of current model-based engineering approaches seem to be somehow contrasting to the very strong technical constraints that are being faced in terms of resource consumption and timing constraints. Even if model-based software engineering does not necessarily lead to larger and less efficient programs, the problem is that most model-based approaches - even if nominally targeting the embedded & real-time domain in particular - also compel the use of higher abstraction implementation technology like object-oriented programming languages or even component-based middleware. This probably also explains why OMG's MDA/MDD approach did not significantly gain ground in the domain of small embedded & real-time systems. If object-oriented concepts and component-based design principles are only facilitated in the analysis and design phases and basic implementation technology is used afterwards, it seems to be even more inapplicable, because then, a significant breach between design model and implementation model, i.e. source code, is the result.

Another problem inherent to the application of higher abstraction techniques is related to the organizational constraints that often have to be faced. As already mentioned, software developers in the respective application areas are not always educated software engineers, so that profound training of the involved personnel is necessary to achieve a common familiarity with the applied concepts and technologies. This of course is a major hindrance to the adoption of such approaches because of the aforementioned immanent cost pressure.

4.2.2 Methodological Incompleteness and Discontinuity

Another challenge arises from the historic survey, software engineering has undergone from the late 1990's. As Section 3.1.2 unveils, the unification process of the various different methodological approaches of the early and mid 1990's, related to the Unified Modeling Language, has lead to a drop out of support for concrete design respectively construction methods in related tools. In fact, while before the unification process each approach had its own notation, method, and related tools, where these tools where equipped with methodical support at least to some extent, this support has more and more been removed from the respective tools afterwards (or the tools have disappeared), so that most of today's modeling tools do indeed not deliver any methodical support.

The lack of profound methodical support in today's tools can be regarded as a major drawback, as a coherent methodological system can only be achieved if notation, method and tool are closely integrated and rely on common concepts and principles (compare the system triangle metaphor in Section 2.3.2). If methodical support is not provided by respective tools, this can thus be regarded as a major hindrance to the application of a method. This is the case not only because the acquisition and adoption of the method is complicated, it is also because potentially automatable or semi-automatable steps of the respective method have to be performed by hand, naturally leading to a lower acceptance of the method or even to its disapproval.

Another aspect that hinders the applicability of model-based construction approaches is that most often tool support may not be regarded as being continuous. That is, from the early requirements analysis up to the detailed design, most activities are well supported by respective modeling tools (although each one of course has its own weaknesses and drawbacks). Nevertheless, in particular the transition from detailed design into the respective implementation are most often not profoundly supported. This is first and foremost the case because flexible and customizable code generation facilities are not offered by todays modeling tools and also because implementation support is mostly not integrated into them, but instead provided by additional third-party development environment.

4.3 Definition of Goals

It is nevertheless assumed that model-based software engineering is a promising approach, especially within the domain of small embedded & real-time systems. This is first, because it accommodates a more systematic approach to software engineering, leading to an enhanced traceability, and second, because it offers an increased potential in terms of reasoning and analyzability. However, this great protential can only be unleashed, if the aforementioned problems and challenges are considerably taken into account.

Being aware of the methodological incompleteness and discontinuity that is inherent to most existing model-based approaches, it has to be statet that an effective model-based solution has to be arranged as a complete, continuous, and integrated methodology. Taking a goal-oriented viewpoint, this may be formalized as:

Goal (Methodological Integrity). *An overall software construction methodological approach has to be delivered in terms of a concise method, an appropriate notation, and adequate tool support, being all related by common concepts and principles.*

(i) ***Methodological Completeness*** *- A wholistic approach has to be provided, covering all constructive software engineering activities from the early analysis up to the late detailed design and its final transition into the resulting implementation. This in particular has to include a precise definition about the structure of the employed models, as well as a thoroughly documented mapping between design and programming language constructs.*

(ii) ***Methodological Integration*** *- To form an overall integrated methodology, both, a method and a supporting tool, have to be provided, being integrated with each other via common concepts and principles. To allow easy adoption and acquisition of the method, and to enable a high degree of automation, dedicated methodical tool support has to be in particular provided, including substantial support for the execution of the method's tasks, as well as for its chronological procedure.*

Model-based software construction can furthermore only be successfully adopted to those small embedded & real-time systems within the marginal application areas, if the very special technical constraints of those systems and additionally the also rather special organizational constraints of their respective application areas are properly reflected, so that the second major goal of the presented approach may be denoted as follows:

Goal (Constraint-Adequateness). *The delivered methodology has to properly reflect the very special technical and organizational constraints of small embedded & real-time systems in the marginal application areas.*

(i) ***Technical Adequacy*** *- It has to be ensured that the very special technical constraints of small embedded & real-time systems are met. That is, in particular*

*resource and timing constraints have to be explicitly and continuously regarded (**Real-Time Awareness**). Concepts that would lead to a breach between design and implementation have to be avoided (**Seamless Continuity**). Furthermore, no specific technology has to be prescribed or enforced, which could hinder the applicability of the approach (**Technological Insensibility**).*

(ii) ***Organizational Adequacy** - The very special organizational constraints, being faced in the marginal application areas have to be met. It thus has to be ensured that a solution is practically applicable and understandable (**Practical Applicability & Understandability**), in particular where software development is performed by others than educated software engineers. It furthermore has to be ensured that standard-conformant languages and tools are employed whenever possible, as the organizational and economical constraints, which are being faced in the marginal application areas, do not facilitate the development of proprietary languages or tools (**Standard Conformity**).*

These goals have to be inescapably addressed if a model-based software construction approach is intended to be capable of meeting the very special characteristics of small embedded & real-time systems, and if it wants to be compatible with the special organizational constraints, which are being faced in the respective application areas, as well.

Part II

The MeDUSA-ViPER Methodology

Chapter 5

A Model-Based Methodology

5.1 Sketching A Solution

Having clearly defined the problems, challenges, and goals, it is time to come up with a solution.

MeDUSA (Method for UML-based Construction of Embedded & Real-Time Software), the method, which forms a central part of the solution, is a model-based software construction method, facilitating the intense use of models as the central engineering artifacts throughout the overall software construction. Being organized around the use case concept, the method was designed to be systematic and straightforward. It uses the UML in its current version 2.1.2 [OMG07d] as underlying notation and - unlike most other published methods, which are rightly referred to as design methods - explicitly covers all software construction steps from the early requirements modeling up the late implementation. This in particular includes a seamless transition of a UML-based detailed design into a procedural implementation in the ANSI-C language, what has been repeatedly emphasized as crucial to the success of any model-based approach within the domain of small embedded & real-time systems, and which has conclusively lead to the formulation of the methodological completeness goal (cf. Section 4.3).

Having repeatedly pointed out the importance of an overall integrated methodological solution, it is clear that MeDUSA, even if being a crucial part of it, does not make up the overall solution. Indeed, as sketched by Figure 5.1, together with the underlying languages it is part of a larger methodological approach, in which *ViPER (Visual Tooling Platform for Model-Based Engineering)* forms the complementary part of the supporting tool.

ViPER delivers graphical UML modeling capabilities, as well as UML-to-ANSI-C code generation (cf. [FNL08]). Additionally a toolkit for editing and simulating of textual, narrative use case descriptions, as proposed by the method (cf. [WNHL08]), is offered, as well as dedicated methodical support for MeDUSA.

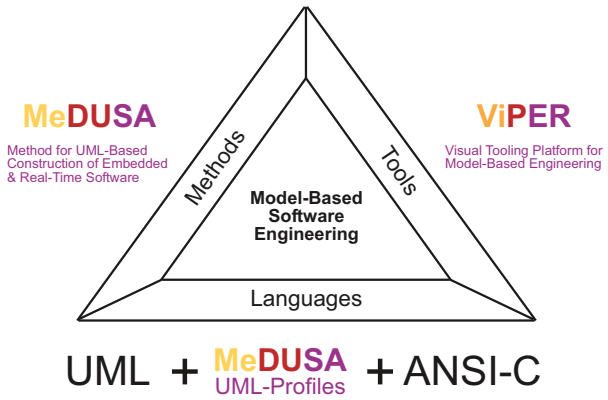


Figure 5.1: The ViPER-MeDUSA Methodology

All three building blocks of the sketched solution will be investigated in detail within the following chapters. As the method forms an integral part of it, it will be taken as a starting point. Subsequently, the employed languages will be paid attention to. That is, it will be elaborated, what parts of the UML are actually used within the construction of all MeDUSA UML models, and it will be detailedly elaborated how design model concepts can be seamlessly transferred into respective ANSI-C equivalents. The description of the methodology will be completed by a presentation of ViPER, the supporting tool. Here, concentration will be spent on the MeDUSA specific methodical support that is offered.

5.2 Related Work

Of course, a methodology as the one presented herein is never developed *in the open countryside*. Before thus introducing the components of the presented solution in terms of MeDUSA, ViPER and their related modeling and implementation languages in the following chapters, closely related work will be introduced in the following.

For MeDUSA, this will be done in terms of Gomaa's *COMET* method [Gom00], which may be denoted as the precursor of MeDUSA, as well as *ROOM*, by Selic et.al [SGW94], which may be quoted as having strong impact on the applied notation and of the integrated modeling between structural and behavioral aspects, as it is incorporated into MeDUSA.

In the context of ViPER, it will be investigated to what extend methodical tool support is provided by currently available state-of-the-art tools, as this is regarded to be the key focus of the herein presented work. Even while the respective functionality is built into ViPER as well, investigating to what extend UML modeling capabilities or UML-based code generation is offered by other approaches, is omitted here. This is, because the offered UML modeling capabilities are anyway more or less comparable, and because - even while there are several shortcomings related to the code genera-

tion facilities of those tools, to which the *ViPER UML2 Code Generator* offers some innovative solutions - this is not regarded to be at the heart of the herein presented methodology. The interested reader may be delegated to refer to [FNL08], [Fun06], or [Kev07] to get an in-depth discussion on this.

5.2.1 Methods

Indeed, besides the *COMET* method, which served as its starting point and its first and foremost inspiration, MeDUSA comprises and incorporates various ideas and practices originating from other, sometimes earlier approaches. It would be tedious work to deliver a complete list of all direct and indirect influences of related development approaches. However, at least the two most relevant ones, namely *COMET* and *ROOM*, have to be quoted to enable a deep understanding of the MeDUSA method, and to allow a well-founded assessment of the actual contribution acquired by it.

COMET

Being published in 2000, *COMET* may be named as one of the *second generation* object-oriented approaches for embedded & real-time systems. While its direct predecessor *Software Design Methods for Concurrent and Real-Time Systems (CODARTS)* [Gom93], could still be characterized to be some sort of structured analysis & design approach, even if it already employed certain object-based/object-oriented concepts, *COMET* can be regarded as being a truly object-oriented method.

In detail, *COMET* is based on the object-oriented iterative life-cycle model denoted by Figure 5.2, out of which it covers *Requirements Modeling*, *Analysis Modeling*, and *Design Modeling*. Being an object-oriented method, requirements modeling is done in terms of use case modeling, using the graphical notation proposed by the UML, extended by a textual, narrative description for each identified use case, which captures the steps of the default scenario as well as single alternative steps.

Analysis Modeling is done in terms of static modeling of the system context and of internal long-living data, and in terms of dynamic modeling of object collaborations for each identified use case, including specification of intra-object behavior for state-dependent objects, participating in the collaborations. In detail, *Static Modeling of the System Context* is done by developing a class diagram, showing external hardware systems, software systems, or human users in the surrounding environment of the system-under-development, as well as corresponding internal interface classes. *Static Modeling of Entity Classes* is then done to identify the internal long-living data objects. It is performed using one or more UML class diagrams, depicting data intensive classes and their relationships in terms of generalizations and associations.

Having identified interface and entity classes during static modeling, *Dynamic Modeling* is then performed to identify further analysis objects/classes. During dynamic analysis of inter-object collaborations, UML collaboration diagrams are developed

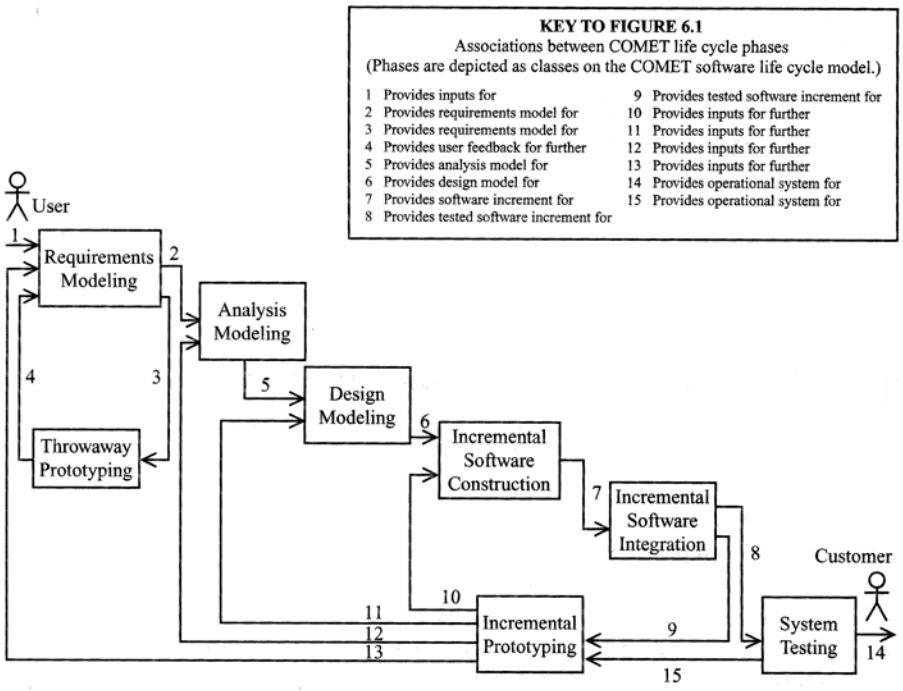


Figure 5.2: COMET Object-Oriented Lifecycle Model (cf. [Gom00])

to find and describe an object collaboration for each identified use case, so that the inter-object behavior of the respective object collaboration jointly performs goal of the respective use case. The identification of the analysis objects, forming an object collaboration for a use case, is supported on the one hand by reflecting the beforehand identified interface and entity classes, and additionally via the object respectively class taxonomy displayed in Figure 5.3, which is referred to as the *Object and Class Structuring Criteria*.

Having modeled the system-under-development in terms of analysis objects together with their inter-object collaborative and their (state-dependent) intra-object behavior, *Design Modeling* can be subsequently performed. First, the identified analysis model artifacts are synthesized, which is done in terms of synthesizing statecharts, the collaboration model, and the static model. Synthesizing statecharts is done in terms of synthesizing the partial state-dependent behavioral facets, a state-dependent control objects unveils in the different object collaborations, it participates in, to an overall statemachine describing its complete intra-object behavior. Synthesizing the collaboration model is done in terms of consolidating the different collaboration diagrams, developed during *Analysis Modeling*, into an overall collaboration model. Synthesizing the static model is done by combining the different class diagrams for the interface and entity classes to an overall static model of the system-under-development, and by extending classes, which can be inferred from the objects, being identified during *Dynamic Modeling*.

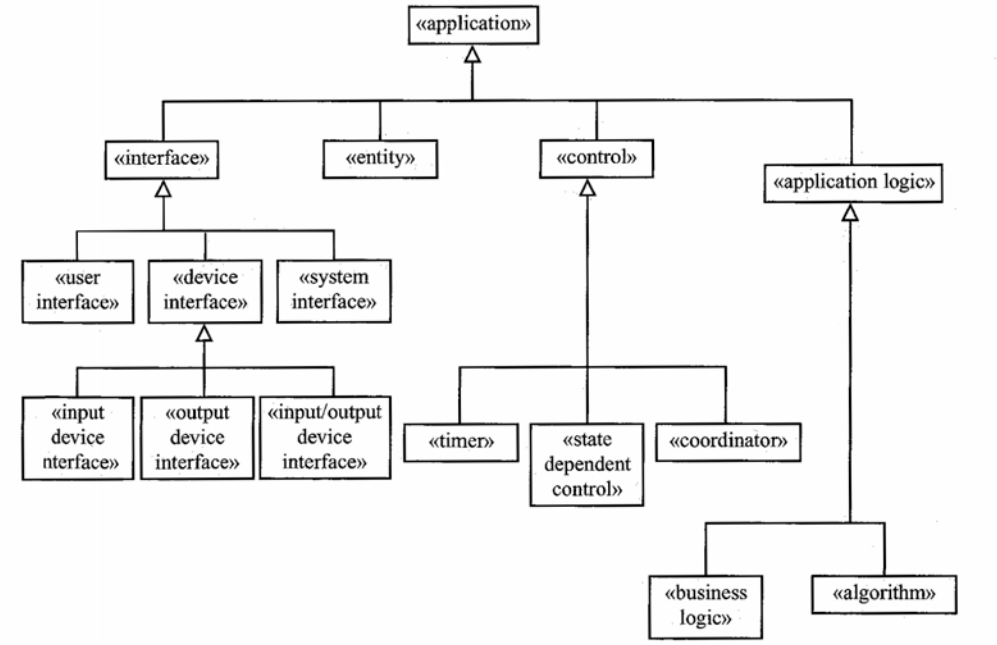


Figure 5.3: COMET Object/Class Structuring Criteria (cf. [Gom00])

After synthesizing the analysis artifacts to a consistent overall analysis model, the overall system architecture can be developed. Here, identification of subsystems is the first step. This is done by applying so called *Subsystem Structuring Criteria*, which basically is a taxonomy of subsystem stereotypes similar to that provided for objects/classes (although it is not disjoint). The goal is to obtain a consolidated collaboration diagram for each subsystem, showing the objects composed by it, as well as a high-level collaboration diagram for the overall system. Based on this initial system architecture the concurrent task architecture is then developed by structuring the subsystems into concurrent tasks and by designing their respective task interfaces, which is collectively referred to as *Task Structuring*. Subsequent to this, the performance of the design is analyzed in terms of a real-time scheduling analysis. If the overall architecture is sufficiently performant, *Class Design* is then executed for each subsystem. Here, the classes' interfaces in terms of all offered operations, as well as the inheritance hierarchies between the identified classes are designed.

Last, the detailed design is developed for each subsystem. This is done in terms of designing the internals of the composite tasks by identifying all nested passive information hiding classes and by designing task synchronization mechanisms for those nested classes, which are accessed by multiple tasks, as well as by designing connector classes to realize inter-task communication. After having designed each task's internal sequencing logic, the detailed design of each subsystem is concluded with a more in-depth analysis of the subsystem's performance. Each subsystem can then be implemented.

A concluding assessment of the *COMET* method may come to the result that compared to other approaches in the respective field, the *COMET* method has some outstanding features, which also mainly influenced the conception of the presented approach. In detail, these are:

- a systematic, iterative approach to develop embedded & real-time software
- taxonomies to support identification and structuring of objects/classes, subsystems, and tasks
- a standard notation, in terms of the UML and extending profiles (taxonomies)
- an integrated performance analysis based on real-time scheduling theory

However, the practical application of *COMET* in industrial pilot projects conducted at the German ABB Research Center as well as ABB Business Unit Instrumentation showed some weaknesses and shortcomings, which were already thematized in earlier publications [NL07a]. In detail, it may be criticized that *COMET*

- facilitates an object-oriented design, thus deteriorating the transformation of the detailed design into a procedural implementation
- introduces a lot of overhead by extensive modeling on the class and object level, which would not be needed in case of an underlying procedural implementation (here, the runtime structure in terms of objects could be modeled directly)
- is based on an outdated version of the UML standard and does thus not make use of the new modeling capabilities offered by the current UML language standard version 2.1.2
- does not facilitate reuse of components, as no means for selection and integration of such components is given

It may thus be concluded - as it was already done earlier (cf. [NMSL04]) - that *COMET* is a by all means a practically applicable design method, having however some drawbacks, which express themselves especially in the context of those constraints, which are being faced in the marginal application areas that are targeted by the herein presented approach. While it is thus not an unrestrictedly adequate method to develop software for small embedded & real-time systems, it may however be characterized as a very systematic method, additionally offering some novel outstanding properties, for instance the provided structuring criteria as well as the integration of performance analysis into software design.

ROOM

Real-Time Object Oriented Modeling (ROOM) was published by Selic, Gullekson, and Ward in 1994, fairly at that time that Gomaa published *COMET*'s predecessor

CODARTS. It has to be mentioned explicitly as well, mainly because of its modeling capabilities in terms of hierarchical structures and because of its close integration between structural and behavioral aspects of a software architecture. Unlike *COMET* or *CODARTS*, *ROOM* does indeed not provide a concise method. Instead, software construction is understood merely as modeling on different levels of abstraction. Using a *phase-independent set of modeling abstractions*, ROOM pursuits the goal of *phase independence*. It thus covers how modeling for different levels of abstraction has to be performed, but it does not provide detailed guidance on what to model in which timely order.

However, the major contribution of *ROOM* is its well designed, very concise and semantically clear modeling language. Unlike other approaches of its time, *ROOM* offers the outstanding possibility to model hierarchical system structures. So called *actors*, *ROOM*'s denomination for objects, i.e. *"independent, concurrently active logical machines"* [SGW94], are defined by corresponding *actor classes*, whose internal decomposition can in turn be specified by other nested actors, thus allowing to model arbitrary nested system structures.

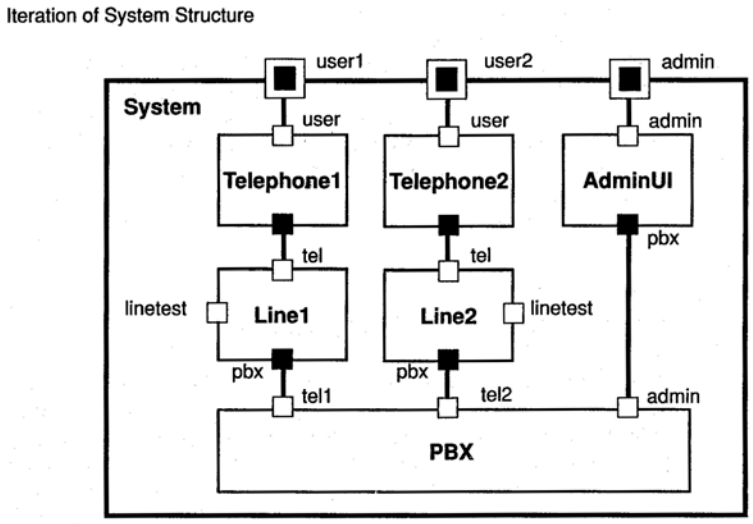


Figure 5.4: ROOM Hierarchical Modeling Example - System (cf. [SGW94])

For illustration purposes, consider the example presented in Figures 5.4 and 5.5, which depicts a simple telecommunication switching system. As denoted by Figure 5.4, the actor definition of a respective *System* actor is specified as a hierarchical structure, being decomposed into nested actors, namely *Telephone1*, *Telephone2*, *AdminUI*, *Line1*, *Line2*, and *PBX*, interconnected to each other via named *Ports* and so called *Bindings*, to establish inter-actor communication. The nested *PBX* actor, whose actor class definition is denoted in Figure 5.5, is itself internally decomposed, leading thus to an overall hierarchical system design (of course, its internally nested actors may in turn be hierarchically structured).

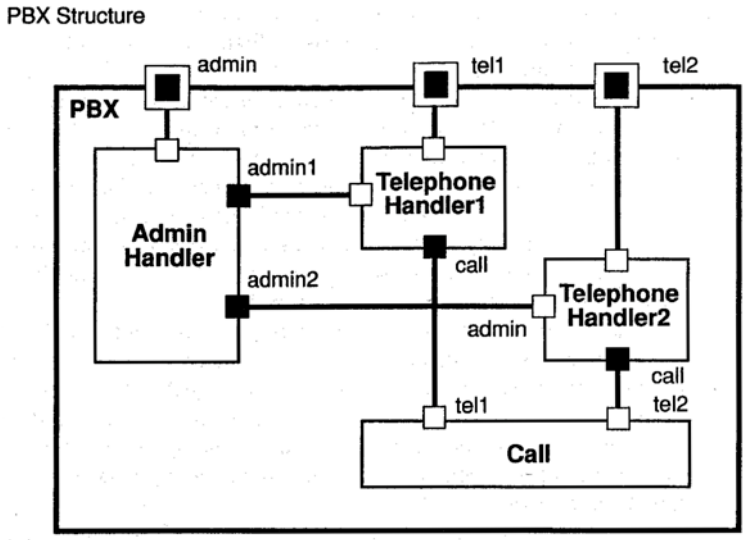


Figure 5.5: ROOM Hierarchical Modeling Example - Subsystem (cf. [SGW94])

The full encapsulation of actors that is being expressed in terms of the ports concept, may be mentioned as another outstanding feature of the *ROOM* language. That is, the externally visible interface of a respective actor is solely defined in terms of its ports, which are the only interaction points, via which communication with the actor can be established. This leads to an explicit definition of all context dependencies, an actor might have, as well as to a clear encapsulation of the actor' internal structure (if it has one). To illustrate this, the *PBX* actor may be quoted again. It fully encapsulates its internal structure towards the other actors inside the system and allows inter-actor communication only via its *tel1*, *tel2*, and *admin* ports.

Last, the seamless integration between structural aspects of a system and behavioral aspects, which is achieved by the *ROOM* language, has to be mentioned. That is, actors have internal behavior that can be expressed in terms of a finite statemachine, being expressed by a so called *ROOMchart*, as exemplarily denoted in Figure 5.6 for the *Call* actor nested inside the *PBX*. In detail, interconnection between structure and behavior is consistently achieved using ports as well. That is, two types of ports are actually supported by the *ROOM* language, namely *relay ports*, which simply prolong incoming external communication to a nested internal actor and vice versa (as in case of the *admin*, *tel1* and *tel2* ports in the PBX actor reference), and *end ports*, which are directly connected to the actor's internal behavior. That is, the arrival of a message at an end port of an actor causes a respective signal to occur within the *ROOMchart*, expressing the internal behavior of the actor, so that state transitions and resulting actions can occur (which may e.g. be the processing of the message and the sending of a respective return message).

While not going further into detail about additional modeling capabilities of the *ROOM* language, it can be concluded that the aforementioned advantageous novelties had significant influence on later approaches, including the one presented here. It can for

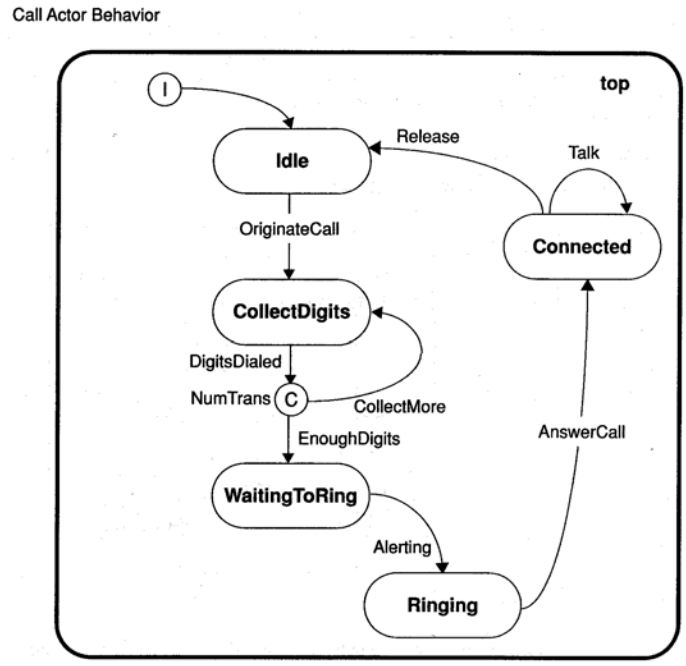


Figure 5.6: ROOM Behavioral Modeling Example (cf. [SGW94])

example be inferred from the fact that via its direct successor *UML-RT*, which was incorporated into Rational's *Rose Real-Time* modeling tool, hierarchical modeling and the encapsulation concept, as well as the close integration between structural and behavioral aspects have finally made its way into the current UML standard version. There, precisely this is now also expressible in terms of composite structures and component diagrams, as well as closely integrated behavioral diagrams as state machines, interactions, or activities.

A significant contingent of *ROOM*'s success may also be attributed to the fact that profound tool support was delivered for the *ROOM* modeling language in terms of the *ObjecTime* tool set. Being equipped with a model manager, editor, validator, compiler and even with model execution capabilities, the *ObjecTime* tool set enabled profound engineering of ROOM models.

However, *ROOM* also had its drawbacks, which in particular also holds for the *ROOM* modeling language, even if aforementioned advantages do somewhat outweigh them. One major drawback is that apart from the explicitness and clarity that the port concept brings in terms of encapsulation and context dependencies, it is somehow not as straight-forward applicable as it seems to be. The different port types (relay, end) in combination with the concept of protocols (a specification of how messages are exchangeable via a certain port) and the conjugation concept (a port is a conjugation if it has a corresponding protocol with an inversed direction, i.e. for every incoming message it has a respective outgoing message and vice versa) makes the composition of actors somewhat unclear and complex. Another disadvantage is the rather intricate

modeling of layered structures and the difficulty to specify non state-dependent actor behavior. The most significant drawback of *ROOM* however may be seen in the lack of a profound method. Even if the modeling language is to an overall extend clear and precise, its applicability is not naturally given, and an uncoordinated and unsystematic application of the respective modeling elements, based on heuristics, may not be regarded as being goal-oriented.

5.2.2 Tools

A complete list of all tools related to the herein provided ViPER tool would of course - as in case of those methods that have somehow inspired the development of MeDUSA - be quite inexhaustible. Various programming and modeling tools as well as integrated development environments for programming and modeling have emerged during the last decades, and all of them probably have added their own partial contributions, so that investigating the originator of all the concepts and principles, being incorporated into one of todays state-of-the-art integrated development environments, is hard to achieve.

Guided by the goal of methodological integration, which was formulated in Section 4.3 as one of the key goals, the focus may thus be spend on investigating, to which extend *methodical support* is offered by current state-of-the-art tools, as this is a strong indicator for the applicability and usefulness of a methodology. Profound tool-based support for the execution of individual tasks (steps, guidances, etc.), as well as sophisticated support on the creation and assessment of the developed work products (metrics, checklists, etc.) can decisively and extensively support a role performer. Even basic methodical support in the form of a simple browser access to an electronic method definition may be helpful.

Most of today's available UML modeling tools however do not come with such a methodical support, not even in its simplest form. One major reason for this may be found within the historic course of the UML, as it was briefly sketched within Section 3.1.2. That is, having originated from various object-oriented methodical approaches, the UML was developed as a unified notation, not incorporating any concrete methodical aspects of the underlying approaches. While most tool vendors offer some sort of installation modes to customize the functionality of their tool to best meet a respective role performer (e.g. developer and designer installation modes), concrete methodical support for a specific method can thus only be observed, where the respective tool vendor is at the same time also method vendor. This is for example the case with IBM Rational, which is vendor of the Rational Unified Process and delivers a set of UML modeling tools in terms of the *IBM Rational Software Development Platform*.

More sophisticated methodical support is on the other hand offered solely by dedicated methodical support tools. Here, *Jaczone Waypointer* may be named, which is an agent-based tool to support the execution of the *Rational Unified Process*. It has the capability to integrate itself into the *Rational Software Development Platform* and may thus be quoted as a good example on how more dedicated methodical support may be provided.

IBM Rational Software Development Platform

The *IBM Rational Software Development Platform* is IBM Rational's four member product-line to facilitate UML-based model-based design. The comprised products, namely the *IBM Rational Software Modeler*, *IBM Rational Systems Developer*, *IBM Rational Application Developer*, and *IBM Rational Software Architect* differ in detail about the concrete programming languages and technologies, being covered, as well as the application domains, being targeted, which ranges from the development of small embedded software up to that of large industrial enterprise applications.

Regarding methodical support they are all equipped with a *Process Advisor* and *Process Browser* component to support development according to the Rational Unified Process. The *Process Browser*, depicted by Figure 5.7, is a hypertext browser, which allows to navigate a UMA-based[1] definition of the *Rational Unified Process*.

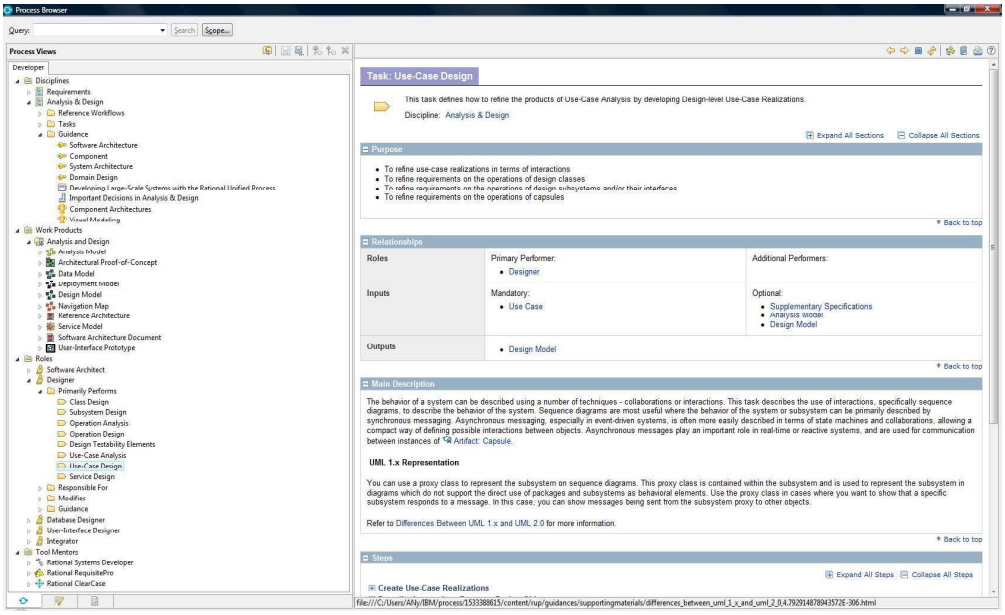


Figure 5.7: IBM Rational Software Development Platform - Process Browser

[1]The methodical support within the IBM Rational Software Development Platform is based on the IBM Rational Method Composer, which in turn uses the Unified Method Architecture (UMA) as underlying notation. Compare Section 6.2.1 for more details.

The *Process Advisor* on the other hand provides context-sensitive guidance, in terms of *Tools Mentors*, *Work Products*, and *Tasks*, covered by the RUP definition. It thus offers some sort of dynamic assistance, as the provided excerpts of the RUP documentation are filtered according to the current selection of resources. So, as outlined by Figure 5.8, when editing a UML use case diagram, the display of RUP contents is limited to those *Tools Mentors*, *Work Products*, and *Tasks* that are related to modeling of use case diagrams.

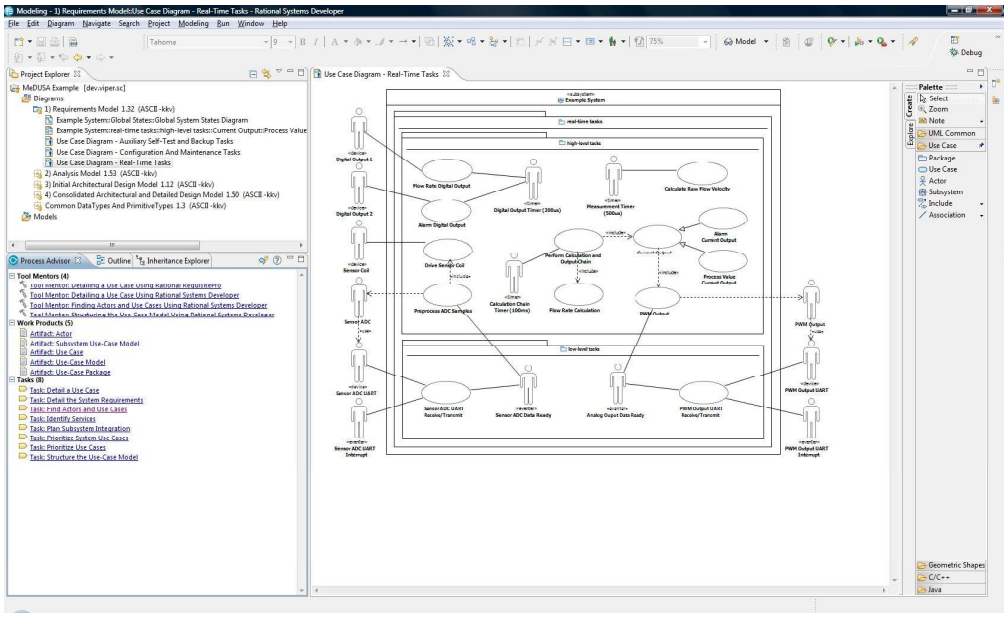


Figure 5.8: IBM Rational Software Development Platform - Process Advisor

Besides offering such a context-sensitive access to the static RUP documentation, dynamic support is not delivered. That is, in particular, no specific wizards or agents are offered by the platform to support a RUP role performer during the execution of a specific task, e.g. by automatizing or semi-automatizing certain steps.

Jaczone Waypointer

Jaczone Waypointer by Ivar Jacobson International is a tool explicitly delivering methodical support for the *IBM Rational Unified Process*[2]. The tool may be integrated with Microsoft Word and the IBM Rational Software Architect to manage the development and refinement of documents and models according to the IBM Rational Unified Process. Jaczone Waypointer is organized around the *Activities* of the IBM Rational Unified Process, so that the *Activity Window*, which is depicted by Figure 5.9, serves as the central interface for interactions with the user.

[2] Actually the process supported by Jaczone Waypointer is a subset and a refinement of the IBM Rational Unified Process, covering parts of the *Requirements*, *Analysis & Design*, and *Project Management* disciplines, and refining them with additional steps and more fine grained activities (cf. [Jac08]).

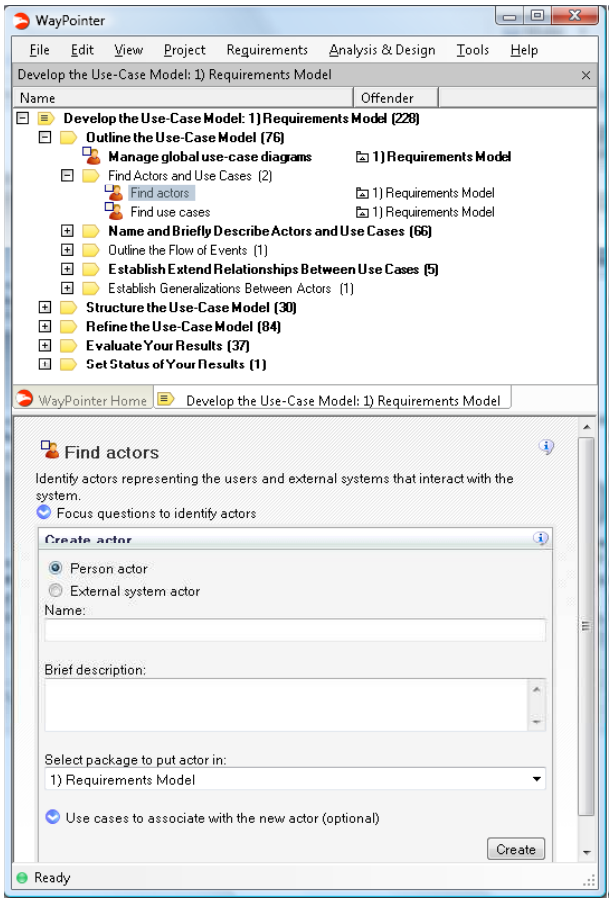


Figure 5.9: Jaczone Waypointer - Activity Window

Within Waypointer, an activity is defined to be *"a well-defined unit of work that a user or a group of users carry out when adopting a role."* [Jac08]. It is usually structured into a set of steps, which are offered for selection by a respective *Activity Agent* within the *Activity Window* of Jaczone Waypointer, associating each step with a set of rules, concretely manifested as checks or conclusions. Steps may also be supported by a set of *Wizards*, which are *"kind of dialogue[s] with the user, during which conclusions and checks are traversed in sequence"*, and which usually *"encapsulate smaller modeling tasks [..] within a specific activity"* [Jac08]. An example for such a *Wizard* is depicted by Figure 5.10. It shows one page of the *Split a Use Case Wizard*, which is available from within multiple *Activity Agents* of the *Requirements Discipline*.

Besides *Wizards*, *Artifact Agents* support a role performer while working with a concrete artifact. As exemplarily depicted by Figure 5.11, an *Artifact Agent "collects a set of rules that are applicable for a particular type of artifact"* [Jac08] and thus helps to point a role performer to inconsistencies and defects within the respective artifact. It also offers a set of *Conclusions*, which may for instance be the proposed execution of a respective *Activity Agent* or *Wizard*.

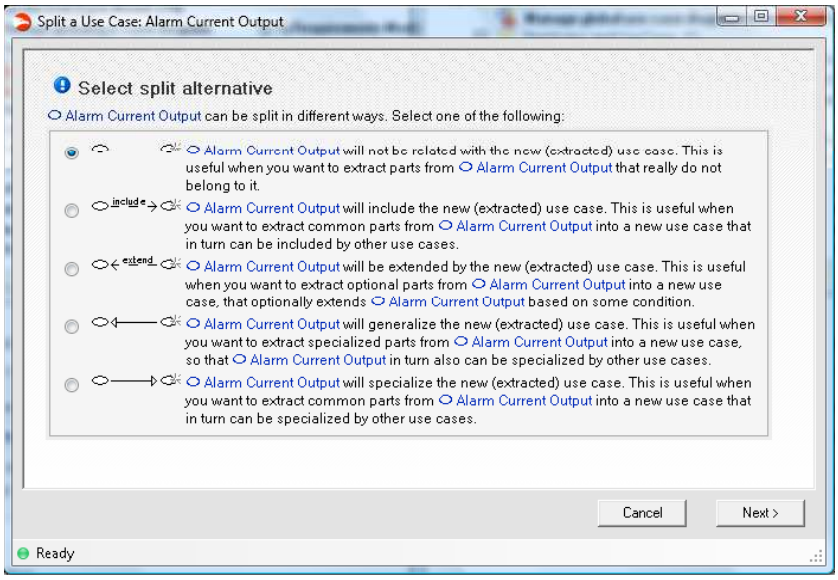


Figure 5.10: Jaczone Waypointer - Wizard Example

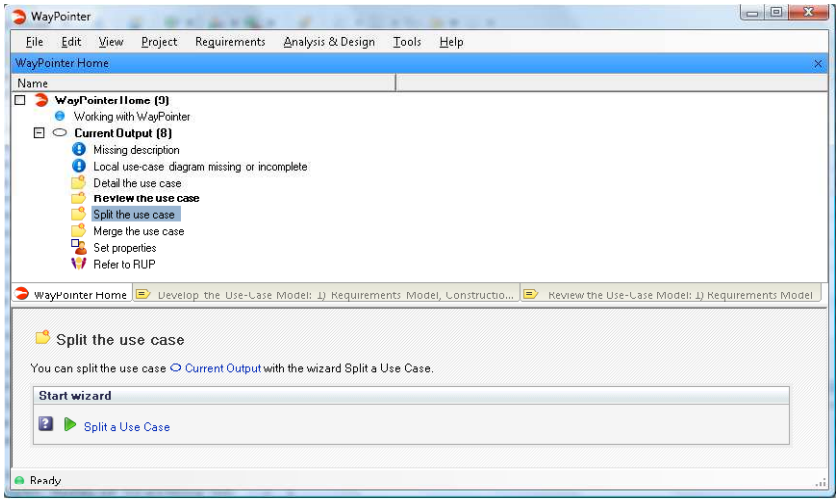


Figure 5.11: Jaczone Waypointer - Artifact Agent Example

While the execution of individual activities and the handling of individual artifacts are thus relatively well supported by the *Jaczone Waypointer*, the timely execution of activities, as defined by the *IBM Rational Unified Process*, are not explicitly thematized. A user can however select to adopt certain roles, as defined by the *IBM Rational Unified Process*, so that the *Jaczone Waypointer* offers only those activities, the respective role is involved in. Nevertheless, the tool does not further guide him with respect to the timely execution of those activities, applicable to his adopted roles.

Chapter 6

Method - MeDUSA

Having started in early 2005 as a slight enhancement to the *COMET* method that was developed in conjunction with the ABB Corporate Research Center, *MeDUSA (MethoD for UML2-based Construction of Embedded & Real-Time SoftwAre)* has undergone several changes in the following years. The evolution of the method hit - as indicated before - its first peak in its initial publication in 2007 [NL07a]. Having gained further experiences in the following, a total revision of the method, denoted as *MeDUSA Second Edition* was published in mid 2008 [NL08]. The MeDUSA definition provided in the following sections is based on this completely revised Second Edition. However, some changes related to the modeling and analysis of timing and concurrency constraints, which have been investigated in depth since the publication of the *Second Edition* have been already incorporated[1].

6.1 MeDUSA-Lifecycle

MeDUSA is a mere *software* construction method. That is, only software engineering concerns are addressed, and the development of electronical and mechanical hardware, which are the other essential parts of system development, are explicitly not covered.

To understand the conception of MeDUSA, those aspects however have to be regarded as well, as indeed the different disciplines are interrelated to each other. An overall system development life cycle may be assumed, as it is outlined by Figure 6.1. While there are system wide activities as the initial elicitation of system requirements or the final integration of software, electronical hardware, and mechanical hardware to an overall system, the development inside the different disciplines may be regarded to be executed more or less concurrently within different development threads.

[1] As MeDUSA is regarded to be a living method, further changes are likely also after the publication of this thesis work. The most recent version of the method definition can always be obtained from the MeDUSA project site [MeDUSA].

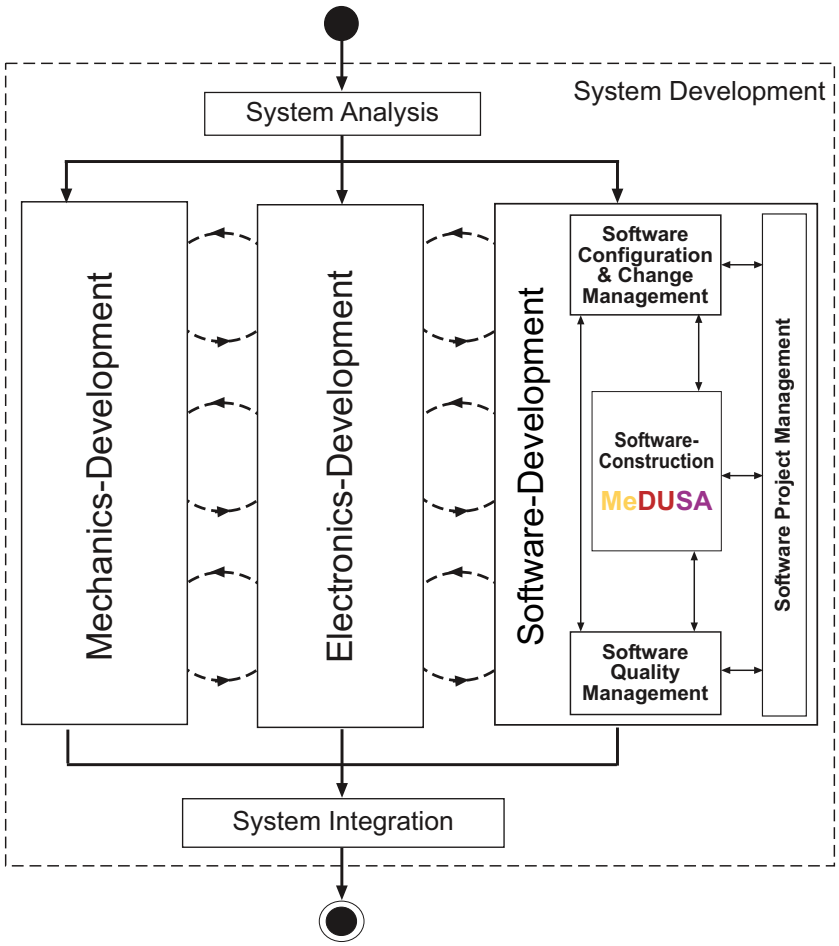


Figure 6.1: The MeDUSA Life Cycle

Those threads are however not unrelated to each other, as indicated within Figure 6.1 as well. Mechanical and electronical hardware are for example closely related, because they have to jointly address questions of housing design or circuit design, both restricted by the non-functional requirements for the overall system. Electronics development and software development are even more intensely coupled, as the electronical hardware indeed forms the embedding system for the embedded software. Thus, all aspects related to the real-time & embedded nature of the software are somehow related also to the electronical hardware, and as such have to be jointly addressed by both disciplines. However, in the context of MeDUSA, it is assumed that hardware development is performed with a slight advance, so that an initial hardware design has been developed when starting the MeDUSA *Requirements Phase*, and a hardware prototype is available not later than before starting *Implementation Phase*, at least in the form of some breadboard, so that parts of the software can already be integrated with the electronical hardware. It is reasonable to do so, because, due to the technical and economical constraints in the respective application domains, hardware development is regarded to be the main driver of system development.

What has to be emphasized again is that MeDUSA is a mere software *construction* and not an overall software development method. That is, out of all software engineering activities only the constructive ones are being covered, and quality management, configuration & change management, as well as project management (cf. Section 2.3.1) are not addressed[2]. MeDUSA thus does not only have to be embedded into a larger system development process, but indeed, as indicated by Figure 6.1, also into some sort of umbrella software development process, that addresses all other software engineering disciplines to their required extend.

6.2 MeDUSA-Definition

With its first publication [NL07a], the MeDUSA method has been formally defined based on the *Unified Method Architecture (UMA)* meta model [Hau05][Hau06], which was developed by IBM as an advancement to OMG's *Software Process Engineering Meta-Model (SPEM)* 1.1 standard [OMG05a].

UMA was chosen for the definition of MeDUSA for two main reasons. First, because profound tool support for editing and publishing of UMA-based method definitions is offered in terms of IBM Rational's commercial *Method Composer* tool, as well as its non-commercial open-source derivate *EPF Composer*, offered by the *Eclipse Process Framework (EPF)* project [EPF]. Second, because UMA was - at the time of MeDUSA's initial publication - IBM's and its related OMG partners' candidate proposal for the then upcoming *SPEM 2.0* standard of the OMG [OMG08], and because it was intended that future versions of UMA would be aligned with the final version of the SPEM 2.0 standard.

As the adoption process of the current SPEM standard had been finished at that time, SPEM 2.0 was then chosen as notation to define the *Second Edition* of MeDUSA [NL08][3] and will thus be used herein as well. Before providing the detailed definition of MeDUSA in the following sections, SPEM 2.0 is therefore outlined shortly in the following.

6.2.1 SPEM 2.0

SPEM (Software Process Engineering Meta-Model) is an official OMG standard used to *"describe a concrete software development process or a family of related software development processes"* [OMG08], and is as such also best applicable to define a more lightweight methodical approach as the MeDUSA method. According to the standard document [OMG08], it incorporates a clear separation of the contentual aspects (what-to-do) of a method or process definition, the so called *Method Content*, and the

[2] An exception to this is the real-time analysis, being covered by MeDUSA, which is a somehow analytical task and may thus also be accounted to software quality management.

[3] Technically however, UMA is still used to define the MeDUSA method library, as the related *Eclipse Process Framework Composer* tool is used to generate its hypertext documentation (cf. Section 8.2.2)

time-related ones (when-to-do-it), which is referred to as the *Process*, as illustrated by Figure 6.2.

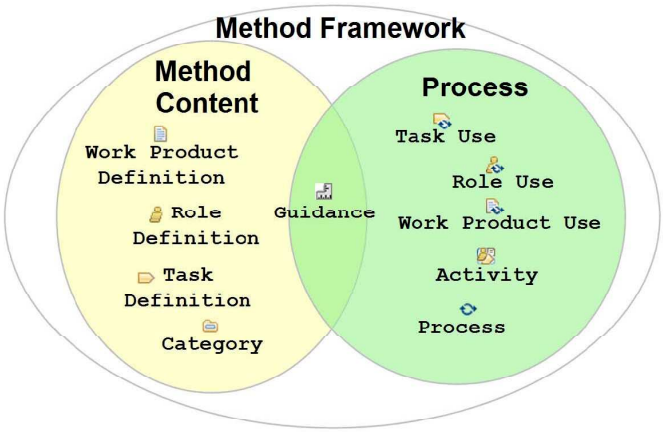


Figure 6.2: Terminology of SPEM 2.0 (cf. [OMG08])

Accordingly, the meta-model defines two basic abstract concepts, namely `Method Content Element` and `Process Element`, as denoted by Figure 6.3, upon which respective concept hierarchies are build up. As already indicated by their exposed illustration in Figure 6.2, `Guidances` play some hybrid role, as they are used in both concept worlds. Being `Extensible Elements`, `Guidances` of various different `Guidance Kinds` are supported. In detail, these are `Template`, `Estimation Consideration`, `Example`, `Checklist`, `Guideline`, `Concept`, `Estimate`, `Practice`, `Term Definition`, `Report`, `Tool Mentor`, `Supporting Material`, and `Whitepaper`.

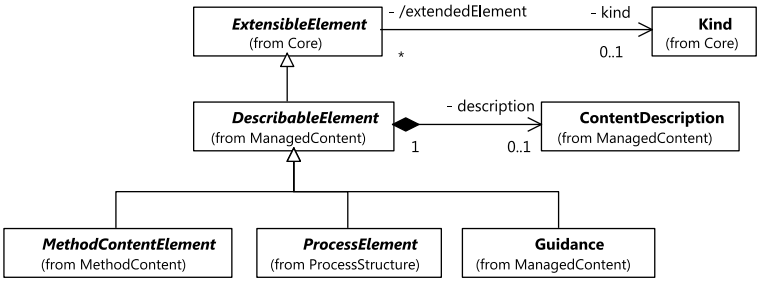


Figure 6.3: SPEM 2.0 Meta-Model - Top-level Hierarchy

Method Content The method content is basically defined in terms of `Task Definitions`, `Role Definitions`, and `Work Product Definitions`, as illustrated by Figure 6.4. `Task Definitions` are defined in terms of atomic `Steps` that describe the detailed actions that have to be performed. Their execution requires respective `Qualifications`, which have to be provided by related `Role Definitions`. To execute a `Task Definition`, a set of `ToolDefinitions` may be employed, which is accordingly used to manage related `Work Product Definitions`. `Role Definitions`

providing the Qualifications for a Task Definition may be defined as its default performers (formalized by Default Task Definition Performers). A Role Definition may be further assigned responsibility for the Work Product Definitions related to a Task Definition (formalized by (Default) Responsibility Assignments). Work Product Definitions may further be specified as (Default) Task Definition Parameters for a Task Definition, meaning that they serve as its input respectively output.

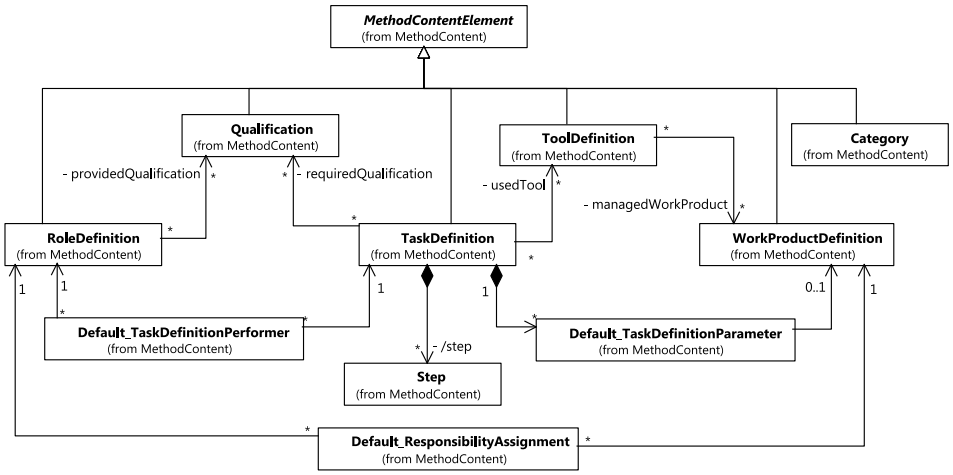


Figure 6.4: SPEM 2.0 Meta-Model - Method Content

To group related Method Content Element, SPEM 2.0 additionally offers an element called Category, which is extended by different Category Kinds for the specific Method Content Elements, namely Discipline (used to group Task Definitions), Role Set, Domain (used to group Work Product Definitions), as well as Tool Category.

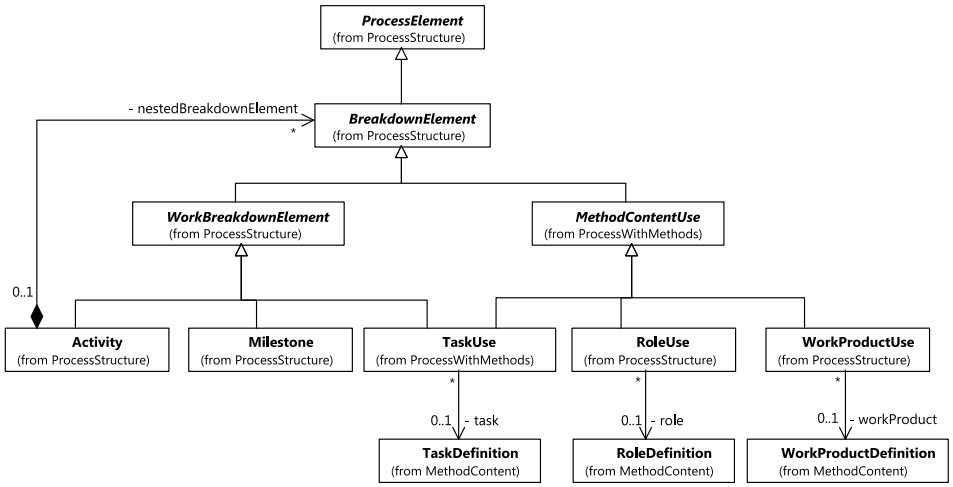


Figure 6.5: SPEM 2.0 Meta-Model - Process

Process The Process part defines how the different Task Definitions, defined in the Method Content, are performed over time. As indicated by Figure 6.5, this is done in terms of a breakdown-structure, built up of Task Uses, which reference respective Task Definitions of the Method Content, Milestones, and Activities. As Activities may furthermore contain nested Breakdown Elements, arbitrary hierarchical structures can be build up. Different Activity Kinds are supported by SPEM 2.0. Besides Iterations and Phases, Process Patterns, and Delivery Processes are supported, as well as Process Planning Templates. While a Process Pattern represents a reusable process building block, a Delivery Process is understood to be a full end-to-end lifecycle process.

Notational Clarification While the clear separation into *Method Content* and *Process*, as it is proposed by SPEM 2.0, is regarded to be very adequate, the employed terminology seems to be misleading, at least as far as the terms Process, Process Pattern, and Delivery Process are concerned. Denominating the definition of the time-related aspects of a method or process as Process seems to be somewhat misleading. The same holds for the term Process Pattern, which is regarded to be quite fuzzy. Last, the term Delivery Process to denote the complete definition of all MeDUSA related activities in their timely order seems to be inappropriate, in particular with respect to the definitions provided in Section 2.3.2, as MeDUSA is regarded to be a method rather than a process.

Because of this an alternative denomination will be used here. The term Method Operations will be used in the following to refer to what it denoted as Process in SPEM 2.0 terminology, namely the definition of the time-related aspects of the method. Additionally, the terms Workflow and Workflow Pattern will be used to refer to the concepts subsumed by the terms Delivery Process and Process Pattern respectively. All other terms are adopted from the current SPEM 2.0 specification.

6.2.2 MeDUSA Method Content

The MeDUSA *Method Content* is defined in terms of six disciplines, grouping 18 MeDUSA defined tasks. In detail, the disciplines, being covered, are:

- *Requirements Modeling Discipline*
- *Analysis Modeling Discipline*
- *Architectural Design Modeling Discipline*
- *Detailed Design Modeling Discipline*
- *Implementation Discipline*
- *Real-Time Analysis Discipline*

While each discipline groups logically and timely related tasks, it does however not define a chronological order between them. And while this should also not imply a strict chronological order amongst the disciplines, five of the six disciplines were named to match certain phases within the software development lifecycle. This was done to denote that those tasks, being comprised by the respective disipline, are usually first executed when the respective phase is commenced.

MeDUSA comprises 12 modeling related tasks, being subsumed by the *Requirements*, *Analysis*, *Architectural*, and *Detailed Design Modeling Disciplines*. The modeling tasks subsumed by a respective discipline may be regarded to be related in a sense that they yield model artifacts on the same level of abstraction. For instance, all tasks subsumed by the *Analysis Modeling Discipline* are subject to modeling of the problem domain. As this in turn requires that their individual contributions have to be consistent and integrabel with each other, the modeling tasks of a respective discipline, may indeed be understood to contribute to a common, shared, discipline specific model, which consistently integrates their individual contributions. Accordingly, the MeDUSA *Method Content* defines three such models[4], namely:

- *Requirements Model*
- *Analysis Model*
- *Design Model*[5]

While an *Implementation Model* may also be defined as the fourth integrated model within MeDUSA (cf. [NL08]), this model indeed manifests itself as mere source code, and is as such not subject to modeling but to implementing. It is further not comparable to the other formal engineering models in terms of its covered abstraction, and does also not yield comparable challenges in terms of consistency and integration.

Having all this in mind, the *Method Content* will be outlined in detail in the following along the defined disciplines. For each discipline, the common purpose of its subsumed tasks will be summarized first, and all roles involved in the execution of its subsumed tasks will be named. Subsequently, each task will be described in detail, naming its contribution in terms of its produced work products, where - for the sake of simplicity - only diagram related contributions will be explicitly named, and the underlying (UML) model contributions will not be elaborated in detail. They are subject to an in-depth discussion within Section 7.1, where the precise structure of all MeDUSA related UML models is investigated.

[4]Note that while most modeling contributions are actually made in terms of UML artifacts, those discipline specific models are not referred to as UML models, because additional contributions, as for instance narrative use case descriptions, may be subsumed. Three discipline specific UML models are defined accordingly, being regarded as fractions of the respective overall, integrated discipline specific models.

[5]As *Architectural Design Modeling* and *Detailed Design Modeling* are regarded to be on the same abstraction level, they share a common *Design Model*.

6.2.2.1 Requirements Modeling Discipline

The *Requirements Modeling Discipline*, whose tasks are jointly performed by a *Requirements Engineer*, is concerned with the elicitation and analysis of the functional and non-functional requirements, as well as with capturing them in a respective *Requirements Model*. As MeDUSA is a use case driven design method, (UML-based) use case modeling is the central task subsumed by this discipline. However, to capture a set of use cases together with their interrelationships in one or more UML use case diagrams is not regarded to be sufficient. Indeed, the details of each use case, i.e. the scenarios covered by it, have to be captured as well. Therefore formulating use case details is the second elementary task defined by this discipline. It can be done by using additional UML behavior diagrams (activity, state machine, or sequence), by providing detailed textual narrative descriptions, as proposed in [WNHL08], or by a combination of both.

Use Case Modeling Eliciting and understanding the requirements of an embedded & real-time software system is the starting point for its development. It is done by specifying all interactions between the software system and its external environment, which is formed by other software or hardware systems[6], as well as by capturing the internally triggered behavior.

According to the current UML specification [OMG07d], which serves as the commonly accepted standard with respect to definition of terms and concepts related to use case modeling, a *use case* is defined to be *"the specification of a set of actions performed by a system, which yields an observable result that is, typically, of value for one or more actors or other stakeholders of the system."*. While use cases have been initially developed to capture the functional requirements of large-scale industrial applications [Jac87] [Jac04], the specification of the software's behavior in terms of actor-system-interactions seems to be promising also in the domain of embedded & real-time systems, where a great fraction of the system behavior is indeed related to the interaction with the external, *embedding* environment.

Contrary to use case modeling of large-scale industrial applications, where it is first and foremost applied to capture only functional requirements, in the context of embedded & real-time systems, especially non-functional timing and concurrency constraints play an outstandingly important role and thus have to be regarded as well. Unfortunately, use case modeling, as currently defined by the UML [OMG07d], does not offer adequate means to specify non-functional requirements.

However, as explicitly and detailedly outlined in [NL07b], modeling of timing and concurrency constraints can be supported by some workarounds. By separating out all triggers of system behaviour into distinct *trigger* actors, as denoted by the *MeDUSA Actor Taxonomy* shown in Figure 6.6, and by thus separating them from the mere pas-

[6] In case of an embedded software system, human users do normally not occur as direct communication partners. Indeed they communicate with the system indirectly via related hardware or software systems in its environment.

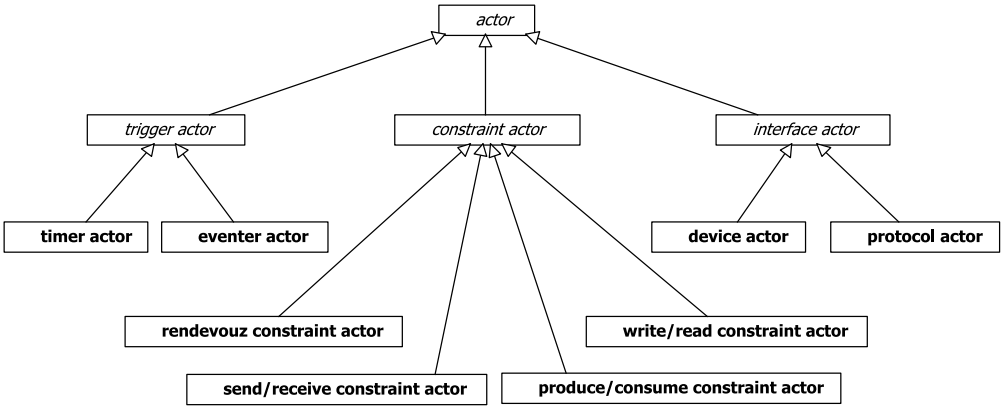


Figure 6.6: The MeDUSA Actor Taxonomy

sive communication interfaces, which are in turn represented by corresponding *interface* actors, important timing and concurrency constraints can be explicitly addressed. That is, timing constraints like timer periods for periodically occuring triggers, or respective worst-case interarrival times for aperiodic ones, can be directly attributed to the respective trigger actors. Behavior being internally triggered periodically, as it is often found in embedded & real-time systems, can thus be consistently modeled with the help of (internal) timer actors. Furthermore, synchronization respectively interference between concurrently executing use cases can be addressed by modeling (internal) constraint actors. Here, bidirectional communication in the form of a *rendezvous* may be differentiated from unidirectional communication in the form of *send/receive*, *produce/consume*, or *read/write*[7].

It has to be mentioned that the use of internal actors to denote internal triggers or synchronization events is not aligned with the currently documented practice of use case modeling. Indeed, Jacobson and Overgaard ([JCJv92]) state that *"the essential thing is that actors constitute anything that is external to the system we are to develop"* and Cockburn [Coc00] explicitly judges the modeling of internal actors to be *"extremely rare, and usually a mistake"*, as the computer system could thereby be treated as a *"white box"*. However, as above proposed modeling practices only facilitate the use of internal trigger and not interface actors, the danger of modeling the system as a *white box* does not seem to be given to a large extend; of course, as always, the concepts can be misused, but they do not entrap it.

MeDUSA defines the output of the *Use Case Modeling* task to be one or more UML use case diagrams, as exemplarily denoted by Figure 6.7. Additionally, if the software system shows global states, for example due to different operation modes, those global states should be explicitly captured in a *Global System States Diagram*. It is realized in terms of a UML state machine diagram, as exemplarily depicted by Figure 6.8.

[7]The *MeDUSA Actor Taxonomy* was updated after MeDUSA's publication as Second Edition within [NL08] to support a more sophisticated modeling of concurrency constraints. Compare [Rit08] for details.

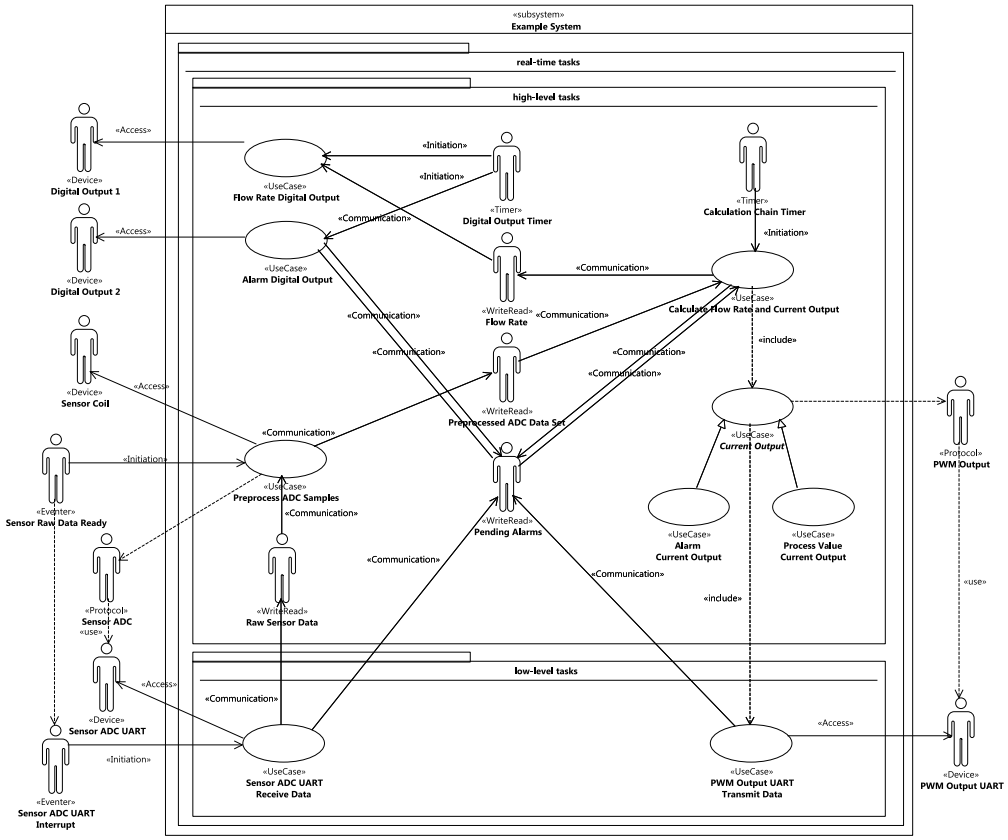


Figure 6.7: MeDUSA Example Use Case Diagram

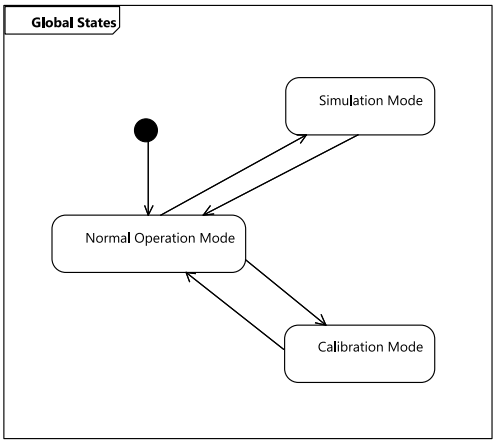


Figure 6.8: MeDUSA Example Global System States Diagram

Use Case Details Modeling A use case represents a variety of different scenarios of interaction between the system and its external environment. Besides a basic behavior, which is understood to be the default scenario, this might - as stated in the UML specification [OMG07d] - *"include possible variations [...], including exceptional behavior and error handling"*. It is thus important to not only describe a use case from a black-box perspective, but to indeed specify its internals in terms of its covered scenarios.

While the UML offers a set of behavioral diagrams to describe the internal details of a use case (activity, state machine, or sequence diagrams), and while a manifold of other notations has been proposed to specify detailed use case descriptions (cf. [HLNW09]), textual narrative descriptions of use case details seem to be widely used and accepted, because of the advantages they offer with respect to understandability. In fact, apart from state-based use case behavior, which probably seems to be best captured by a state-based formalism, describing the details of a use case in a textual narrative form is regarded to be advantageous to other formalisms. This is, because a textual formalism is formal enough to allow validation of consistency and integrity of a narrative description model, while the advantages of natural language with its increased capabilities in terms of readability and understandability are preserved. A *flow-oriented* formalism, as proposed in [WNHL08], furthermore offers the advantage that it nicely supports the intuitive creation of detailed descriptions. That is, starting with a default scenario, which is described completly and explicitly in all detail, the so called *main flow*, further scenarios can be modeled in an incremental manner, by simply describing the differences they show towards the latter.

The work product of the *Use Case Details Modeling* task is thus defined to be *Narrative Use Case Description* for each identified use case, as exemplarily denoted by Figure 6.9.

Use Case `Current Output`

Main Flow	
Start	
1	Alternative Extension Point : *Choose Between simulation and calculation*
2	Specialization Extension Point : *Calculate actual current*
3	Alternative Extension Point : *Current stored*
4	Validate that current does not exceed span limits.
5	Alternative Extension Point : *Current validated*
6	Calculate PWM output signal.
7	Normalize.
8	Include Use Case `PWM Output`.
9	Alternative Extension Point: *End*
End	
Alternative Flow *Simulate Current*	
Start	At *Choose Between simulation and calculation*, if simulation mode has been set
1	Use simulation current as actual value.
End	Continue at *Current stored*
Alternative Flow *Raise "Limits exceeded" alarm.*	
Start	At *Current validated*, if the current exceeds span limits
1	Raise "Limits exceeded" alarm.
End	Continue at *End*

Figure 6.9: MeDUSA Example Use Case Description

Where this seems to be better applicable - a *Use Case Details Diagram* in terms of a respective UML behavior diagram, as it is exemplarily denoted in Figure 6.10 may also be employed.

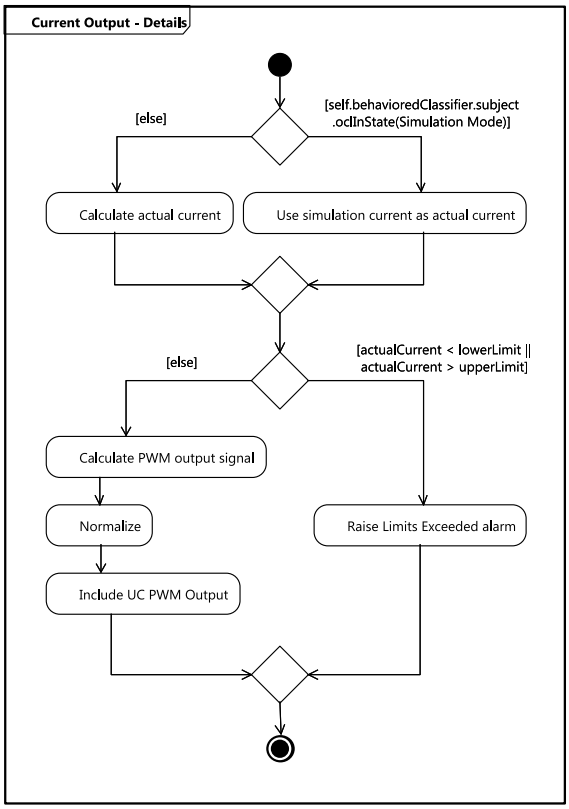


Figure 6.10: MeDUSA Example Use Case Details Diagram

6.2.2.2 Analysis Modeling Discipline

While the *Requirements Modeling Discipline* is concerned with the elicitation and analyis of requirements in terms of use cases, the objective of the *Analysis Modeling Discipline*, whose tasks are jointly performed by a *System Analyst*, is to gather a profound and more detailed understanding of the problem domain. This is done by constructing an *Analysis Model* in terms of *analysis objects*, whose individual and collaborative behavior performs the system behavior as it has been captured in the use cases, which were identified during *Requirements Modeling*. The construction of an *Analysis Model* can conceptually be broken down into three main objectives, namely:

- identification of the analysis objects needed to perform the system's behavior
- capturing the collaborative, inter-object behavior of the identified objects
- capturing the individual, intra-object behavior of each identified object

The identification and classification of analysis objects is guided by the respective *MeDUSA Object Taxonomy*, similar to as it is done for actors by means of the *MeDUSA Actor Taxonomy*. It is denoted in Figure 6.11 and proposes a disjoint categorization of analysis objects into *trigger*, *interface*, *control*, *entity*, and *application-logic* objects.

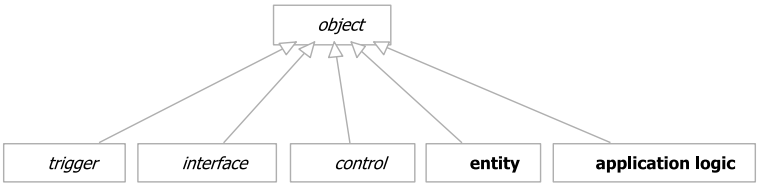


Figure 6.11: The MeDUSA Object Taxonomy

Trigger and interface objects are the system-internal representations of (internal and external) event sources, as well as external interfaces, being already captured in the requirements model, so that they can be easily transferred from respective trigger and interface actors. Entity objects represent long-living data the software has to be kept track of, while control and application-logic objects encapsulate behavioral aspects of the software, which is a state-dependent or coordinative control-flow in case of the first, as well as certain business-logic or a specific algorithm in case of the latter. Their identification is a more creative process as there are no direct correspondences in the *Requirements Model*.

The identification of the different object types is subsequently performed. That is first the identification of trigger and interface, as well as entity objects is done independently in terms of *Context Modeling* and *Information Modeling*. Successively the identification of the remaining control and application-logic objects is performed in terms of *Inter-Object Collaboration Modeling*, which is concerned with identification of an object collaboration for each identified use case. Last, the internal behavior of each identified object is captured in terms of *Intra-Object Behavior Modeling*.

Context Modeling *Context Modeling* is done to identify those analysis objects that encapsulate triggers of system behavior, as well as those realizing interfaces to the external hardware devices and software systems. As the external environment of the software system has already been captured in terms of respective trigger and interface actors in the *Requirements Model*, the identification of trigger and interface objects is usually pretty much straight-forward.

(a) MeDUSA Trigger Object Taxonomy (b) MeDUSA Interface Object Taxonomy

Figure 6.12: The MeDUSA Trigger & Interface Object Taxonomies

However, it has to be pointed out that the identified trigger and interface objects do not represent the event sources or external interfaces themselves, as the actors did, but indeed encapsulate their software-internal respresentations, so that a one-to-one mapping does not necessarily have to be the case. Indeed, a single trigger actor may lead to multiple internal trigger objects to denote different logical event sources, while multiple interface actors may indeed be represented internally by a single interface object.

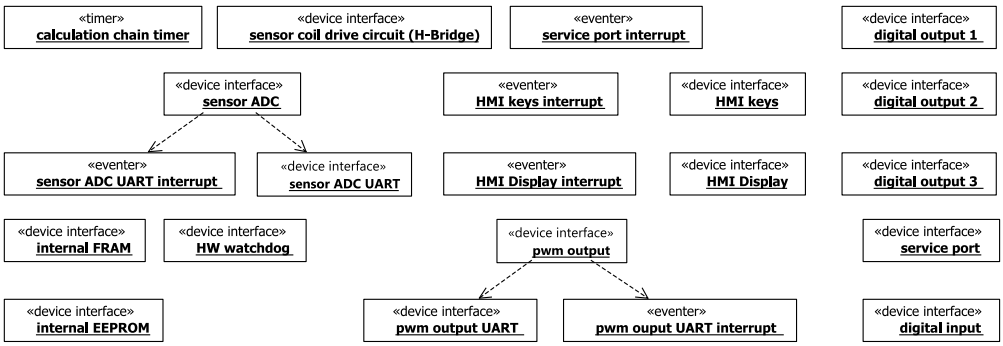


Figure 6.13: MeDUSA Example Context Diagram

As exemplarily shown in Figure 6.13, the outcome of the *Context Modeling* task is defined as a UML object diagram, which denotes all relevant timer and interface objects, being referred to as the *Context Diagram*.

Information Modeling *Information Modeling* is performed to identify data intensive (analysis) objects, so called *entity* objects. Entity objects are most likely accessed by several use cases, and thus represent the persistent, i.e. long-lasting data, the software has to keep track of. While embedded & real-time software is normally not expected to be very data intensive, at least if being compared to an information system, quite a number of entity objects might actually be involved. In case of the measurement example device, being quoted in Section 4.1.1, a significant amount of data is for example related to configuration and monitoring of the device, keeping track of everything the user is able to configure or check.

While the use cases may implicitly deliver valuable input for the identification of entity objects as well, those objects cannot be identified as easily and straight-forward as interface or trigger objects, as they are most often not explicitly denoted within the detailed description of a use case. Often, they are indeed only inferrable implicitly, for example where they are mentioned to be exchanged between the system and its related actors, or where their values have some impact on the course of events taken during a system-actor interaction. It thus has to be pointed out that *Information Modeling* is a quite creative task that probably has to be performed in several iterations, before a consistent set of entity objects has been acquired.

MeDUSA defines the outcome of the *Information Modeling* task to be one or more *Information Diagrams* in the form of UML object diagrams to denote all entity objects together with their respective relationships. An example for such a diagram is denoted by Figure 6.14, which depicts an *Information Diagram*, developed for the example system, pointed out in Section 4.1.1.

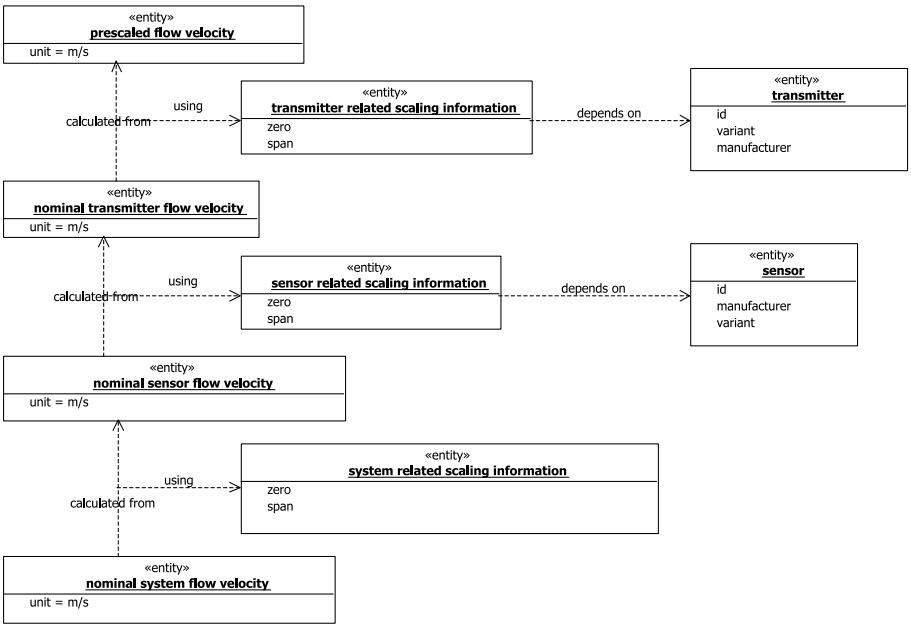


Figure 6.14: MeDUSA Example Information Diagram

Inter-Object Collaboration Modeling Having already inferred trigger and interface as well as entity objects during *Context Modeling* and *Information Modeling*, the remaining *control* and *application-logic* objects now have to be successively identified in order to gain a complete and consistent *Analysis Model*. *Application-logic objects* encapsulate arbitrary functionality related to the given application domain, for example an algorithm or a piece of business logic, which usually accesses more than one entity object and is therefore, or because it might likely change, encapsulated into an individual object. *Control objects* in turn encapsulate mere control logic. As denoted by Figure 6.15, they can be further classified into *state-dependent control objects* and non state-dependent *coordinator objects*, dependent on their behavioral characteristics.

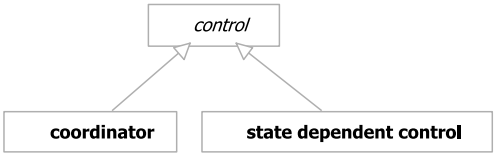


Figure 6.15: The MeDUSA Control Object Taxonomy

While *Context Modeling* and *Information Modeling* support the identification process by analyzing the requirements model from a rather static viewpoint (i.e. not the detailed flow of events subsumed by the use cases, but the *static* information captured within it, is of main interest), identification of the remaining control and application-logic objects is performed by taking a rather dynamic viewpoint. That is, the detailed flow of events of each use case is analyzed and transferred into message-based communication of an object collaboration. The identification of those objects, participating in the respective object collaboration, is thus implicitly performed, where the need for a collaboration participant is recognized.

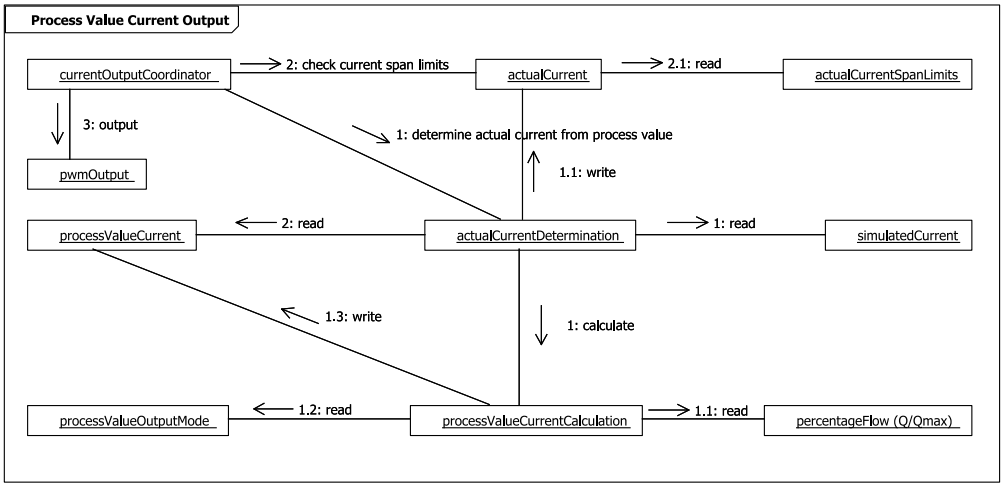


Figure 6.16: MeDUSA Example Inter-Object Collaboration Diagram (Communication)

The outcome of the *Inter-Object Collaboration Modeling* is a number of *Inter-Object Collaboration Diagrams*, depicting the different object collaborations. These diagrams may be developed as UML sequence or communication diagrams, as depicted by Figures 6.17 and 6.16, dependent on which formalism is best suited.

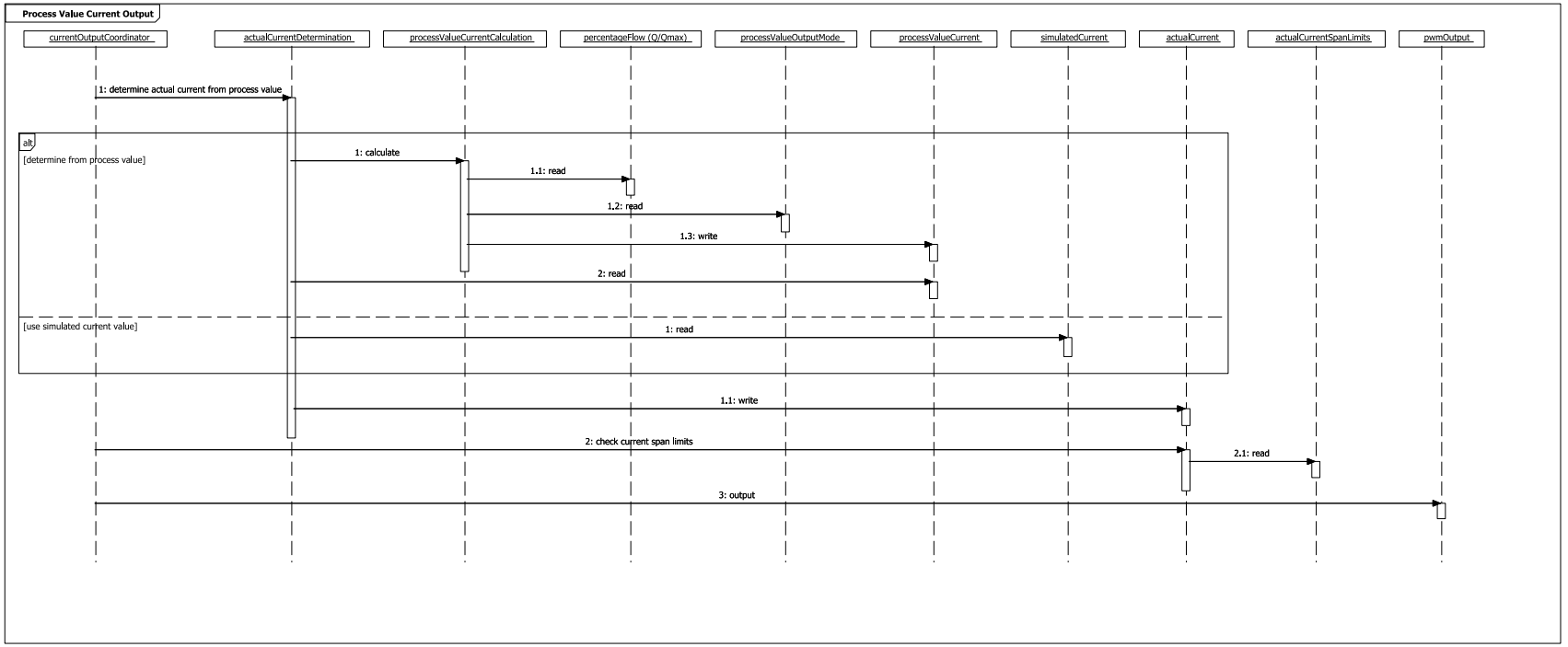


Figure 6.17: MeDUSA Example Inter-Object Collaboration Diagram (Sequence)

Intra-Object Behavior Modeling *Intra-Object Behavior Modeling* is the conceptually last task covered by the *Analysis Modeling* discipline. While *Inter-Object Collaboration Modeling* captures the behavior of an object only with respect to what is unveiled in the context of a respective object collaboration, *Intra-Object Behavior Modeling* focusses on synthesizing the individual behavioral fragments, an object shows in the different collaborations, it participates in, into an overall integrated intra-object behavior model.

Intra-Object Behavior Modeling is in particular useful for *control objects*, showing state-dependent behavior. Here, just by being inferred implicitly from the partial behavior, the object unveils in the different collaborations, the overall object behavior may not be obvious and apparent. To faciliate an easier accessibility, synthesizing of the individual behavioral fragments into an overall integrated state machine model is necessary. Indeed, in extremes, it might not be clear before such a synthesizing that the partial behavioral fragments of a respective control object were indeed modeled to be inconsistent with each other, so that *Intra-Object Behavior Modeling* helps to identify such problems at an early stage.

Besides modeling intra-object behavior of *state-dependent control objects*, which is regarded to be a mandatory part of *Intra-Object Behavior Modeling*, it might be helpful to explicitly capture the behavior of other non-trivial objects as well, possibly using different behavioral formalisms offered by the UML. It might thus for example be an option to explicitly capture an algorithm, encapsulated by an *application-logic object*, with the help of a UML activity diagram, or to document the behavior of a non state-dependent *coordinator object* with the help of a UML sequence diagram.

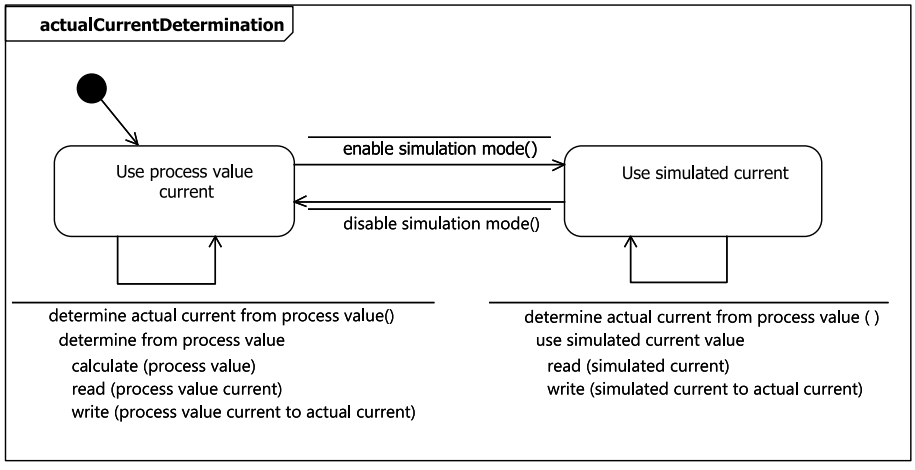


Figure 6.18: MeDUSA Example Intra-Object Behavior Diagram

Hence, the work products produced by the *Intra-Object Behavior Modeling* task are defined to be one or more *Intra-Object Behavior Diagrams* in the form of a UML state machine (as exemplarily shown in Figure 6.18), sequence or activity diagram. For *state-dependent control objects* the development of a *Intra-Object Behavior Diagram* in the form of a UML state machine diagram is regarded to be mandatory due to above

mentioned reasons, while in case of other analysis objects the effort of creating such a diagram has to be individually weighted up against the benefits it provides.

6.2.2.3 Architectural Design Modeling Discipline

As indicated by its denomination, the *Architectural Design Modeling Discipline*, which tasks are jointly performed by a *System Architect*, is basically concerned with the definition of the software architecture. According to Bass, Clements and Kazman, a *software architecture* may be defined as *"the structure or structures of the system, which comprise software elements, the externally visible properties of those elements, and the relationships among them"* [BCK03]. While this definition is regarded to be quite appropriate and sound, it has the slight disadvantage that - even it this is indeed subsumed - it is not emphasized that the relationships between the respective software elements can not only be defined from a structural perspective, but indeed also from a behavioral one. Taking this into consideration and additionally that in the context of MeDUSA, the respective software elements on the architectural level are denoted as *subsystems*, the objective of *Architectural Design Modeling Discipline* can be formulated as defining the software architecture in terms of:

- identifying subsystems and characterizing their externally visible interfaces
- identifying the structural relationships between the identified subsystems
- identifying the behavioral relationships between the identified subsystems

The first subgoal, namely the identification of subsystems and their externally visible required and provided interfaces, is conceptually broken down into two closely related tasks, namely *Subsystem Identification* and *Subsystem Consolidation*, which in practice may be decided to be performed in one integrated step[8]. *Subsystem Identification*, the conceptually first task, is concerned with identifying subsystems and their (preliminary) interfaces by grouping together analysis objects, thus partitioning the *Analysis Model*. *Subsystem Consolidation* successively deals with consolidating the identified subsystems. That is, the analysis objects grouped into the subsystems and the subsystems' provided and required interfaces, which were initially merely inferred from the behavioral relationships between the allocated analysis objects, have to be consolidated under design considerations. Having performed this, and thus having gained a consistent partitioning of the software into well defined and strongly encapsulated subsystems with properly defined required and provided interfaces, the integration of the subsystems can be defined from a structural and behavioral perspective. This is covered by the *Structural System Architecture Modeling* and *Behavioral System Architecture Modeling* respectively.

[8] The work products produced by the *Subsystem Identification* task are regarded as intermediate work products, which have to be refined and detailed by the successively performed *Subsystem Consolidation* task, so merging the two conceptually separated tasks together is a legal option for an experienced *System Architect*.

Different to the *Requirements Modeling* and *Analysis Modeling Disciplines*, there is no distinct *Architectural Design Model* to be jointly created by the tasks of the *Architectural Design Modeling* discipline. Indeed, the respective *Design Model* is shared between the *Architectural Design Modeling* and *Detailed Design Modeling Disciplines*, as both reside on the same abstraction level. The *Design Model* is further outstanding because it may - different to the other modeling discipline specific models - be realized as a distributed model, being split into several model fragments (one for each subsystem as well as one for the system-wide artifacts, integrating the subsystem specific model fragments as some kind of *umbrella* model), to allow the distributed development of the detailed design for each subsystem.

Subsystem Identification Identifying subsystems, the central building blocks of the software architecture, and their externally visible interfaces is the first essential objective covered by the *Architectural Design Modeling Discipline*. As this objective is rather complex it is conceptually broken down into two tasks, out of which *Subsystem Identification* is concerned with the identification of subsystems and their preliminary interfaces based on the information captured in the *Analysis Model*. According to Jacobson, Christerson, Jonsson, and Övergaard, who denote the *"task of subsystems"* to be *"to package objects in order to reduce the complexity"* [JCJv92], identification of subsystems is performed by grouping together analysis objects and thus dividing the *Analysis Model*.

Obviously, the identification of subsystems is no straightforward task but a rather creative and inventive one, which has to combine conflicting objectives, out of which - in the context of embedded & real-time software - the two most outstanding are maintainability and performance. In terms of maintainability, *locality in changes* has been named by Jacobson et. al. [JCJv92] as the most important design principle that has to be regarded during the division of objects into subsystems. It refers to the principle of *"predicting what the system changes will look like, and then making the division on the basis of this assumption"*. Emphasizing the performance aspect, *task coupling* has been named in [NL08] as a second design principle of comparably outstanding importance, whose essence is to keep the number of tasks spanning multiple subsystems as low as possible to reduce synchronization overhead and thus improve the system performance. Other principles like *functional coupling* (cf. [JCJv92]) or *reusability* (cf. [NL08]) may additionally be named in this context.

Having divided of analysis objects into subsystems, the externally visible interfaces of each subsystem have to be successively retrieved by investigating the inter-object communication captured in the *Analysis Model*. That is, all messages, being exchanged between the objects, being composed by a respective subsystem, and those objects, being composed by other subsystems, have to be transferred into operations of respective required or provided interfaces of that subsystem. Those interfaces subsequently have to be aggregated to ports, which represent the single interaction points of the subsystem. It has to be pointed out that - in order to obtain fully self-encapsulated subsystems - the required and provided interfaces should indeed be modeled individually for each subsystem and interfaces should not be shared. However, as the subsystems will have

to be integrated with each other via their interfaces during *Structural System Architecture Modeling*, they should already be designed in such a way, that communicating subsystems offer respective compatible required and provided interfaces. Due to this, a straightforward and simple approach to modeling the required and provided interfaces of a subsystem is to design a set of required and provided interfaces for each external subsystem, the respective subsystem has to communicate with.

The identification process is supported and its results are documented by several work products that describe the identified subsystems from both, a structural as well as a behavioral perspective. The structural viewpoint is described by an *Initial Structural Subsystem Design Diagram* and an *Initial Structural Subsystem Interface Design Diagram*. The first, a UML composite structure diagram, shows the (analysis) objects (in terms of parts) and ports composed by the subsystem, as exemplarily shown by Figure 6.19.

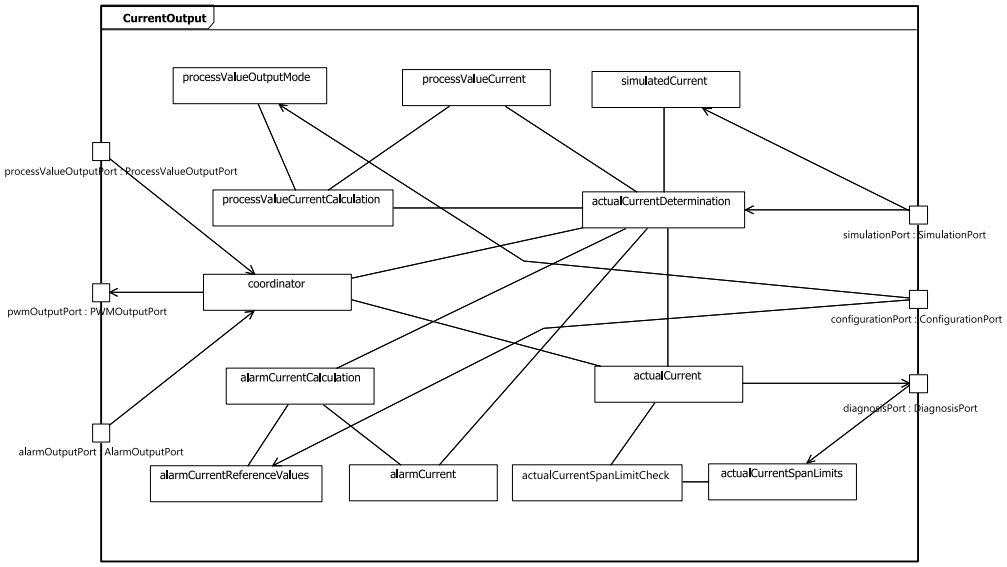


Figure 6.19: MeDUSA Example Initial Structural Subsystem Design Diagram

The second, a UML class diagram, denotes the signatures of the required and provided interfaces and their aggregation to the ports via respective usage or realization relationships to the respective port's type. It is exemplarily shown by Figure 6.20.

The behavioral aspects inherent to a subsystem are in turn captured by one or more *Initial Behavioral Subsystem Design Diagram(s)*, as well as one or more *Initial Behavioral Subsystem Interface Design Diagram(s)*. The *Initial Behavioral Subsystem Interface Design Diagram(s)* capture the externally visible behavior of the subsystem, offered via its ports. They are developed as UML (protocol) state machine diagrams either to specify the communication capabilities of the subsystem as a whole, as exemplarily shown in Figure 6.21, for a single port, or even for a single interface.

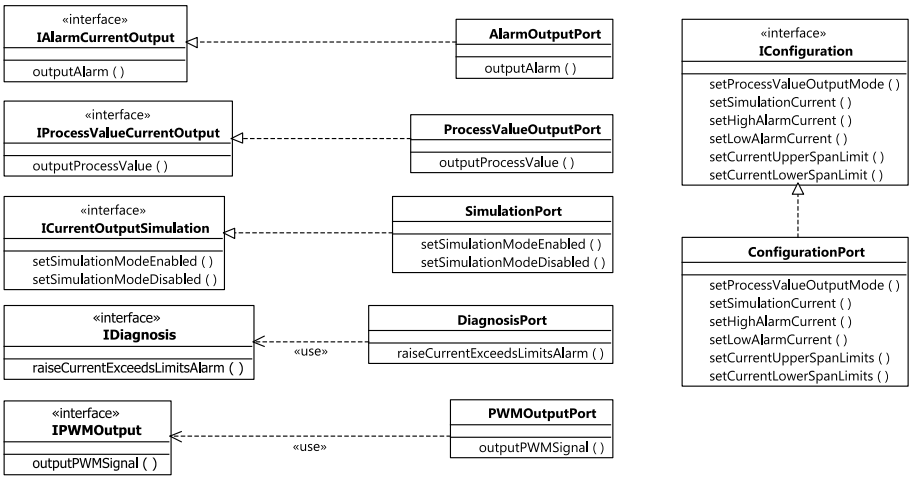


Figure 6.20: MeDUSA Example Initial Structural Subsystem Interface Design Diagram

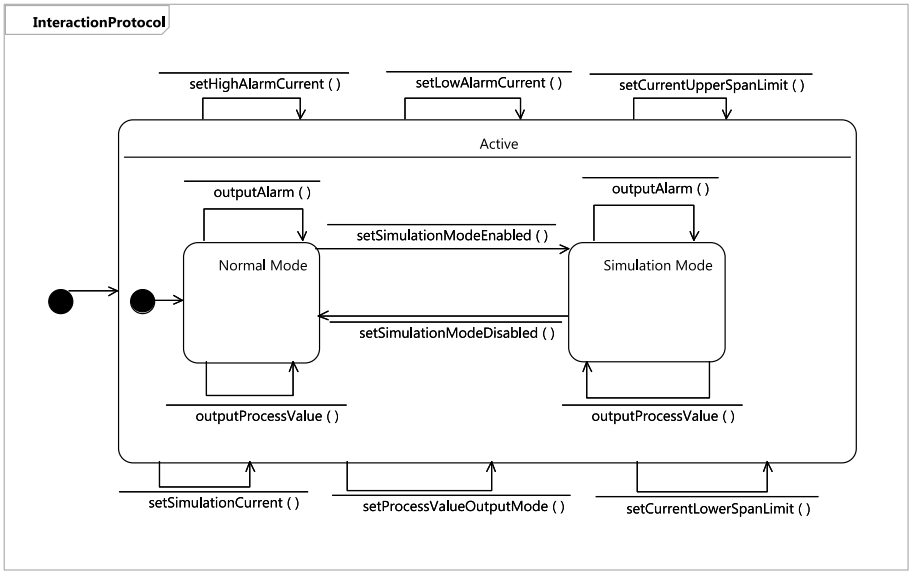


Figure 6.21: MeDUSA Example Initial Behavioral Subsystem Interface Design Diagram

The *Initial Behavioral Subsystem Design Diagrams*, as exemplarily denoted by Figure 6.22, developed in the form of UML sequence diagrams, in turn capture the internal behavior of the subsystem, that is they reflect how externally (via ports) or internally (via trigger objects) stimulated behavior manifests itself in terms of internal message communication amongst those parts and ports, composed by the subsystem.

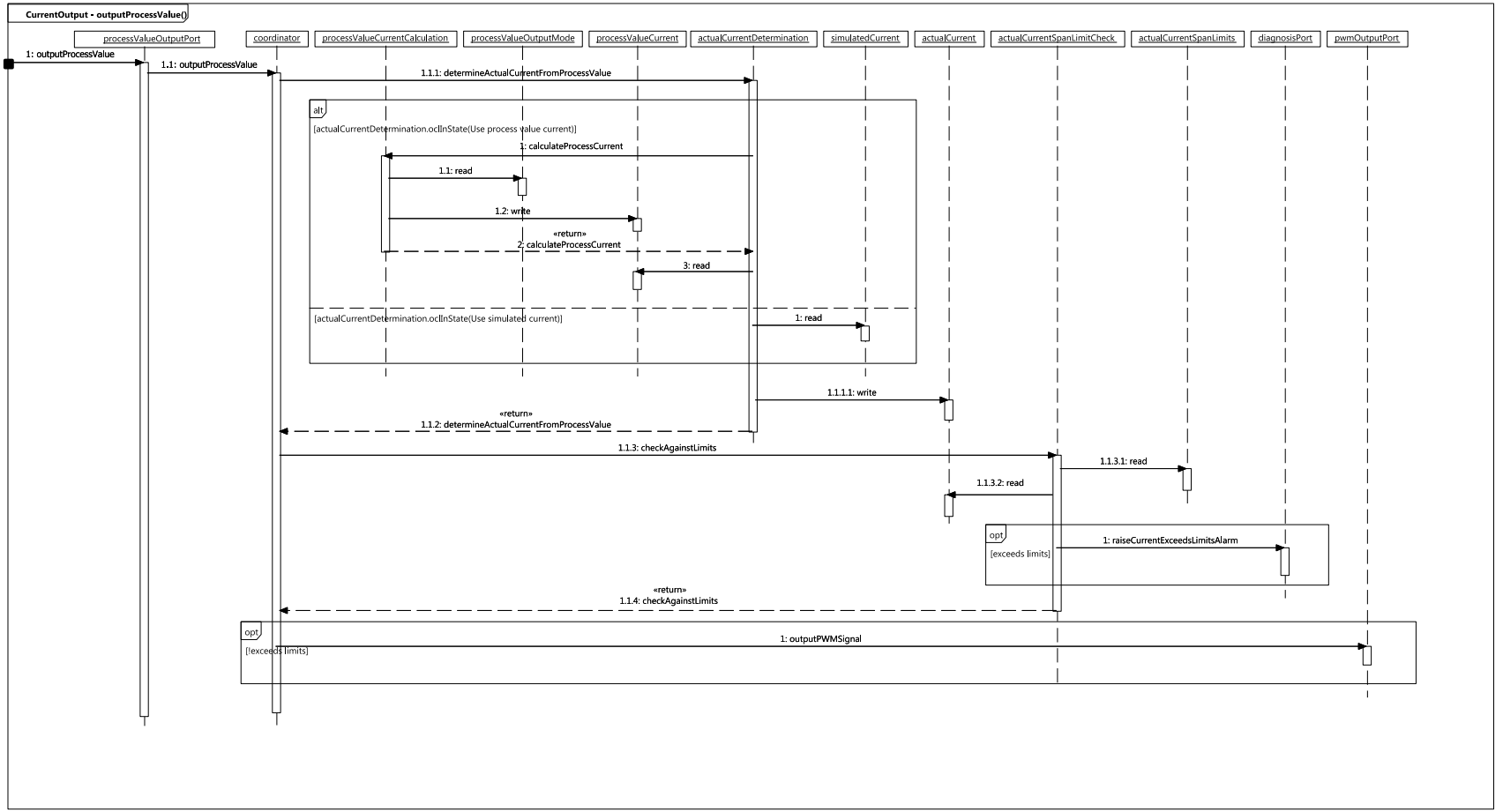


Figure 6.22: MeDUSA Example Initial Behavioral Subsystem Design Diagram

Subsystem Consolidation While *Subsystem Identification* is merely concerned with the division of (analysis) objects into subsystems, *Subsystem Consolidation* now explicitly addresses the consolidation of the initially identified subsystems under design considerations, affecting both, the internal decomposition as well as the externally visible interfaces, which has so far been merely inferred from the inter-object communication of the *Analysis Model*.

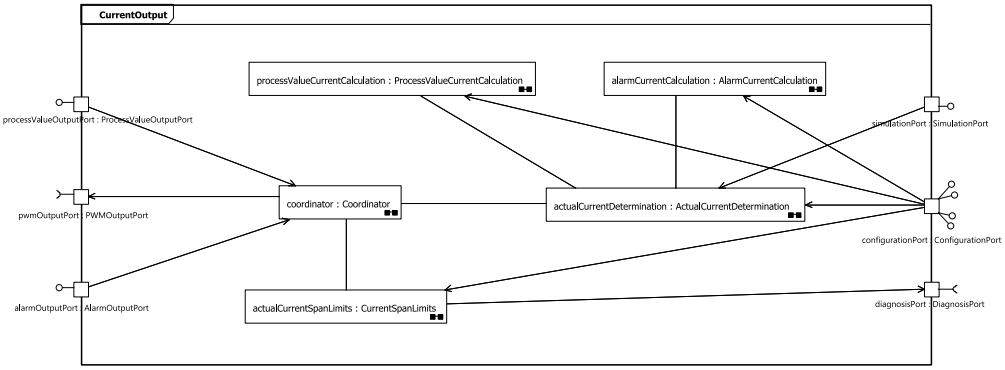


Figure 6.23: MeDUSA Example Consolidated Structural Subsystem Design Diagram

That is, the analysis objects partitioned into the identified subsystems now have to be transferred into design objects, which should have a strong internal cohesion and only loose coupling towards other design objects, what is basically achieved by splitting apart or by merging analysis objects.

The resulting structural decomposition of the subsystem is documented by a revision of the *Initial Structural Subsystem Design Diagram*, which was developed during *Subsystem Identification* for each subsystem, being referred to as a *Consolidated Structural Subsystem Design Diagram*. An example is depicted by Figure 6.23.

Further, from a behavioral viewpoint, the subsystem-internal communication between the resulting design objects, which was initially inferred from the inter-object behavior, has to be updated accordingly. Further detail also has to be added in terms of all parameters that have to be exchanged with the messages. Revisions of the *Initial Behavioral Subsystem Design Diagrams*, accordingly referred to as *Consolidated Behavioral Subsystem Design Diagrams*, as exemplarily denoted by Figure 6.24, are developed for this purpose.

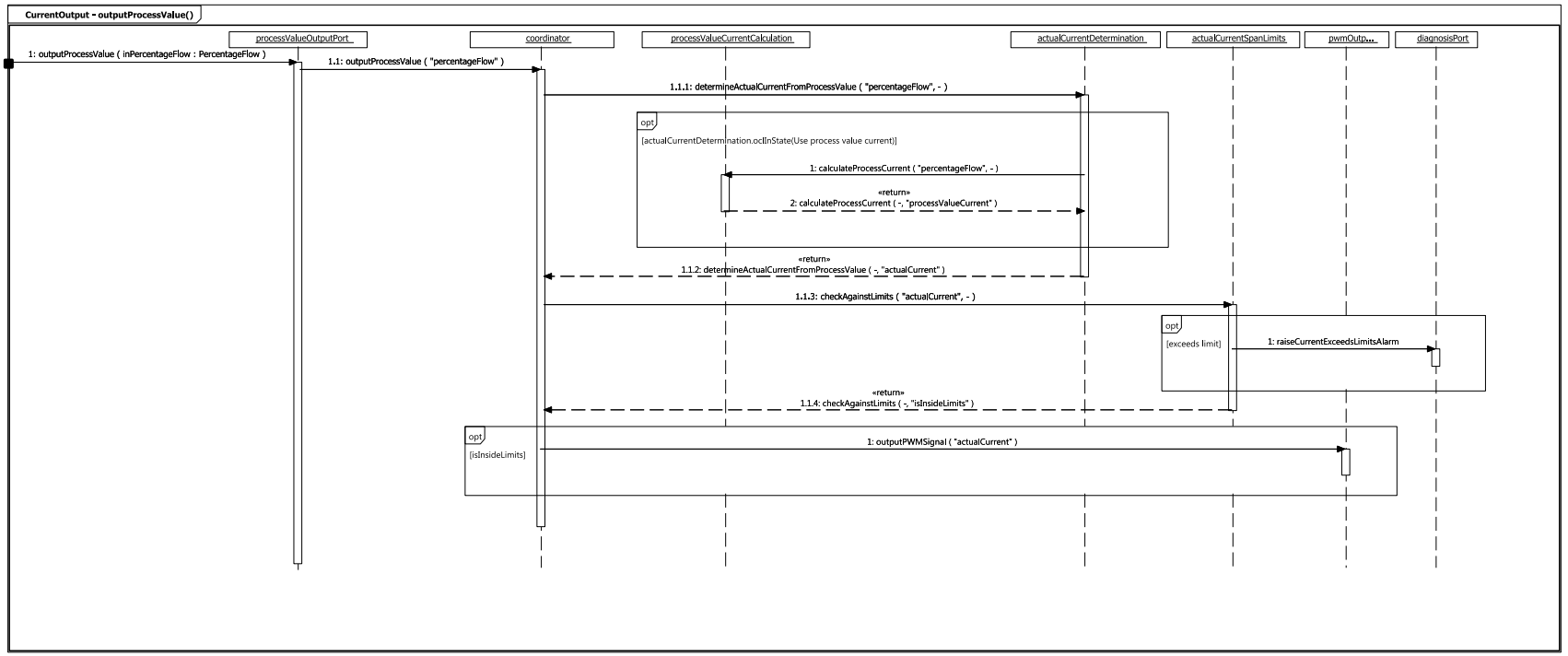


Figure 6.24: MeDUSA Example Consolidated Behavioral Subsystem Design Diagram

What has to be done for the internal subsystem decomposition also holds for the externally visible interfaces of each subsystem. Here, the allocation of operations to respective interfaces, as well as that of interfaces to respective ports has to be reconsidered, first to reflect the changes imposed by the consolidation of the subsystem decomposition, second due to considerations related to the integration capabilities of the subsystem. It may for example be reasonable to split an interface into several smaller ones in order to separate out several distinct aspects, resulting in a better integrateability and an enhanced reusability of the subsystem. It may also be reasonable to design an interface so that it best matches a respective matching interface of an already existing subsystem that has to be reused. Additionally to this restructuring, all interfaces have to be enriched with supplementary detail. That is, the signatures of all interface operations have to be fully specified in terms of their parameters and parameter types, whose detailed design has to be evolved as well.

As in case of the internal subsystem decomposition, updated versions of the *Initial Structural Subsystem Interface Design Diagrams* and *Initial Behavioral Subsystem Interface Design Diagrams* are developed, which are accordingly referred to as *Consolidated Structural Subsystem Interface Design Diagrams* and *Consolidated Behavioral Subsystem Interface Design Diagrams*. Examples for both diagrams are provided in Figures 6.25 and 6.26 respectively.

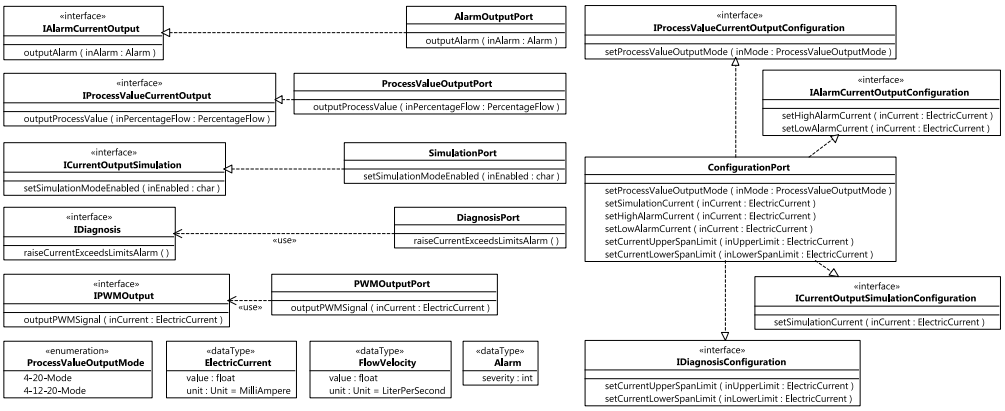


Figure 6.25: MeDUSA Example Consolidated Structural Subsystem Interface Design Diagram

The conclusive outcome of the *Subsystem Consolidation* task may thus be understood as a set of fully encapsulated subsystems with only explicit context dependencies (in terms of provided and required interfaces, exposed via ports) and well designed internal decompositions. Due to the conceptually very close relationship between *Subsystem Identification* and *Subsystem Consolidation*, an experienced *System Architect* may even decide to perform both tasks in a single step, thus eliding the initial diagrams, being defined as outputs of the *Subsystem Identification*, and directly evolving respective consolidated diagrams. The contribution to the underlying *Design Model* will be identical, as the initial diagrams are sort of intermediate work products.

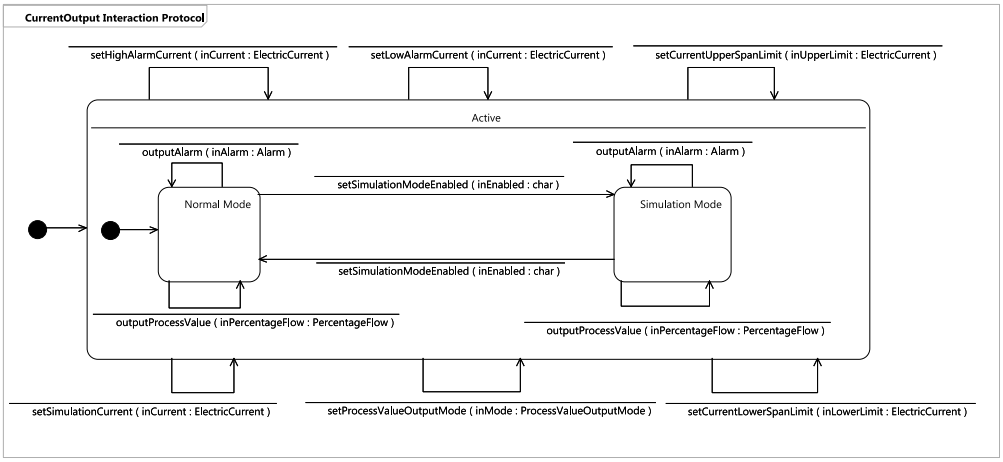


Figure 6.26: MeDUSA Example Consolidated Behavioral Subsystem Interface Design Diagram

Structural System Architecture Modeling Having defined the individual subsystems in terms of their composed design objects and their externally visible interfaces, the next step is to structurally integrate those subsystems via their exposed interfaces to an overall software system. That is, subsystems have to be assembled in such a way that each required interface, exposed by a subsystem, is *served* by a respective provided interface of another. In detail, *wiring* is performed by designing an *assembly connector* between those ports, exposing (signature) compatible required and provided interfaces. By this, it is ensured that each context dependency, a subsystem discloses via a required interface, is indeed fulfilled by another subsystem via a respective provided interface, so that the all subsystems structurally fit together and thus form an overall integrated system. To this extend, *Structural System Architectural Modeling* may be understood to be more concerned with *proofing of concept*, rather than actual creative modeling.

The work product produced by the *Structural System Architecture Modeling* is a UML composite structure diagram representing the overall system as an enclosing component, whose internal structure is manifested in the form of composed parts, representing the aggregated subsystem (component) instances. It is denoted as the *Structural System Architecture Diagram* and it is recommended, as demonstrated in Figure 6.27, to employ the notation of *ball-and-socket* assembly connectors, thereby explicitly denoting the respective interfaces forming the basis for each wiring.

Behavioral System Architecture Modeling While *Structural System Architecture Modeling* specifies the structural aspects of the system architecture, thus validating that the subsystems fit together from a structural viewpoint, *Behavioral System Architecture Modeling* aims at defining how the subsystems are integrated with each other from a behavioral perspective, thus validating that the subsystems are also well designed to this extent and that they can thus be well integrated to an overall system behavior.

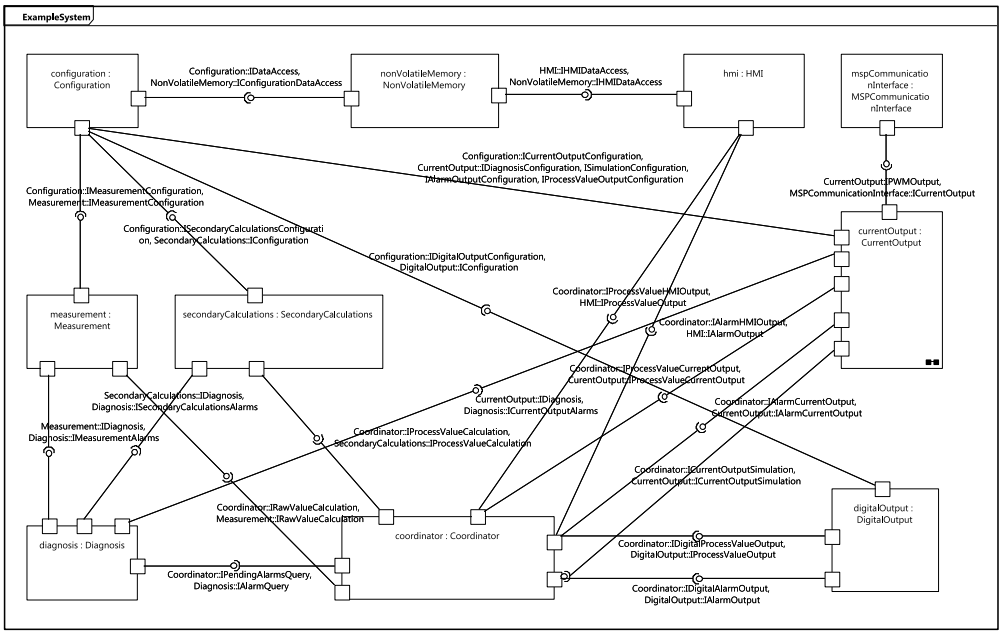


Figure 6.27: MeDUSA Example Structural System Architecture Diagram

That is, all system-wide behavioral threads, i.e. those affecting several subsystems, have to be mapped to the underlying structural system architecture, by specifying how they are manifested in terms of operation calls, being exchanged between the participating subsystems via their respective ports. To this extend, the *Behavioral System Architecture Modeling* task may be regarded to be more of a validation rather than construction nature. However, the specification of how the system-wide communication is established between the identified subsystems is of valuable input to any *Subsystem Designer* and *Subsystem Implementer*, when creating the detailed class design for a subsystem and when successively implementing these respective classes, as well as for the *Real-Time Analyst* when performing an analysis.

The work products produced during *Behavioral System Architecture Modeling* are *Behavioral System Architecture Diagram(s)*, as exemplarily depicted by Figure 6.28. They are developed in the form of UML sequence diagrams, which depict the involved subsystem instances (and optionally their composed ports) and capture the inter-subsystem communication in terms of messages, which are exchanged via their respective ports.

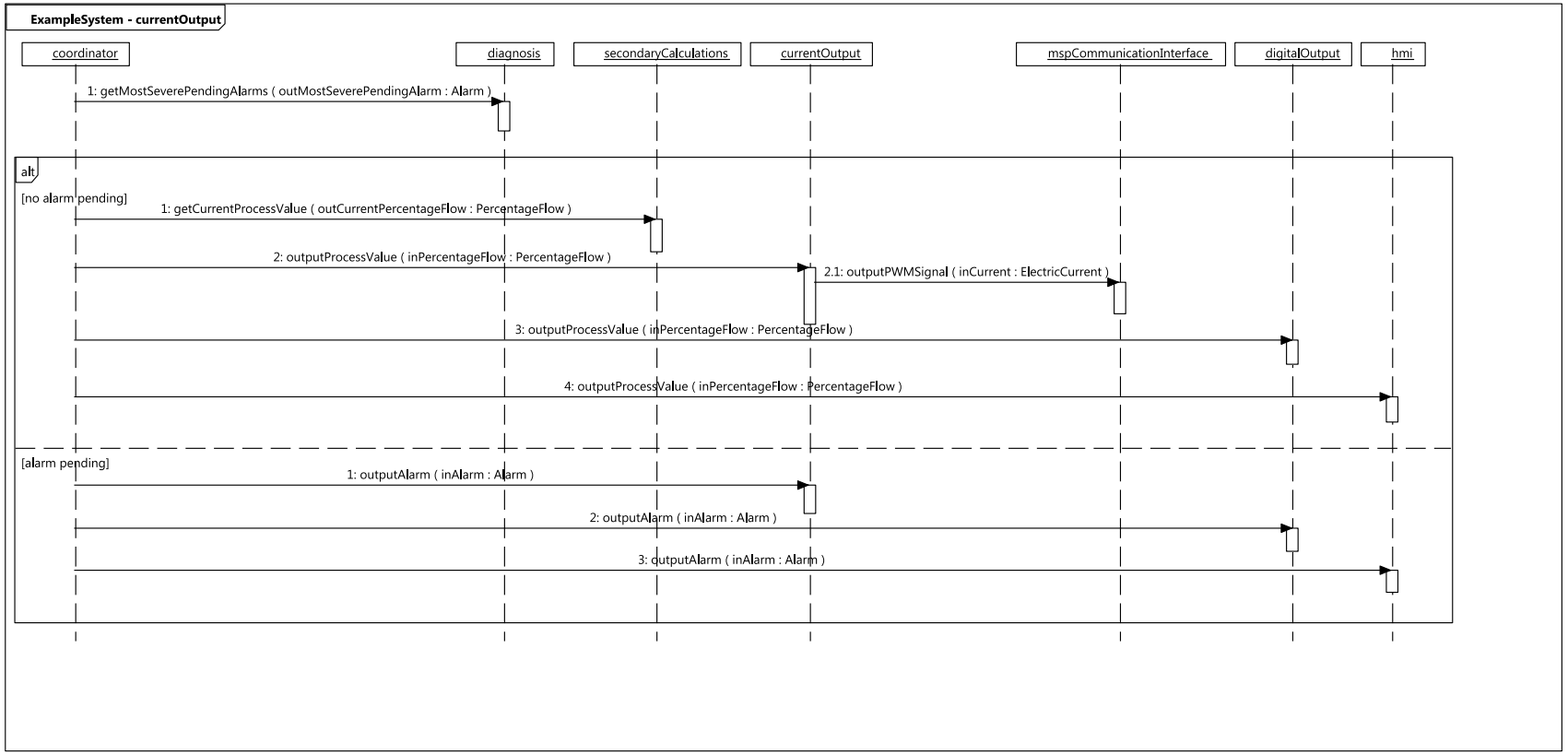


Figure 6.28: MeDUSA Example Behavioral System Architecture Diagram

6.2.2.4 Detailed Design Modeling Discipline

Based on the thoroughly defined software architecture, which is specified in terms of a set of self-encapsulated, collaborating subsystems, the *Detailed Design Modeling Discipline* is concerned with the development of a detailed design for the internal decomposition of each subsystem. That is, while the detailed design of all exposed interfaces, of all related ports' types, and of all (data) types, being used as parameter types in the signatures of the exposed interfaces' operations, has already been developed during *Subsystem Consolidation*, the detailed class design for those design objects, forming the internal decomposition of a respective subsystem, now has to be developed. Following Jacobson et. al., who state that *"subsystems may also be used as handling units in the organization"* [JCJv92], *Detailed Design Modeling* is performed individually for each subsystem by a respective *Subsystem Designer*.

Detailed Structural Design Modeling While the detailed design for all classes and data types related to the externally visible interfaces of the subsystems has already been developed during *Architectural Design Modeling*, *Detailed Structural Design Modeling* is concerned with developing detailed classes for all design objects being internally composed by a respective subsystem.

In detail, for each part belonging to a subsystem's internal decomposition, a class has to be developed (to type the part), and associations have to be designed to type all connectors, composed by the subsystem. That is, for every assembly connector, being established between two parts, a respective association has to be designed between those classes, being used as types for the respective parts, and for every delegation connector, such an association has to be designed accordingly between the types of the related part and port.

The operations of each class have to be inferred from the messages, being specified by the *Consolidated Behavioral Subsystem Design Diagrams*, which has been developed during *Subsystem Consolidation*. Additional input may be taken from those *Intra-Object Behavior Diagrams*, being developed during *Intra-Object Behavior Modeling*, which may in turn also lead to a set of attributes (e.g. to reflect the object's state, in case a state machine was used to specify intra-object behavior). The same holds for those classes, being designed as types for the subsystem's ports. Here, in case a protocol state machine was developed during *Subsystem Consolidation*, attributes and operations may be identified, additional to the operations that were already derived from the port's required and provided interfaces. In case of entity objects, further attributes may also be derived from the slots that were captured in the *System Information Diagram(s)*.

The work product produced by the *Detailed Structural Design Modeling* is a so called *Structural Detailed Design Diagram*, which is developed for each subsystem in the form of a UML class diagram, as outlined by Figure 6.29. It denotes the classes, which have been designed as types for the parts and ports composed by the subsystem, as well as associations between those classes, according to the connectors, being composed.

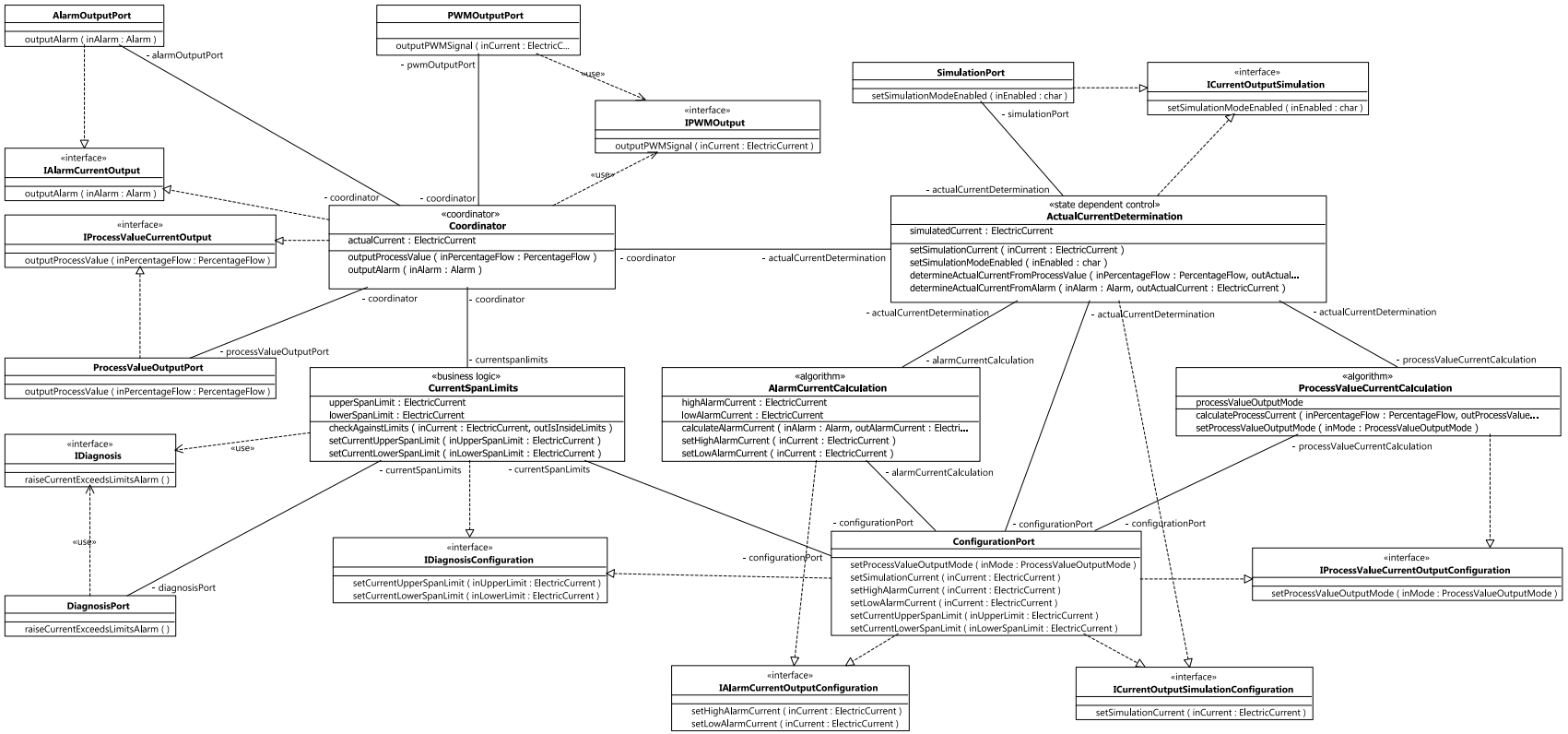


Figure 6.29: MeDUSA Example Structural Detailed Design Diagram

Detailed Behavioral Design Modeling Based on the internal object behavior that has been initially captured in the *Intra-Object Behavior Diagrams*, the internal behavior of the design objects now has to be consistently modeled. That is, the behavioral specification captured in the *Intra-Object Behavior Diagrams* has to be updated to reflect the changes made to the internal subsystem decomposition during *Subsystem Consolidation* (i.e. if design objects were merged together or split apart). Further it has to be enriched with additional detail, and it has to be integrated with the structural specification developed during *Detailed Structural Design Modeling*.

Behavioral Detailed Design Diagrams are developed for this purpose, in the form of a UML state machine diagram to depict the overall object behavior, as denoted exemplarily by Figure 6.30, or in the form of activity diagrams for individual behavioral aspects (i.e. operations), dependent on which granularity and which behavioral formalism is best suited.

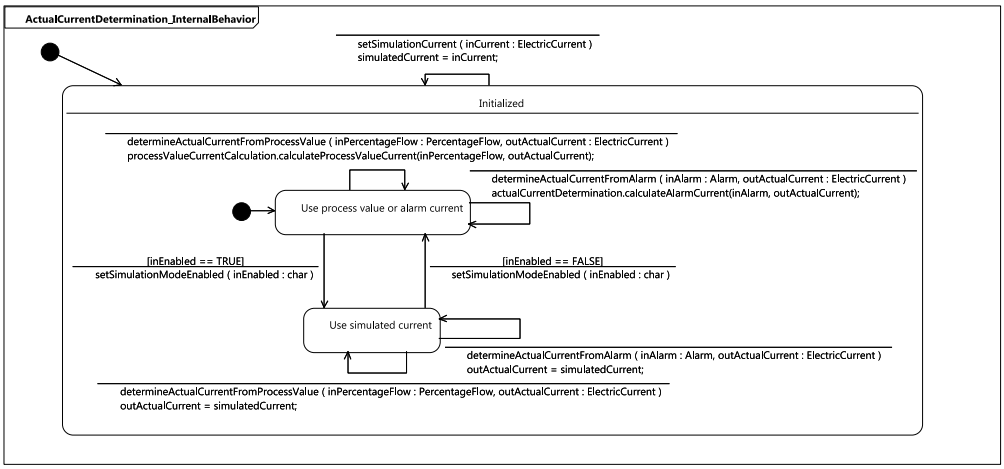


Figure 6.30: MeDUSA Example Behavioral Detailed Design Diagram (State Machine)

It has to be pointed out that, because modeling those diagrams requires some effort, it is of course only reasonable for those objects, whose internal behavior is neither trivial nor obviously clear, and can further not be inferred from the *Consolidated Behavioral Subsystem Design Diagrams*. Therefore, respective diagrams will most likely only be developed for *state-dependent control* and *application-logic* objects with non-trivial behavior.

6.2.2.5 Implementation Discipline

Regarding the covered life cycle phase, the *Implementation* discipline is the conceptually last constructive discipline covered by MeDUSA. It is concerned with transferring the software design, as developed by the tasks of the *Architectural Design Modeling* and *Detailed Design Modeling Disciplines*, into source code and enriching this source code to a complete and valid code base. Due to this, the *Implementation Discipline* is

conceptually split into two parts, namely the mere transformation of the information, which is captured explicitly in the *Design Model*, referred to as *Code Generation*, as well as the evolution of the generated skeleton source code into a fully featured code base, what is being performed by the *Implementing* and *Integrating* tasks respectively. While *Implementing* is concerned with the development of missing code details for the individual subsystems and is thus performed by a respective *Subsystem Implementer*, *Integrating* deals with the development of glue code needed to integrate the subsystems into an overall software system. It is thus performed by a single responsible *System Integrator*.

Code Generation Transferring the structural and behavioral information, captured within the *Design Model*, into source code is what *Code Generation* is concerned with. Explicitly targeting a seamless transition into a procedural implementation language, MeDUSA does not facilitate object-oriented concepts (inheritance and polymorphism) throughout its modeling tasks. Therefore, the transition of the structural information, being captured in the *Design UML Model*, into respective procedural source code equivalents is relatively seamless and straight-forward and can thus to a large extend be automatically performed by a respective code generation tool (cf. Section 7.3 for a detailed survey on how the structural artifacts, captured in a MeDUSA-conformant *Design UML Model* can be transferred into ANSI-C code). While some exceptional behavioral aspects like the intra-object behavior, specified by a sufficiently detailed and consistent state machine, can as well be automatically transferred, most of the behavioral code fragments will have to be added manually. This is, because the gap that naturally exists between the abstraction levels reflected in the design model and resultant source code is mainly manifested in behavioral aspects, so that for example method bodies are not - or only insufficiently - specified, and can thus not be automatically generated.

Code Generation is jointly performed by the *Subsystem Implementer* as well as the *System Integrator*. The reason for the involvement of both roles is that the source code, being related to the externally visible interface of a subsystem, is as well important for the *Subsystem Implementer*, who has to implement the subsystem according to its specification, as well as for the *System Integrator*, who has to develop the glue code to integrate all subsystems (as well as the initialization code, etc.). The work products being produced are manifested in terms of several, yet incomplete source code files, which are jointly referred to as the *Skeleton Code*, thus emphasizing that details have to be added by subsequent tasks.

Implementing Taking the *Skeleton Code* produced for a subsystem during *Code Generation* as a starting point, the *Implementing* task is concerned with adding all necessary detail to obtain a complete source code base for a respective subsystem. This in particular comprises the implementing of all method bodies, which could not be automatically transferred by the *Code Generation* task, as well as all code details going beyond the abstraction level captured in the *Design Model*, as for example hardware related code for initialization or configuration of the underlying platform.

The work product produced by the *Implementing* task is the detailed code that is needed to transfer the subsystem related skeleton code into a fully-featured code base (for that subsystem). It has to be pointed out that while *Code Generation* can mostly be automated, *Implementing* still has to be regarded as a mostly handcrafted task (even if nowadays being supported by profound development tools), which requires experience and training and good knowledge of the underlying hardware platform.

Integrating *Integrating* is the conceptually last task covered by MeDUSA. As already outlined before, it is concerned with the development of glue code, needed to integrate the source code of the different subsystems into a valid and complete source code base for the overall software system. While most of the structural code fragments needed for the integration of the subsystems may already be automatically generated during *Code Generation*, there are parts that have to be manually dealt with, for instance the code related to the initialization and startup and to the configuration of the underlying hardware platform. As mentioned before, in order to guarantee a seamless and smooth integration, *Integrating* is performed by the *System Integrator* in close cooperation with the *Subsystem Implementers*, who are responsible for the *Implementing* of the individual subsystems. The outcome is the glue code needed for the integration of the different subsystems, as well as for initialization purposes.

6.2.2.6 Real-Time Analysis Discipline

As repeatedly motivated, real-time requirements related to timing and concurrency issues are of outstanding importance for embedded & real-time systems and thus have to be regarded intensely and as early as possible. It is thus an essential aspect of MeDUSA to continuously perform a real-time analysis, based on the work products of all constructive modeling disciplines, i.e. *Requirements Model*, *Analysis Model*, and *Design Model* respectively.

Naturally, the real-time analysis gains precision and significance in the same extend, the underlying models, gain expressiveness and accuracy. That is, while a *Preliminary Real-Time Analysis*, being based on the *Requirements Model*, may merely be employed to get an initial, yet vague, impression on possible performance and schedulability issues of the later software system, the *Interim Real-Time Analysis* and *Conclusive Real-Time Analysis*, being performed on the basis of the *Analysis Model* and *Design Model* respectively, yield increasingly detailed and resilient results. Nevertheless, even an early real-time analysis based on the *Requirements Model* is regarded to be important and worthwhile, as it might give early indications on the feasibility and realizability of the software and might help to identify hot spots that have to be especially investigated throughout adjacent modeling tasks of the *Analysis Modeling* or *Architectural Design Modeling Disciplines* respectively.

All tasks covered by the *Real-Time Analysis* discipline, namely *Preliminary Real-Time Analysis*, *Interim Real-Time Analysis*, and *Conclusive Real-Time Analysis* are performed by the *Real-Time Analyst*, who has to be profoundly educated in real-time

analysis techniques, having in particular fundamental experience in the estimation of CPU consumption times, something directly influencing the quality and the resilience of the analysis results.

Preliminary Real-Time Analysis While the consequent application of the *MeDUSA Actor Taxonomy* (cf. Figure 6.6) allows to explicitly express timing and concurrency issues in a way consistent to the current UML standard definition, it furthermore offers the possibility of analyzing the use case model with respect to those real-time requirements. Annotating timer periods or respective worst-case inter-arrival times to the identified timer and eventer actors and estimating execution times for the identified use cases (in detail for the respective worst-case execution time scenario subsumed by each use case), a performance analysis based on real-time scheduling theory and event sequence analysis may be applied already to the use case model.

While such an analysis - no matter how accurate and detailed it is performed - remains rather vague and does thus not allow to confirm the schedulability of a later design, it might provide early information about the feasibility and realizability of the software with respect to a respective hardware platform. That is, it might help to identify critical spots that have to be especially investigated, e.g. by constructing an early prototype. It might also help to decide that a selected hardware platform[9] is not suitable at all.

The output of the *Preliminary Real-Time Analysis* is defined to be an *Initial Task Report* as well as an *Initial Schedulability Report*, as exemplarily shown in Figure 6.31 and 6.32. While the first lists the identified system-actor interaction threads, the respective worst case scenario, and the estimation of the related CPU consumption (the utilization is computed as documented in [NL08]), the latter documents the results of the schedulability analysis, performed as detailedly documented in [Gom00].

Initial Task Report

#	Trigger	Scenario	Frequency (T_i)	CPU consumption (C_i)	Utilization (U_i)	Priority (P_i)
t_1	Sensor ADC Interrupt	Collect and preprocess ADC samples from sensor	25 μs	5 μs	0.2	HIGH (1)
t_2	Measurement Timer	Calculate Raw Flow Velocity from ADC samples	500μs	70 μs	0.14	HIGH (2)
t_3	Calculation Chain Timer	Calculate Flow Velocity, Volume and Mass Flow from Raw Flow Velocity and output them by PWM	100 ms	14.5ms	0.145	MED (4)
t_4	Digital Output Timer	Output Process Value on Digital Output	200μs	3μs	0.015	HIGH (3)
...	...	...	...	...	...	...

Figure 6.31: MeDUSA Example Initial Task Report (excerpt)

[9] As outlined in Section 6.1, MeDUSA assumes that hardware development is performed slightly ahead of software development.

Initial Schedulability Report

Task t_1 is an aperiodic, interrupt-driven task with a worst case inter-arrival time of $T_1 = 25\mu s$ and a CPU consumption time of $C_1 = 5\mu s$. It has the highest priority.

1. **Preemption time by higher priority tasks with periods less than** t_1. There are no tasks with periods less than t_1.
2. **Execution time** C_1 **for task** t_1. Execution time is $5\mu s$ what leads to a utilization of $5\mu s/25\mu s = 0.2$.
3. **Preemption by higher priority tasks with longer periods**. No tasks fall into this category.
4. **Blocking time by lower priority tasks**. Task t_2 may block task t_1 because it accesses the ADC samples collected by task t_1. We assume that the blocking time (needed to read out the ADC samples) can be estimated to $4\mu s$, which leads to a blocking utilization during period T_1 of $4\mu s/T_1 = 4\mu s/25\mu s = 0.16$.

The worst case utilization of task t_1 can thereby be computed as execution utilization + blocking utilization = $0.2 + 0.16 = 0.36$, which is well below the utilization bound of 0.69, so task t_1 will meet its deadline.

Task t_2 is a periodic task with a period of $T_2 = 500\mu s$ and a CPU consumption time of $C_2 = 70\mu s$. It has the second highest priority.

1. **Preemption time by higher priority tasks with periods less than** t_1. Task t_2 could be preempted by task t_1, which has a shorter period but a higher priority. The preemption utilization of task t_2 is 0.2
2. **Execution time** C_2 **for task** t_2. Task t_2 has an execution time of $70\mu s$, which leads to a CPU utilization of 0.14.
3. **Preemption by higher priority tasks with longer periods**. No tasks fall into this category.
4. **Blocking time by lower priority tasks**. Task t_3 may block task t_2 because it accesses the raw flow velocity calculated by task t_2. We assume that the blocking time (needed to access the flow velocity) can be estimated as $3\mu s$, which leads to a blocking utilization during period T_2 of $3\mu s/T_2 = 3\mu s/500\mu s = 0.006$.

The worst case utilization of task t_2 can thereby be computed as $0.2 + 0.14 + 0.006 = 0.346$ which is below the utilization bound of 0.69, so task t_2 will also meet its deadline.

Task t_3 is a periodic task with a period of $T_3 = 100ms$ and a CPU consumption time of $C_3 = 14.5ms$. It has the lowest priority of the four regarded tasks.

1. **Preemption time by higher priority tasks with periods less than** t_3. Task t_3 could be preempted by tasks t_1, $t2$ and $t4$, which all have a shorter period and a higher priority. The summarized preemption utilization of these tasks is 0.355
2. **Execution time** C_3 **for task** t_3. Task t_3 has an execution time of $14.5\mu s$, which leads to a CPU utilization of 0.145.
3. **Preemption by higher priority tasks with longer periods**. No tasks fall into this category.
4. **Blocking time by lower priority tasks**. Task t_3 has the lowest priority of the regarded tasks, so no tasks fall in this category.

The worst case utilization of task t_3 can be computed as $0.355 + 0.145 = 0.5$, which is below the utilization bound of 0.69, so task t_3 will also meet its deadline.

Task t_4 is a periodic task with a period of $T_4 = 200\mu s$ and a CPU consumption time of $C_4 = 3\mu s$. It has the third highest priority of the regarded tasks.

1. **Preemption time by higher priority tasks with periods less than** t_4. Task t_3 could be preempted by task t_1, which has a shorter period and a higher priority. The preemption utilization of task t_1 is 0.2.
2. **Execution time** C_4 **for task** t_4. Task t_4 has an execution time of $3\mu s$, which leads to a CPU utilization of 0.015.
3. **Preemption by higher priority tasks with longer periods**. Task t_4 can be preempted by task t_2, which has a higher priority and a longer period. Preemption utilization of task t_2 is $C2/T4$, which is $70\mu s/200\mu s = 0.35$.
4. **Blocking time by lower priority tasks**. Task t_4 may be blocked by lower priority task t_3 when it tries to obtain the next process value to be outputted on the digital output. As t_3 does need to block the process value for exclusive write access, we assume that blocking time will be around $5\mu s$, so a blocking utilization during period T_4 of $5\mu s/T_4 = 5\mu s/200\mu s = 0.025$ does result.

The worst case utilization of task t_4 can therefore be computed to $0.2 + 0.015 + 0.35 + 0.025 = 0.59$, which is below the utilization bound of 0.69, so also task candidate t_4 will meet its deadline.

Figure 6.32: MeDUSA Example Initial Schedulability Report (excerpt)

Interim Real-Time Analysis With the detailed specification of inter-object message communication, as it is captured in the *Analysis Model*, a more detailed real-time analysis can be performed, compared to the one being based on the *Requirements Model*. While the work products being produced, namely an *Unconsolidated Task Report* and an *Unconsolidated Schedulability Report*, are very similar to those initial reports, being produced during *Preliminary Real-Time Analysis*, they may now yield more detailed and resilient results. That is, CPU consumption times can now be estimated on the basis of individual messages (i.e. the time an object needs to consume a message and respond to it), being exchanged by the objects in the context of respective collaborations,

It has to be pointed out again that, even if valuable information can already be inferred from such an analysis, the overall task design is not performed earlier than during *Architectural Design Modeling*. In fact, the allocation of tasks to subsystems is one major criteria, which may be applied to the division of objects, as it is performed by *Subsystem Identification* and *Subsystem Consolidation*.

Conclusive Real-Time Analysis Based on the overall software architecture, which explicitly specifies the system decomposition in terms of fully encapsulated subsystems, thus also specifying the allocation of tasks among those subsystems in terms of the divided trigger objects, a *Conclusive Real-Time Analysis* can be performed. That is, as the consolidation of objects, which is performed during *Subsystem Identification* and *Subsystem Consolidation*, has lead to a final task design as well (as the active trigger objects are of course also affected by it), the schedulability of the overall system can now be analyzed in all detail. In this context, the communication and synchronization overhead that has been introduced with the respective subsystems (of course inter-subsystem communication is more *expensive* than intra-subsystem communication, which manifests itself within the subsystem's internal decomposition) can be explicitly regarded.

The *Conclusive Real-Time Analysis* is performed using the same techniques than *Preliminary Real-Time Analysis* and *Interim Real-Time Analysis*. Being based on the most detailed and precise model that is developed, the *Conclusive Real-Time Analysis* is the last real-time analysis task being performed, as the analysis of the system's performance and schedulability may subsequently be analyzed by directly executing and simulating the system, or respective parts of it. The *Detailed Design Modeling* discipline does indeed not change the allocation of active objects, and a subsequent real-time analysis is therefore not proposed by MeDUSA[10]. However, performance problems, which have been identified by the *Conclusive Real-Time Analysis*, indeed have to be regarded before going into *Class Design Modeling*, as a non-optimized detailed design for the internal decomposition of a subsystem may of course affect the overall system performance in a significant negative way.

[10]Indeed, it is assumed that the overhead to perform another detailed analysis does not pay of here, as subsequently, code fragments may be generated and directly simulated and analyzed by their execution.

6.2.3 MeDUSA Method Operations

The MeDUSA *Method Operations* is defined in terms of a workflow and five workflow patterns, which correspond to the five software construction life-cycle phases covered by the method, namely

- *Requirements Workflow Pattern*
- *Analysis Workflow Pattern*
- *Architectural Design Workflow Pattern*
- *Detailed Design Workflow Pattern*
- *Implementation Workflow Pattern*

Each workflow pattern defines how the modeling tasks, comprised by each discipline, are executed to construct a respective discipline-specific model, and how the tasks of the *Real-Time Analysis* discipline are continuously applied. The defined workflow patterns are introduced in detail in the following paragraphs, followed by the definition of the overall *MeDUSA Workflow*, which structures the method execution into five phases, each describing iterations of the respective workflow patterns.

Requirements Workflow Pattern The *Requirements* workflow patterns defines the activities related to the construction of the *Requirements Model*. As outlined by Figure 6.33, the pattern describes the simultaneous application of *Use Case Modeling* and *Use Case Details Modeling*, as well as that of a *Preliminary Real-Time Analysis* to ensure, real-time and concurrency constraints are adequately accounted.

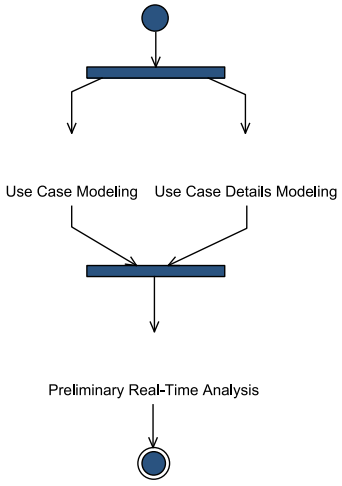


Figure 6.33: The MeDUSA Requirements Workflow Pattern

The parallel execution of *Use Case Modeling* and *Use Case Details Modeling*, as indicated by Figure 6.33 is of course stylized, but it indicates that the two tasks are closely related. In practice, the development of the *Requirements Model* will be a rather iterative process that is realized by the interlocked execution of both tasks. That is, it will start with the retrieval of an initial set of use cases, which will then be successively described in detail, leading in turn to a revision of the initially identified use cases, and so on. For instance, the description of a use case's details might give valuable information about the adequacy or inadequacy of the granularity of the use cases (often beginners tend to model too fine-grained use cases, resulting in too short descriptions).

As soon as an overall integrated *Requirements Model* of satisfying quality has been gained, a *Preliminary Real-Time Analysis* can be performed, to identify potential performance problems and to ensure that real-time and concurrency constraints are adequately reflected. If being automated by means of an analysis tool, such an analysis may - different to its indication in Figure 6.33 - be executed not only on the final, but as well on intermediate revisions of the *Requirements Model*.

Analysis Workflow Pattern The *Analysis* workflow pattern pools the activities related to the construction of the *Analysis Model* to gain a detailed and profound understanding of the problem domain.

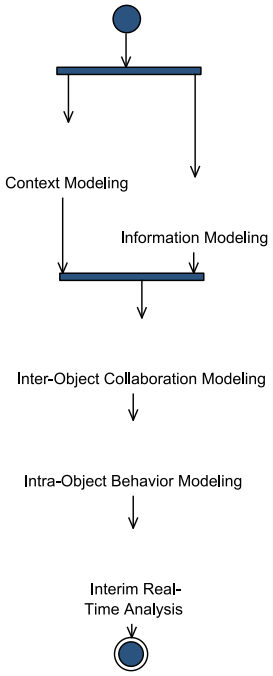


Figure 6.34: The MeDUSA Analysis Workflow Pattern

As outlined by Figure 6.34, it is started with the conceptually parallel execution of *Context Modeling* and *Information Modeling*, to identify *trigger*, *interface*, and *entity* objects, in accordance to the *MeDUSA Object Taxonomy* (cf. Figure 6.11). Sub-

sequently, *Inter-Object Collaboration Modeling* is performed to identify object collaborations for the identified use cases and by this also the remaining *control* and *application-logic* objects. Having captured the internal behavior of the identified objects in terms of *Intra-Object Behavior Modeling* as well (at least for state-dependent control objects), the *Analysis* workflow pattern is concluded by an *Interim Real-Time Analysis* to identify potential performance problems that have to be in particular regarded during subsequently performed design activities.

It has to be emphasized that while indeed no software architecture is designed yet, and while understanding of the problem domain is the central goal being targeted, many design decisions are actually being anticipated during execution of the *Analysis* workflow pattern. That is, the impact of a analysis-related modeling decision on the later subsystem identification and consolidation is great and should not be underestimated. As an example, the decision of modeling a piece of application logic either by a self-contained *algorithm* object, separated from the data it accesses, or by attributing it to a respective *entity* object, may be quoted, as it directly affects the division of objects into subsystems.

Architectural Design Workflow Pattern While *Analysis* is concerned with understanding the problem domain, *Architectural Design* is about building a solution. That is, the *Architectural Design* workflow patterns subsumes the tasks to construct the overall software architecture in terms of decomposition of the system into fully encapsulated and self-contained subsystems, which collaboratively perform the system behavior.

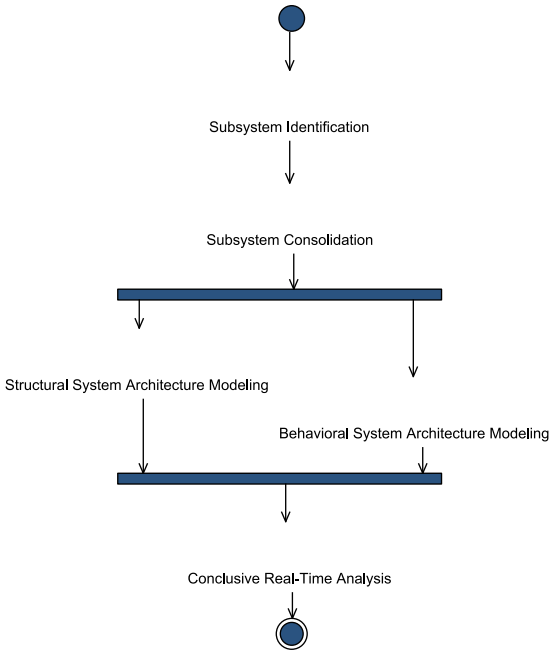


Figure 6.35: The MeDUSA Architectural Design Workflow Pattern

Accordingly, as outlined by Figure 6.35, *Subsystem Identification* is initially performed to divide the analysis objects of the *Analysis Model* into subsystems. The subsequent execution of *System Consolidation* ensures that the initial subsystem decomposition is sustainable under design considerations. That is, it literally turns the initially divided analysis objects into design objects, while indeed also consolidating the interfaces of the identified subsystems according to their later integration needs. As already indicated, *Subsystem Identification* and *Subsystem Consolidation* are strongly related to each other, so they may be jointly executed in one step. *Structural System Architecture Modeling* and *Behavioral System Architecture Modeling*, which are subsequently performed, are then concerned with integrating the different subsystems, which have been individually defined in terms of their structural and behavioral properties and context dependencies, into an overall software system. While Figure 6.35 indicates that the two activities are somehow executed simultaneously, it has to be pointed out that *Structural System Architecture Modeling* is started with some advance, as it establishes the structural relationships between the identified subsystems, based on which the behavioral relationships, being addressed during *Behavioral System Architecture Modeling*, are then defined. Both activities are however strongly intertwined, what is why they are indicated within Figure 6.35 as parallelly executing activities.

To analyze the feasibility of the developed software architecture in terms of real-time and concurrency aspects, a *Conclusive Real-Time Analysis* is successively performed. Unlike the previously executed *Preliminary Real-Time Analysis* and *Intermediate Real-Time Analysis*, the analysis is now based on the actual task design, as it manifests itself after the division and consolidation of all active *trigger* objects, so precise and resilient can now be inferred on the actual schedulability of the system.

Detailed Design Workflow Pattern Having defined the overall software architecture, the *Detailed Design* workflow pattern, as denoted by Figure 6.36, may be executed to develop the detailed design of each identified subsystem.

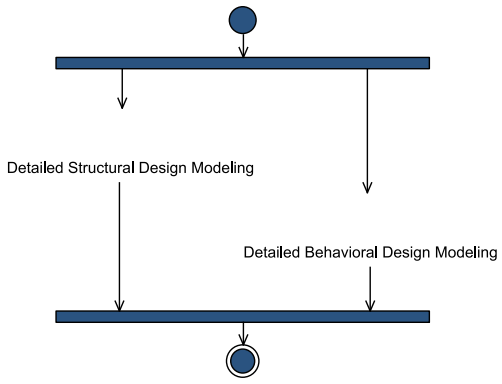


Figure 6.36: The MeDUSA Detailed Design Workflow Pattern

Here, at first information hiding classes for all design objects have to be developed, as well as for the subsystems' owned ports. The specifications of individual object be-

haviors, which were initially modeled during *Analysis Modeling* now also have to be updated (to reflect the changes of *Subsystem Consolidation*) and further detailed. This is done in terms of *Detailed Behavioral Design Modeling*. As in *Architectural Design Modeling*, both activities are performed pretty much in parallel, while *Detailed Structural Design Modeling* of course has to be started with some slight advance, as it defines the structural elements, which may then be referred to during *Detailed Behavioral Design Modeling*. As the subsystem's interfaces have already been explicitly defined before, the *Detailed Design* workflow pattern can - different to all preceding ones - be performed in parallel for each subsystem.

Implementation Workflow Pattern Based on the detailed design, as it is captured in the *Design Model*, the *Implementation* workflow pattern can be performed. It is concerned with transferring the information, captured in the *Design Model*, into respective source code and with adding all necessary code details to build up a valid and consistent source code base from it.

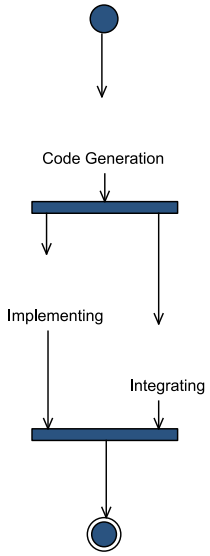


Figure 6.37: The MeDUSA Implementation Workflow Pattern

Conceptually, this is performed in two steps. First, as outlined by Figure 6.37, code generation is performed to generate skeleton code from the structural and parts of the behavioral information, being explicitly captured in the *Design Model*. Second, the code generated for each subsystem has to be enriched with all code details that are neccessary to turn it into a valid source code base for that subsystem, and the glue code, needed to integrate the different subsystem-related source code building blocks, has to be implemented. As indicated in Figure 6.37, *Implementing*, which is concerned with enriching the source code of each subsystem with the needed details, and *Integrating*, which addresses the development of the missing glue code, are pretty much executed in parallel. As already motivated in Section 6.2.2.5, a close cooperation between both activities is inevitable to ensure consistency of the overall source code base.

MeDUSA Workflow According to the five software construction lifecycle phases being covered, the *MeDUSA Workflow*, as denoted by Figure 6.38, is defined in terms of five phases, namely *Requirements Phase*, *Analysis Phase*, *Architectural Design Phase*, *Detailed Design Phase*, and *Implementation Phase*, which are sequentially executed.

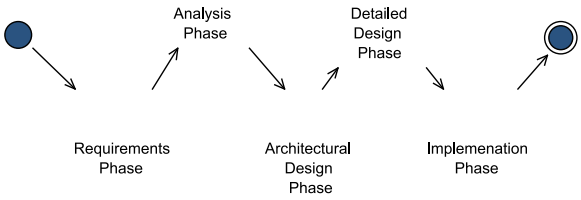


Figure 6.38: The MeDUSA Workflow

Each phase is a self-contained unit, concerned with the development of a respective discipine specific model. As already mentioned, all design related activities rely on a single *Design Model*, so with the conclusion of the *Architectural Design Phase* the *Design Model* is indeed not yet complete. It may however be regarded as being self-contained at that time, as it captures the overall software architecture. The *Implementation* discipline is also somehow outstanding, as it does not produce a model in the literal sense; however, the source code that is developed during its execution may be regarded as a model as well.

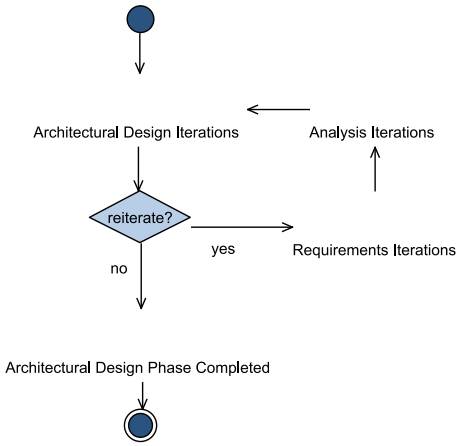


Figure 6.39: The MeDUSA Architectural Design Phase

As exemplarily denoted by Figure 6.39 for the *Architectural Design Phase* (the other phases are defined accordingly), each respective phase specifies the iterative execution of the homonymous workflow pattern and allows reiterations of those patterns, addressing earlier lifecycle phases, where this is regarded to be necessary. What has to be emphasized in this context is the explicit iterative nature of the method that manifests itself in those backflows. As Jacobson et. al. exemplarily point it out in the

context of architectural design: *"When the division into subsystems is made, in some cases it may also be desirable to modify analysis objects also. This may be the case, for instance, when an entity object has separate behavior that is functionally related to more than one subsystem. If this behavior is extracted, it may be easier to place the entity object in a subsystem."* [JCJv92]. It is - to point that out in all clarity - not the confession of a mistake, if the activities of an earlier lifecycle phase are reiterated, but a natural and essential aspect of a creative process that leads to models of higher quality and to greater awareness and justifyability of the made design decisions.

6.3 Reflective Characterization

As already pointed out, MeDUSA was not designed from scratch, but by taking into consideration approved principles, concepts, and techniques. While *COMET* and *ROOM* have been already named (cf. Section 5.2.1), and may be regarded as some sort of precursors, MeDUSA further incorporates many concepts of even earlier approaches, outlined within Section 3.1. As far as its overall method design is concerned, a general affinity towards those object-oriented approaches of the 1990's cannot be denied, where *OOSE* [JCJv92] and the closely related *Unified Object Modeling* [RS99], as well as *COMET* have to be explicitly named due to the high degree of similarity.

The modeling of requirements in the form of use cases or scenarios, which is quite essential for those approaches, has been originally proposed by Jacobson et. al. [JCJv92], and can for instance also be found in *OBA* [RG92]. The modeling of the system's external context and intensive long-living data, as incorporated into MeDUSA's *Context Modeling* and *Information Modeling* tasks, reaches even back to those *Structured Analysis and Design* approaches of the 1980's ([Jac83], [WM85]) and was subsequently picked up by some object-oriented approaches, including also *COMET*. The dynamic modeling of object collaborations, as it is incorporated into MeDUSA in terms of *Inter-Object Collaboration Modeling* has to be credited mainly to OOD [Boo91] and OOSE [JCJv92].

The idea of introducing object structuring criteria to support the identification of objects during analysis may be as well traced back to the object-oriented approaches of the 1990's ([SM88], [CY91], [JCJv92]), and was later incorporated as one of its essential ingredients into Gomaa's *COMET* method. The differentiation into active and passive objects, as it is incorporated into the MeDUSA Object Taxonomy may be traced back to Booch [Boo86], and has found - via HOOD [Rob92] - its way into those object-oriented real-time methods like *COMET* or *ROOM*.

Modeling of internal object behavior may be similarly classified as originating from those object-oriented approaches of the early 1990's. It was in particular promoted by Harel [Har88][HG96] as well as Shlaer and Mellor [SM92]. Not least, it has found its way into *ROOM*, where the modeling of individual object's behavior in terms of state machines is quite essential (cf. Section 5.2.1). The subsequent division of analysis objects into subsystems, as the first step of system design, is of course an essential

activity, being coverd by all object-oriented approaches. Jacobson et. al. [JCJv92] may however be explicitly named here, as he names a number of design principles (locality in changes, functional coupling, etc.) to support the division process.

Regarding the specification of a software system's architecture, of course a myriad of publications can be found. However, the hierarchical manner, in which a MeDUSA architecture is organized in terms of fully encapsulated, collaborating subsystems with only explicit context dependencies and a well-structured internal decomposition, may most likely be traced back to *ROOM*, whose concise and coherent architectural modeling language was already pointed out in Section 5.2.1. The concepts of ports and connectors, as it is intensely applied by *ROOM* to preserve encapsulation, was of course intensively promoted also by component-based approaches. In this context, *Catalysis* [DW98] may be explicitly named, where respective concepts were first represented by means of the UML.

While the development of a detailed class design for the identified design objects, as it is performed by *Detailed Structural Design Modeling*, is indeed a rather mechanical process inside MeDUSA, where consolidation is performed on the level of objects, this has of course been of major interest to the research community in the early 1990's. What has been not so intensively regarded is the transfer of those concepts into source code equivalents. While *ROOM* may be quoted as a notable exception, which - even if not explicitly documented - supports a transition of a *ROOM*-based design into a C++ implementation by means of the ObjectTime toolset, most methods being published in the 1990's were developed as mere design methods, thus not covering the implementation phase. Additionally, not much third-party research interest seems to be documented related to this, in particular on how component-based concepts like ports and connectors can be seamlessly transferred. While Wong may be named as one of the few that explicitly describes the transformation of those concepts for object-oriented languages [Won93], a mapping of port and connector concepts into a procedural, non object-oriented implementation language, does not seem to have been intensively regarded.

However, even while adopting several approved and advantageous concepts, MeDUSA may be rightly denoted as a self-contained method. That is, its *continuity* and *systematics* as well as its prominent characteristics set it apart. They will be shortly outlined in the following.

Model-Based As it is quite obviously inferrable from the definition of MeDUSA, models play an outstandingly important role throughout the overall method, and modeling, the engineering of models, is MeDUSA's most essential activity. Therefore, MeDUSA may be unreservedly denoted as a *model-based* method. While the potential of automatic transformations between the employed discipline-specific models increases with the onwarding life-cylce, not all transformations are indeed automizable. Some remain to be quite creative, so that according to the definitions provided in Section 2.3.4, MeDUSA may not be unambiguously denoted as being *model-driven*, even if some part of the research community would probably denote it as such.

Use Case-Driven MeDUSA may further be characterized as *use-case driven*. That is, use cases are the central concept, around which essential tasks of MeDUSA are being performed. Being identified during *Requirements*, use cases indeed affect *Analysis* and *Architectural Design* phases, e.g. during *Inter-Object Behavior Modeling* or *Behavioral System Architecture Modeling*.

Instance-Driven & Class-Based While the use case concept is closely related to object-oriented engineering, MeDUSA is not and object-oriented method, as the application of the central object-oriented concepts of inheritance and polymorphism is indeed not enforced to meet the goal of *seamless continuity* (cf. Section 4.3). Indeed, all of MeDUSA's modeling activities from the early *Requirements Modeling* up to the late *Architectural Design Modeling* (regarding the subsystems' interfaces) or even *Detailed Design Modeling* (regarding the subsystems' internal decompositions) are based on objects rather than classes, so that MeDUSA might be accordingly denoted as an *instance-based* method. Due to this the application of inheritance is not enforced, and object-oriented concepts may only be optionally applied during *Detailed Design Modeling*, in case a *Subsystem Designer* decides on this. MeDUSA however does not prescribe to use object-oriented concepts, as it has the goal of a seamless transition from detailed design into a procedural implementation. Being oriented at classes and objects and disregarding inheritance and polymorphism, MeDUSA may thus - according to the categorization provided by Wegner [Weg87] - be characterized more as a *class-based* method[11].

The main reason why MeDUSA was designed to be *instance-driven* and *class-based* is that this is regarded to be more seamless, compared to an approach, which vacillates between modeling on the class and modeling on the object level, and is - as experience has shown - as such easier amenable to developers in the marginal application domains.

Real-Time Aware The explicit handling and awareness of real-time constraints may be quoted as a further distinct characteristic of MeDUSA. That is, through its *Actor Taxonomy* and *Object Taxonomy*, MeDUSA facilitates the explicit handling of real-time and concurrency constraints throughout the overall development lifecycle. The real-time awareness further manifests itself in the continuous real-time analysis that has been incorporated into the method. While *COMET* did only propose a respective schedulability analysis after the task design, which is performed late during architectural design, MeDUSA incorporates a respective real-time analysis to be continuously performed, beginning with a preliminary early analysis on the basis of the identified use cases.

[11] The usage of the `Class` concept within MeDUSA corresponds to what is covered by a *class-based* programming language (cf. [Weg87]). It is therefore reasonable to denote MeDUSA as a *class-based* method, even if classes actually play a subordinate role, due to the *instance-driven* nature of the method.

Chapter 7

Languages - UML & ANSI-C

Regarding overall software development, several languages may be named as being involved, reaching from informal natural language, used to formulate requirements, up to very formal languages, as for example used for modeling, implementation, or even testing. Here, of course, the focus will be on software construction only, and it will of course be limited to *formal* languages. In this context, two such languages may be named, namely the UML modeling language, which is used to specify nearly all of the involved models[1], as well as the implementation language, namely ANSI-C, which is used to implement the resulting source code.

To support a detailed and profound understanding of the method, going beyond what can be inferred from the example diagrams provided in Section 6.2, it has to be precisely laid out, which UML concepts are actually applied throughout the tasks of the method, and how the precise structure of the underlying UML models is constituted. Even while not all of these model elements may be directly visible in the UML diagrams, as a diagram of course hides certain details, a clear understanding of the precise UML model structure is of great significance to ensure consistency and traceability. The basic structure of the three MeDUSA UML models, namely *Requirements UML Model*, *Analysis UML Model*, and *Design UML Model* will thus be detailedly laid out in the following Section.

Even while the UML is a very comprehensive modeling language, it does not cover all MeDUSA defined concepts to the required extend. MeDUSA's taxonomies are of course not covered by the UML, and there are a number of discrepancies, which may be attributed to the instance-driven nature of the method. Having defined the precise structure of all MeDUSA defined UML models in terms of the employed model elements and their relationships, it can be easily illustrated, where a discrepancy in concepts between the UML and MeDUSA is noticeable. Based on this, the necessary extensions to the UML, realized by means of MeDUSA UML profiles can then subsequently be introduced within Section 7.2.

[1] Except those extensions, being introduced to textually describe use case details (cf. Section 6.2.2.1), all formal MeDUSA models are indeed specified using the UML.

Having precisely defined the structure of the MeDUSA *Design UML Model*, the generation of source code, which is based on it, can then also be addressed. That is, it can be broken down to a specification on how to transform the employed UML concepts (i.e. meta classes) into respective implementation language constructs. This will be done in the concluding section of this chapter. What has to be pointed out is that, while such a mapping may usually be arbitrary complex, covering a lot of optimizations and improvements, and may as such be applicable only within certain specific constellations, the generation schema presented in Section 7.3 is meant to be generally applicable. That is, it does not reflect what may be achievable in terms of optimization, but rather serves to demonstrate feasibility.

7.1 MeDUSA UML Models

The precise structure of the MeDUSA related UML models has to be defined in terms of a detailed specification about which kind of modeling elements are contributed via a respective UML diagram, and on how those modeling elements are structurally related to each other.

Defining the structure of the *Requirements UML Model*, *Analysis UML Model*, and *Design UML Model*, could be done formally by characterizing their respective underlying meta-model in terms of concrete and abstract syntax, as well as the static and dynamic semantics that applies to the set of all valid MeDUSA UML models (cf. Section 2.3.3). Being a UML-based method, this could be done in terms of a specialization of the UML meta-model, in terms of its abstract syntax and its static semantics (as the concrete syntax and dynamic semantics do not affect the structure and can be preserved). While such a formal approach would lead to the desired goal of an exact specification of the structure of those UML models, being developed in the context of MeDUSA, it has some drawbacks. First, it would be very complicated and costly to achieve, as the UML specification is a quite vast document. There would also be the danger of restricting the modeling capabilities too strong, so that the flexible application of MeDUSA could not be further guaranteed. Further, the concrete mapping between the actual UML diagram elements and their underlying modeling artifacts would still remain somehow vague.

Therefore, a different, less formal but more practical approach is chosen here. For each UML diagram, being defined as work product of a MeDUSA task, the relevant UML concepts, i.e. meta-classes as well as related meta-associations, will be briefly sketched, based on the example diagrams that were introduced in Section 6.2. Due to the large number of UML concepts being involved, this will not be done exhaustively but for the relevant key concepts only, in particular, where a more restricted static semantics than that defined by the UML is employed. If for example a `State Machine` is used to represent global system states in a *Requirements UML Model*, it will be specified how the `State Machine` is structurally integrated into the model (i.e. it is attached as `owned Behavior` of the system `Component`), but it will not be detailedly laid out that the inherent contribution actually comprises also a nested `Region` and

various contained States, Pseudostates, Final States and Transitions, or even other UML elements.

The notation that is applied to describe the instance specification of the UML models is that of UML object diagrams, where instance specifications are used to represent model elements, i.e. instances of meta-classes, and links are used to represent instances of meta-associations, thus depicting relationships between model elements. Due to a lack of space, those instance specifications will be limited to only reflect the aforementioned key concepts. A more elaborated specification of the example MeDUSA UML models' structure can be found in [NL08].

7.1.1 Requirements UML Model

As defined by MeDUSA's method content, the *Requirements UML Model* is constructed indirectly by creating one or more *Use Case Diagram(s)*, a *Global System States Diagram* in the form of a UML state machine diagram, as well as *Use Case Details Diagrams*, which are developed either as a UML activity diagrams, as depicted by Figure 6.10, sequence, or state machine diagrams, for those use cases, which are not alternatively described by narrative textual descriptions.

Use Cases - External Relationships The structure of the *Requirements UML Model* is rather straightforward. It is basically determined by those artifacts, being contributed by the *Use Case Diagram*, which is usually the first diagram being developed during *Requirements Modeling*. As outlined by Figure 7.1, a Component, representing the system, all external Actors, as well as all Dependencies (between Actors) and those Associations originating from them, are directly contained by the root element of the *Requirements UML Model*, which is a Model.

All other artifacts are directly or indirectly contained by the system Component. That is, Use Cases may be directly contained, or indirectly via Packages, which might be introduced to group Use Cases and related elements. They thus also contain all internal Actors, Associations between internal Actors and Use Cases, and all Dependencies originating from a nested element (Actor or Use Case). Packages can contain other Packages so that an arbitrarily nested hierarchical structure can be gained. Include, Extend, and Generalization relationships, which are modeled between Use Cases, are directly contained by the Use Case, serving as source of the relationship.

Use Cases - Internal Details The key artifacts, being contributed by the *Use Case Details Diagrams* differ, dependent on the behavioral formalism that is applied. In case of an activity diagram, as depicted by Figure 6.10, the main artifact, being contributed, is of course an Activity, which is - as depicted by Figure 7.1 - associated as owned Behavior to the respective Use Case. It usually contains a single Initial and Final Node, as well as a couple of named Opaque Actions, which are used to

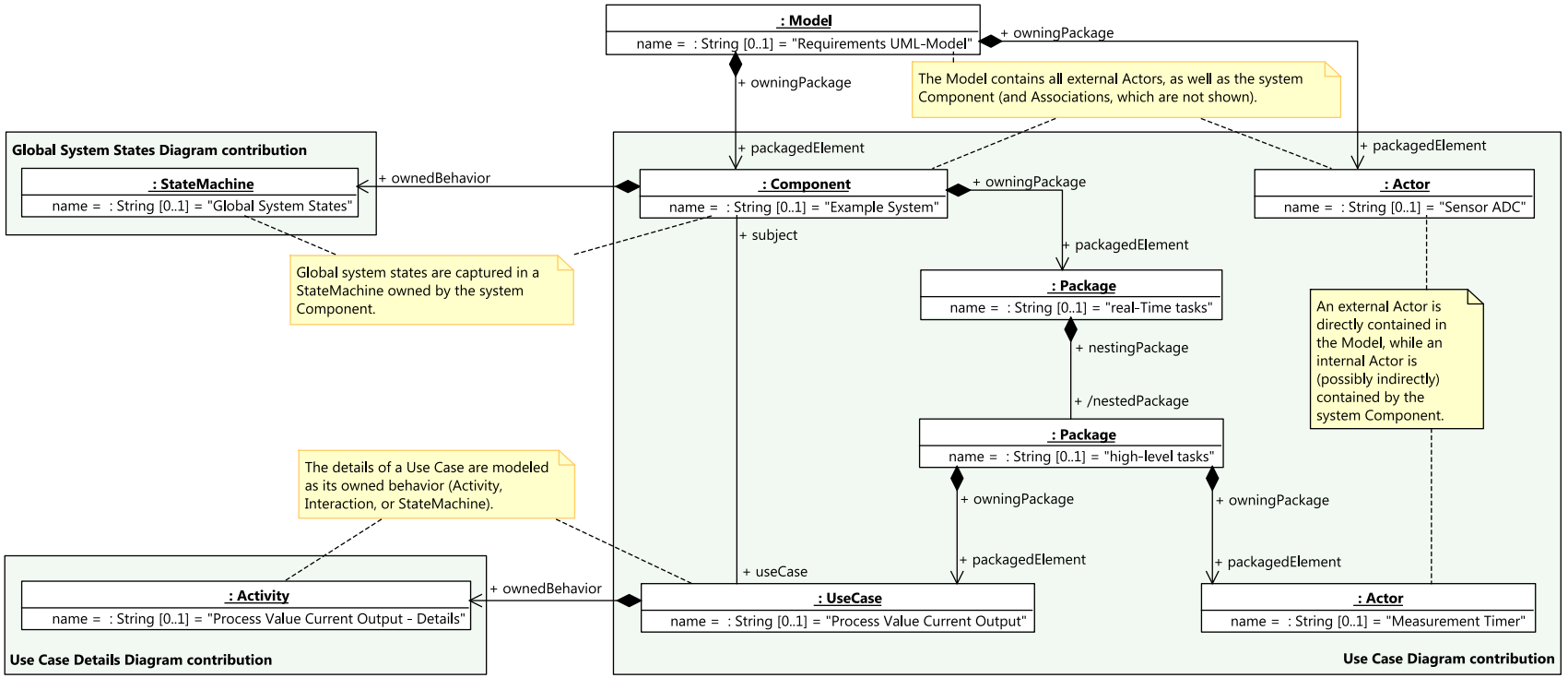


Figure 7.1: Examples of Key Artifacts contained in a MeDUSA Requirements UML Model

represent individual steps within the system-actor interaction, which is detailedly described. Decision Nodes and Merge Nodes, in combination with Control Flows are used to model the different scenarios subsumed by the Use Case. To depict, which scenario might occur under which conditions, Guards can be attached to those Control Flows, originating from a *Decision Node*. Within the Guards, arbitrary boolean conditions can be specified. One option within such an expression might be to use an *Opaque Expression*, which might refer to a global system state by means of the oclInState OCL expression, as it is exemplarily shown in Figure 6.10. This situation is represented within Figure 7.1 accordingly.

Global System States The key artifact contributed via the *Global System States Diagram* is a State Machine, which is - as depicted by Figure 7.1 - directly contained by the system Component as its owned Behavior. As the State Machine is meant to only reflect the different global system states and transitions between them, its detail level may be kept rather low. That is, neither Triggers (with Guards) nor Effects have to be specified.

7.1.2 Analysis UML Model

The structure of the *Analysis UML Model* is affected by the contributions of those UML object diagrams, which are developed during *Context Modeling* and *Information Modeling*, those UML communication and sequence diagrams, being developed during *Inter-Object Collaboration Modeling*, as well as those behavior diagrams (activity, state machine), being developed during *Intra-Object Behavior Modeling*. While the root element of the *Analysis UML Model* is of course again a Model, its structure is not as hierarchical as that of the *Requirements UML Model*. That is, the Instance Specifications (used to denote objects as well as links) and the Dependencies, being contributed by the *Context Diagrams* and *Information Diagrams*, are for example directly contained in the Model, rather than being grouped into Packages, as outlined by Figure 7.2.

Collaborative (Inter-Object) Behavior The Interactions, which are the key elements contributed via the *Inter-Object Collaboration Diagrams*, as well as all Behaviors (State Machines, Activities, Interactions), being contributed via the *Intra-Object Behavior Diagrams* are also directly contained within the Model, as indicated by Figure 7.2. This is, because - due to the characteristic of MeDUSA to be *instance-driven* - no Behavioral Classifiers that could serve as containers have been identified at the time those diagrams are developed.

Within the Interactions, being contributed via the *Inter-Object Collaboration Diagrams*, Lifelines are used to represent the individual analysis objects, and Messages are modeled between them accordingly via Message Occurrence Specifications, which on the one hand serve as Message Ends, and on the other hand cover the respective Lifeline. As there is no enclosing Behaviored Classifier to own the

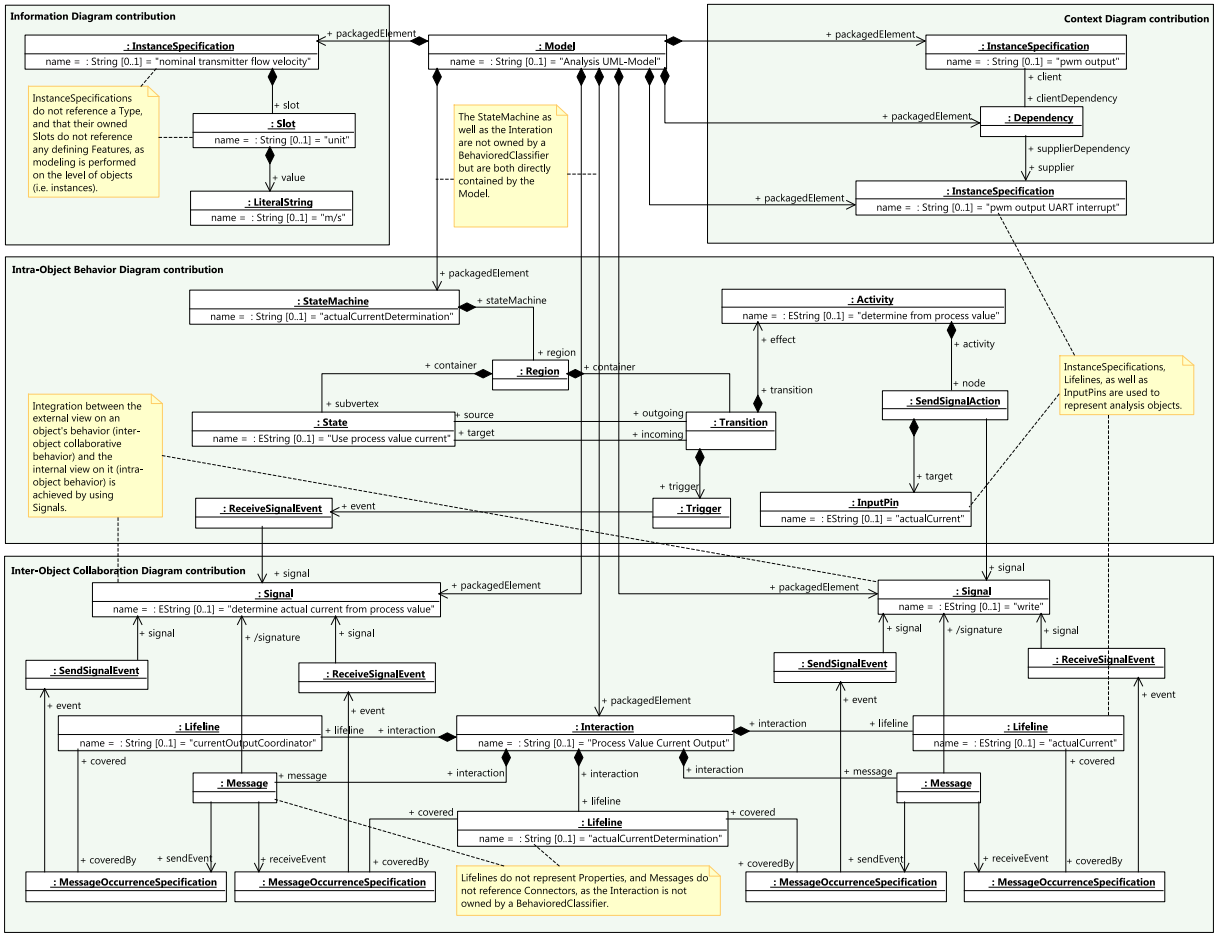


Figure 7.2: Examples of Key Artifacts contained in a MeDUSA Analysis UML Model

Interaction, those Lifelines and Messages are not associated to certain structural elements, too. That is, Lifelines do not reference represented Properties and Messages do not specify Connectors, even if this could be modeled.

Internal (Intra-Object) Behavior The artifacts, being contributed by the *Intra-Object Behavior Diagram*, of course differ dependent on the respective behavioral formalism, which is applied. In case of a State Machine, States and Transitions may be named as the most significant contributed elements. A Transition between States usually requires a Trigger and causes a certain *Effect*, which is an arbitrary Behavior, being executed when the Transition is taken.

Signals are used as a means to integrate both, the internal and the external view on the behavior of an analysis object, so literally seen, they represent the messages that an analysis object might understand. That is, on the one hand they are referenced by the sending and receiving Message Ends of the Messages, being represented as Message Occurrence Specifications in those Interactions that depict inter-object collaboration behavior, and thus serve as Signatures of those Messages. On the other hand they might be referenced from within the State Machine or Activity, which is used to model intra-object behavior. As outlined by Figure 7.2 exemplarily in case of a State Machine, the reaction of an object to the receipt of such a Signal may be the transition into a new State, what is modeled by specifying it as the Event of a respective Trigger (via a Receive Signal Event). The fact that the sending of Messages is triggered from within the internal behavior of an object, may be modeled by specifying a Send Signal Action as part of the Effect of a Transition, as shown in Figure 7.2.

It has to be pointed out that, while the UML defines that a Signal *"triggers a reaction in the receiver in an asynchronous way"*, this is not the intended meaning in a MeDUSA *Analysis UML Model*. In fact, the usage of a Signal should not indicate any details about the synchronism or asynchronism of a message. While the UML proposes to use Signals to denote asynchronous messages, and Operations to denote arbitrary ones, within a MeDUSA *Analysis UML Model* this differentiation is not applied, as Operations cannot be consistently modeled. This is, because in difference to Signals, which are themselves Classifiers, Operations are just Behavioral Features, and as such have to be owned by a respective Classifier, which is due to the *instance-drivenness* of the method, not yet designed within the *Analysis UML Model*. Therefore, Signals, being directly contained by the Model, are used, to depict both, synchronous as well as asynchronous messages.

As Signals are used within a MeDUSA *Analysis UML Model* and Operations are used within a MeDUSA *Design UML Model*, both referring to the same concept, namely the response of an analysis respectively design object to the receipt of a respective message. The term *Message* will thus be consistently used in the following to refer to both concepts for the sake of clarity[2].

[2]This is also reflected by a respective Stereotype in the MeDUSA *Analysis UML Profile* and *Design UML Profile*, introduced in Section 7.2

7.1.3 Design UML Model

Similar to the external and internal view on the behavior of an individual analysis object, as it is captured by the *Inter-Object Collaboration Diagrams* and *Intra-Object Behavior Diagrams*, the internal decomposition and the external interfaces of an individual subsystem, as well as the structural and behavioral relationships established between different subsystems, are captured during *Architectural Design Modeling* and *Detailed Design Modeling*.

As subsystems - in the context of MeDUSA - are regarded to be fully self-encapsulated architectural building blocks with only explicit context dependencies, the internal decomposition of a subsystem is well decoupled from its externally visible structural and behavioral properties, which are relevant to integrate the subsystems. Thus, as far as the externally visible interfaces of a subsystem have been fixed, as it is done during *Architectural Design Modeling*, the internal decomposition of the different subsystems may then be developed independently during *Detailed Design Modeling*. Because of this, the *Design UML Model* - unlike the *Requirements UML Model* and *Analysis UML Model* - may be developed as a fragmented model, where each subsystem is specified in a subsystem-specific model fragment, and the integration of the individual subsystems is described inside a system-related model fragment. It is thus reasonable to depict the structure of the *Design UML Model* along the different model fragments that build up the overall model.

Subsystem Model-Fragment

Structural Specification of a Subsystem's Internal Decomposition The initial contribution to a subsystem-specific model fragment is of course added via the *Initial* and *Consolidated Structural Subsystem Design Diagrams*. That is, as depicted by Figure 7.3, a `Component` representing the subsystem, as well as externally visible `Ports` with their respective `Types`, being modeled as `Classes`, are created this way, as well as the internally composed `Parts`, owned by the subsystem `Component`. Even while the *Consolidated Structural Subsystem Design Diagram* is developed as a revision of the *Initial Structural Subsystem Design Diagram*, its detail level is conceptually the same, in a sense that the same kind of UML model elements are contributed.

Structural Specification of a Subsystem's External Interfaces While `Types` for the internally composed *Parts* are not developed before *Detailed Design Modeling*, the specification of `Classes` as `Types` for the identified `Ports` is already done during *Architectural Design Modeling*. The reason for this is that they are needed to specify the required and provided `Interfaces`, which are exposed by a `Port`. Those `Interfaces`, together with the `Operations` that they comprise, are contributed via the *Initial* and *Consolidated Structural Subsystem Interface Design Diagrams*, as depicted as well by Figure 7.3. As in case of the *Initial* and *Consolidated Structural Subsystem Design Diagrams*, the *Consolidated Structural Subsystem Interface Design Diagram*

is a revision of the *Initial Structural Subsystem Interface Design Diagram*. In contrast to above mentioned diagrams however, additional artifacts are contributed by the consolidated revision of the diagram, namely Parameters of Operations together with their respective Types, which are mostly Data Types.

The fact that an Interface is exposed as required or provided by a Port is determined in this context, by whether the Port's Type specifies an Interface Realization, as outlined in Figure 7.3, or Usage (not depicted in Figure 7.3) towards the respective Interface. It has to be emphasized that - to achieve the goal of complete self-encapsulation - all Interfaces have to be modeled from the viewpoint of the individual subsystem, so that the specified Interfaces are also directly contained by the subsystem Component, and no Interfaces of other subsystem-related model fragments are referenced.

Behavioral Specification of a Subsystem's External Interfaces The behavioral aspects of the externally visible properties of each subsystem are specified via the *Initial* and *Consolidated Behavioral Subsystem Interface Design Diagrams*. Here, one or more Protocol State Machines are developed. In case behavior that affects more than one Port is depicted, the respective Protocol State Machine is owned directly by the subsystem Component, as depicted within Figure 7.3. In case a single Port is addressed, the respective Protocol State Machine is owned by the Port's Type. Similar to the *Intra-Object Behavior Diagram*, being developed during *Analysis Modeling*, the integration between the behavioral and structural view on the subsystem's externally visible Interfaces is achieved by specifying Triggers for the Protocol Transitions of the respective Protocol State Machine. Effects are not modeled for Protocol State Machines, as defined by the UML.

Similar to the *Analysis UML Model*, where Signals and respective Receive Signal Events and Send Signal Actions are used to integrate the different behavioral views, Operations and respective Receive Operation Events are uses accordingly to integrate the behavioral protocol of the subsystem's Interfaces with their structural specifications. Note that according to the UML, an Operation may be used to depict both, a synchronous call as well as an asynchronous one. As the messages arriving at a subsystem are specified as Operations in its provided Interfaces, those Operations can be directly referenced, and Signals, which were used in the *Analysis UML Model* for this purpose, do not have to be modeled within the *Design UML Model*. As far as messages arriving at a subsystem's provided Interface occur as Triggers for a Protocol Transition within such a Protocol State Machine, a respective Receive Operation Event is modeled, referencing the respective Operation of the Interface, as depicted exemplarily within Figure 7.3.

Behavioral Specification of a Subsystem's Internal Decomposition The specification of the internal behavior of each subsystem, as captured via the *Initial* and *Consolidated Behavioral Subsystem Design Diagrams*, is basically achieved by adding respective Interactions as Owned Behavior to the subsystem Component. Life-

lines are used to represent the internally composed Parts as well as Ports of the subsystem. That is, in difference to those sequence diagrams, being developed during *Analysis Modeling*, *Lifelines* now specify a certain Property, which they represent, and Messages reference a corresponding Connector.

Messages arriving at a Port from the external environment of the subsystem are denoted by a formal *Gate*, from which a Message is then modeled to the Lifeline that represents this Port. All Messages arriving at a Port, even if originating from within the internal decomposition of a subsystem, have to specify respective Send Operation Events and Receive Operation Events to refer to the corresponding Operation of the Port's exposed Interfaces. Those Send Operation and Receive Operation Events are specified as Events of the Message Occurrence Specifications, which in turn serve as Message Ends of the *Messages*, as outlined by Figure 7.3. This is similar to what was modeled via the *Inter-Object Collaboration Diagrams* during *Analysis Modeling* in terms of Send Signal Events and *Receive Signal Events*. For those Messages however, targeting a Lifeline of an internal Part, this is not possible to refer to a corresponding Operation, as the Types of those Parts are not designed earlier than during *Detailed Design Modeling*. It has to be pointed out that this indeed is a violation of the well-formedness rules (i.e. the static semantics) of the UML (each Message Occurrence Specification indeed has to specify an Event, and each Send Operation Event and Receive Operation Event indeed has to reference a respective Operation). However, it is regarded to be only a temporal violation, as establishing those references is regarded to be subject to the *Detailed Design Modeling*, so that after the execution of this task, the *Design UML Model* is then consistent again.

The difference between the *Initial* and the *Consolidated Behavioral Subsystem Design Diagrams* is similar to that between the *Initial* and *Consolidated Behavioral Subsystem Interface Design Diagrams*. That is, *arguments* are additionally specified, so that the overall detail level is increased. However, in contrast to the *Consolidated Behavioral Subsystem Interface Design Diagram*, where another behavioral formalism is applied, here Instance Values and associated Instance Specifications are used instead of *Input Pins*.

Detailed Structural and Behavioral Specification of a Subsystem The *Detailed Structural Design* of a subsystem's internal decomposition is modeled in terms of *Structural Detailed Design Diagrams*. Here, besides those Classes typing the Parts of a subsystem's internal decomposition, which were already mentioned, Associations are designed, corresponding to the Connectors being modeled via the *Initial* and *Structural Subsystem Design Diagrams*. Their Association Ends, Properties being referenced as Defining Ends of the related Connector Ends, are modeled simultaneously as Owned Attributes of the involved Classes and as Member Ends of the respective Association.

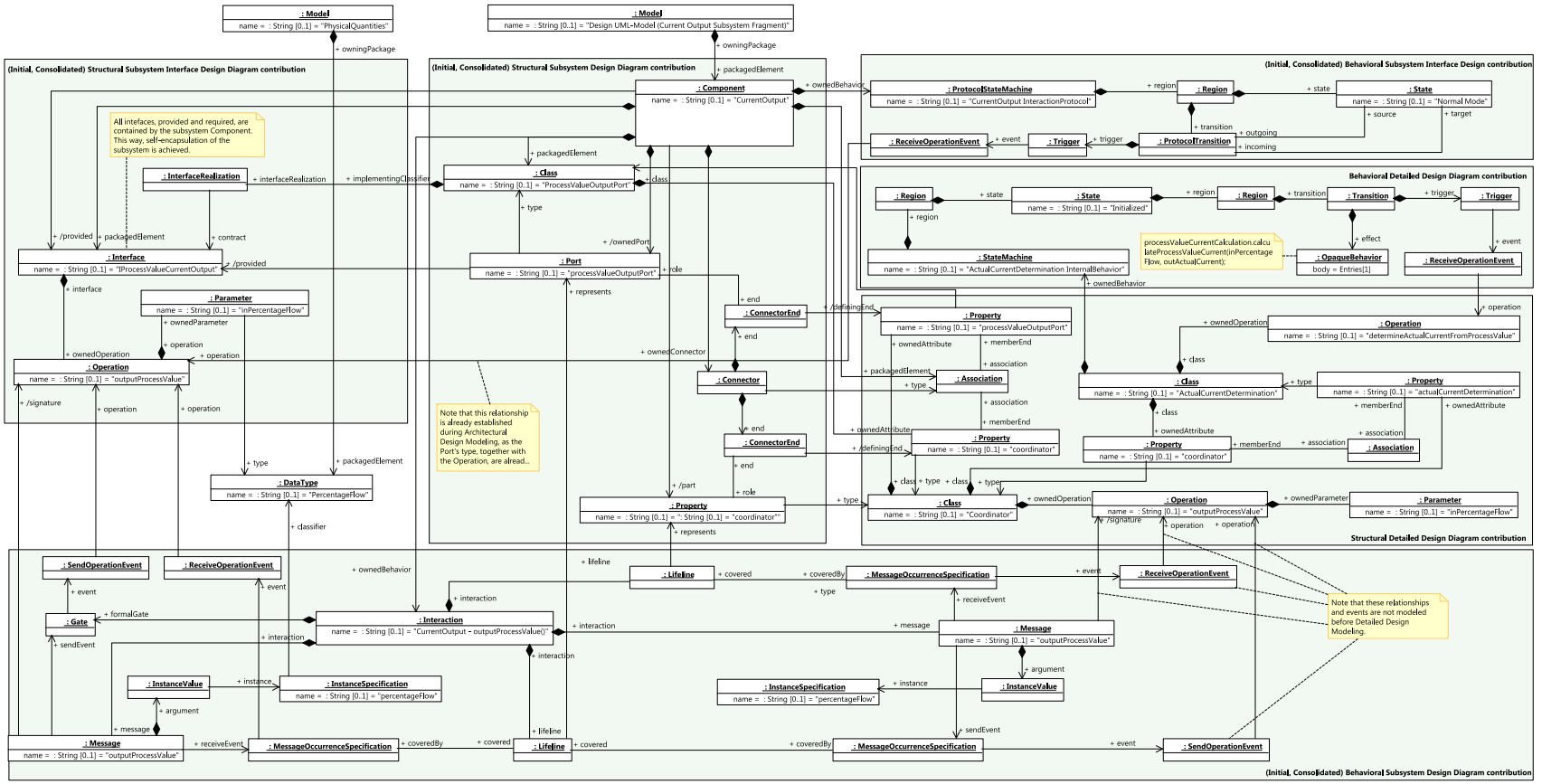


Figure 7.3: Examples of Key Artifacts contained in the Subsystem fragment of a MeDUSA Design UML Model

The *Detailed Behavioral Design* is captured in terms of a State Machine or a couple of Activities, being developed to depict internal behavior of important design objects, dependent on the concrete type of the *Behavioral Detailed Design Diagram*, which is employed for this purpose. In case a State Machine is used to depict the overall behavior of a design object, it is modeled as Classifier Behavior, i.e. Owned Behavior of the Class designed for that object during *Detailed Structural Design Modeling*. An example for this is depicted by Figure 7.3. In case one or more Activities are employed to describe certain behavioral aspects, they are modeled as Methods, i.e. behavioral specifications, of the Classes' Owned Operations.

Further integration between the structural and behavioral specifications of an individual design object is achieved by referencing structural and behavioral features of the respective Class from within the State Machine or Activity. Receive Operation Events and Call Behavior Actions may for example be used as Triggers and Effects of Transitions within a State Machine, referencing respective Operations. Read Object Link End Actions may be employed to access the Structural Features of an Operation's owning Class in case an Activity is used. It may however also be the case that no further integration is modeled, which holds for example if Opaque Behaviors are used to specify detailed behavior (e.g. as Triggers or Effects of Transitions within a State Machine).

System Model-Fragment

The system-related model fragment, whose structure is exemplarily depicted by Figure 7.4, basically contains those artifacts needed to integrate the different subsystems to an overall system. Its central element, directly contained by the root Model, of course is a representation of the system itself, for which purpose a Component is employed. Subsystems are represented by different model elements, dependent on whether the structural or behavioral view on the system is depicted.

Structural Specification of System Decomposition From a structural perspective, as it is captured by the *Structural System Architecture Diagram*, which is a UML component diagram, the system is regarded to be a composite structure, where subsystems are thus represented as composed Properties, that is Parts. This is very similar to how the internal decomposition of each subsystem is modeled by means of *Initial* and *Consolidated Structural Subsystem Design Diagrams*. The only difference is that there *Properties* represented objects, while here indeed subsystem instances, are represented.

Accordingly, structural relationships are depicted by means of Connectors between the respective Properties. In difference to the *Initial* and *Consolidated Structural Subsystem Design Diagrams*, which developed as UML composite structure diagrams, here *Connectors* are explicitly denoted as assembly or delegation Connectors (via the Connector Kind). They are thus differently represented in the diagrams as well (the so called ball-and-socket notation is employed for assembly connectors to de-

pict that the relationship is established via the provided and required interfaces of the subsystem Components, as shown in Figure 6.27). Similar to the internal decomposition of a subsystem, Associations are developed as Types of the Connectors. In contrast to those Associations, being developed as part of the *Detailed Structural Design* however, all Association Ends are directly owned by the Association itself and not by the Port's typing Classes. The reason for this is that to preserve self-encapsulation of the subsystems, a direct reference of a Port's Class to that of another subsystem's Port's Type is not desirable.

The integration between the system-specific and subsystem-specific model fragments is realized via those Parts, which serve as representation of the subsystems in the system's internal decomposition. Being Typed Elements, those Parts refer to the subsystem Component, they represent, as their Type. Further, as depicted by Figure 7.4, all Connectors, being established between those Parts indeed refer to the respective Ports of the represented subsystem as involved Roles.

Behavioral Specification of System Decomposition The behavioral view on the system architecture, as developed by means of the *Behavioral System Architecture Diagram*, basically contributes those artifacts related to UML sequence diagrams, i.e. Interactions, Lifelines, and Messages, similar to the *Initial* and *Consolidated Behavioral Subsystem Design Diagrams*.

As in case of the *Initial* and *Consolidated Behavioral Subsystem Design Diagrams* and the *Initial* and *Consolidated Structural Subsystem Design Diagrams*, integration with the structural view is realized by specifying the represents properties of the Lifelines to refer to those Properties, representing the subsystem (instances), and by establishing references of all modeled Messages to those *Connectors*, via which they are exchanged. Further, integration towards the subsystem specific model fragments is achieved by referencing the Operations, being defined by the provided and required Interfaces of the subsystem Component's Ports, via Send Operation Events and Receive Operation Events from the message Ends (which are specified by means of Message Occurrence Specifications), as depicted by Figure 7.4.

In contrast to the *Initial* and *Consolidated Behavioral Subsystem Design Interfaces*, Gates are not represented within the *Behavioral System Architecture Diagrams*. The reason for this is that all Triggers are encapsulated by respective *trigger* objects, which are composed by the different subsystems, so that on the system level, all behavior actually originates from a respective subsystem (as there is no need for any external triggers).

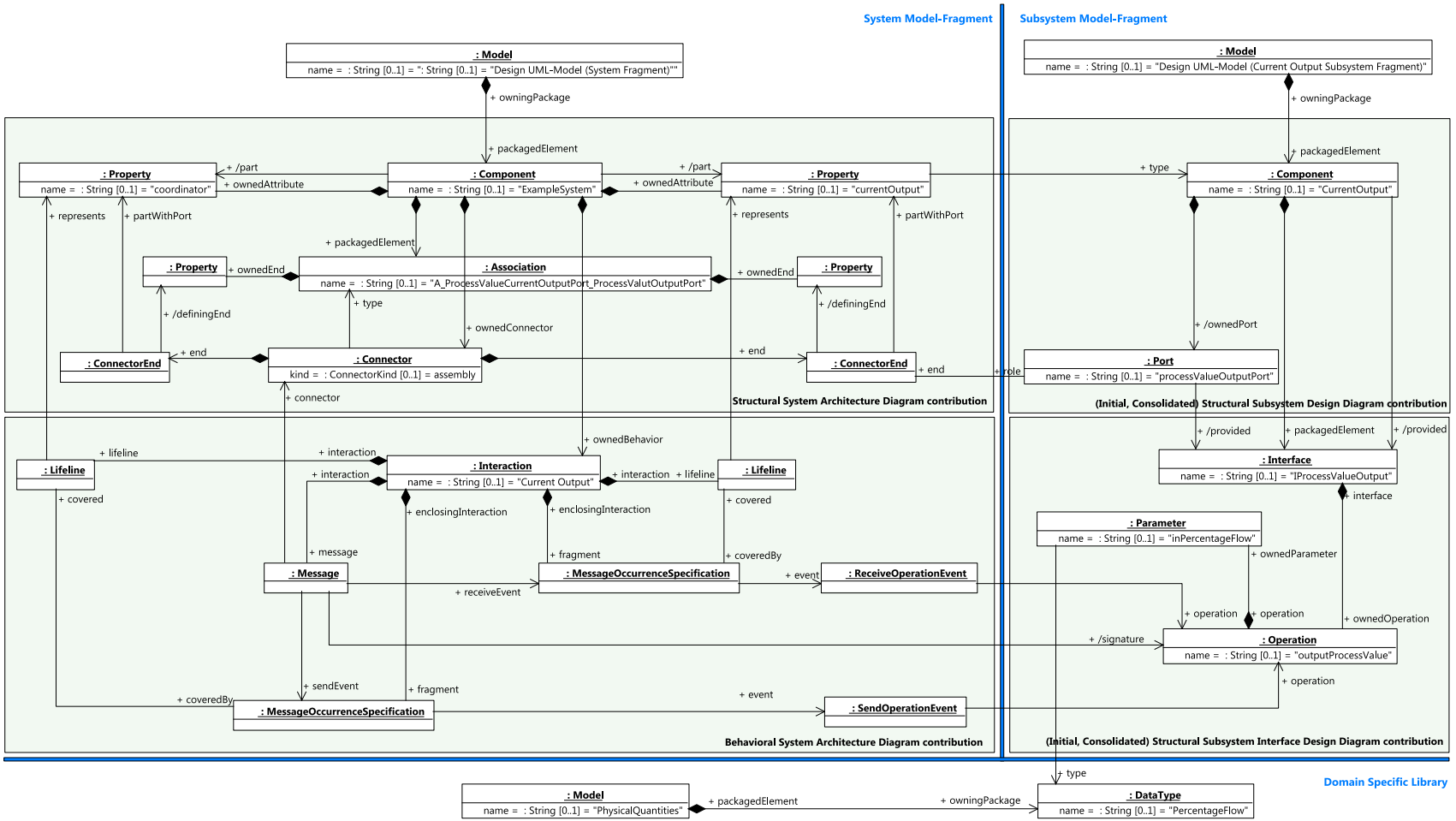


Figure 7.4: Examples of Key Artifacts contained in the System fragment of a MeDUSA Design UML Model

7.2 MeDUSA UML Profiles

As stated before, MeDUSA, and ViPER, its supporting tool, which is introduced in detail in the following chapter, are based on the UML as their underlying modeling language. Having originated from a unification of the *OOSE*, *OMT*, and *OOD* notations, the UML pretty much covers all basic concepts needed by the class-based MeDUSA method. However, the UML is not unrestrictedly applicable as notation for MeDUSA. By having originated from object-oriented approaches, where structural modeling was indeed pretty much performed on the level of *classes*, the UML was designed around the *classifier*[3] concept, and classifier instances - such as objects - are represented differently dependent on the respective structural or behavioral formalism that is applied, as properties, instance specifications, lifelines, input pins, output pins, or even others.

The result is a discrepancy in concepts between the *instance-driven* MeDUSA method and its underlying *Classifier*-based notation. *Objects*, which are more or less the central concept within MeDUSA are not directly mapped to a single UML concept, so that within MeDUSA's UML models they have to be represented by various modeling elements of different kinds. However, as repeatedly laid out, a methodology is only formed, if method(s), language(s), and tool(s) are seamlessly integrated by common concepts. What contributes to this as well is that the classification of actors and objects according to the *MeDUSA Actor Taxonomy* and *MeDUSA Object Taxonomy* respectively, can also not be adequately represented by the UML, as it does of course not reflect those MeDUSA specific categorizations. An extension of the UML by means of its build-in profile extension mechanism, thus is the best option to overcome these unattractive problems and guarantee a seamless integration.

A `Profile` defines a set of `Stereotypes`, which can be used to *stereotype*, i.e. classify, model elements (one speaks of the stereotype to be *applied*). Formally, a stereotype is defined as *"a kind of [Meta-]Class that extends [Meta-]Classes through Extensions"* [OMG07d]. That is, if a stereotype is defined to *extend* a meta-class, (an instance of) the stereotype can be applied to any instance of the respective meta-class. Stereotypes can further contain attributes, whose instance values are then referred to as *tagged values*. The major benefit of the profile extension mechanism is that it is a lightweight mechanism, which can be applied without loosing standard conformity. That is, because - in difference to the mechanisms offered by the *Meta-Object Facility (MOF)* [OMG06b] [OMG07c], used for the definition of the UML meta-model itself, the profile extension mechanism is not a *"first-class extension mechanism"*. That is it *"does not allow for modifying existing metamodels"*, but is more understood to be *"a straightforward mechanism for adapting an existing metamodel with constructs that are specific to a particular domain [..]"* [OMG07d]. UML models, to which a profile is applied, can thus still be developed with standard UML tools, while certain domain specific tools can extract and process the information, additionally kept in the model.

[3] *Classifier* is a super concept of *Class* and denotes a *"classification of instances"* (cf. [OMG07d]).

Besides better aligning the concept *worlds* of MeDUSA and the UML, the definition of MeDUSA specific UML profiles thus further allows to populate MeDUSA UML models with all additional information, which are needed for example for the real-time analysis that is continuously performed throughout the method, as well as with information, being additionally needed as input to a code generation tool.

7.2.1 Requirements UML Profile

As outlined before, the *MeDUSA Requirements UML Profile*, depicted by Figure 7.5, serves the primary purpose of reflecting the *MeDUSA Actor Taxonomy*, in order to enable modeling of non-functional timing and concurrency constraints, which are not expressible by means of the UML[4]. It thus defines an `Actor Stereotype`, which is specialized according to the *MeDUSA Actor Taxonomy* into `Trigger`, `Interface`, and `Constraint Actor` stereotypes.

In case of `Trigger Actors`, which are the single initiators of use cases and may never occur as secondary actors, it is reasonable to specify details about the timely occurrence of triggers. That is, in case of `Timer Actors`, a period may be specified, and further, whether the timing events occur only singular or on a regular basis (acyclic vs. cyclic). For `Eventer Actors` a minimum inter-arrival time of two adjacent events or a maximum arrival rate may be specified, in case the events occur in a bounded or bursty manner. If the occurrence of events is only describable by some statistical function, it is regarded to be unbounded. In case it cannot be anticipated at all, it is denoted as irregular. As a `Trigger Actor` may be the initiator of several associated use cases, priorities and deadlines of the triggered behavior are formalized by means of an `Initiation` stereotype, which is applied to each association between a `Trigger Actor` and related use cases.

`Interface Actors` in turn never occur as primary actors, as laid out before. They represent resources, which are accessed by one or more use cases. In case a hardware device or software system, which is represented as an `Interface Actor`, can only be accessed in a mutually exclusive way, this can be reflected by a respective tagged value within the corresponding stereotype. In either case, as it is reasonable to denote the duration of access that is attributed to a respective use case, an `Access` stereotype is defined, which has to be applied to all associations between use cases and `Interface Actors`.

To specify concurrency constraints in terms of synchronization or interference of concurrently executing use cases, a `Constraint Actor` may be employed. As depicted within Figure 7.5, such an actor may be denoted to depict a synchronization between two concurrently executed use cases in terms of some bidirectional communication, or some sort of message or data transfer (*send/receive*, *produce/consume*, or *write/read*), which may cause some synchronization or interference between the affected use cases (cf. [Rit08]).

[4]Neither the UML itself nor the upcoming UML Profile for MARTE [OMG07a] offer respective expressiveness, as sketched in [NL07b]

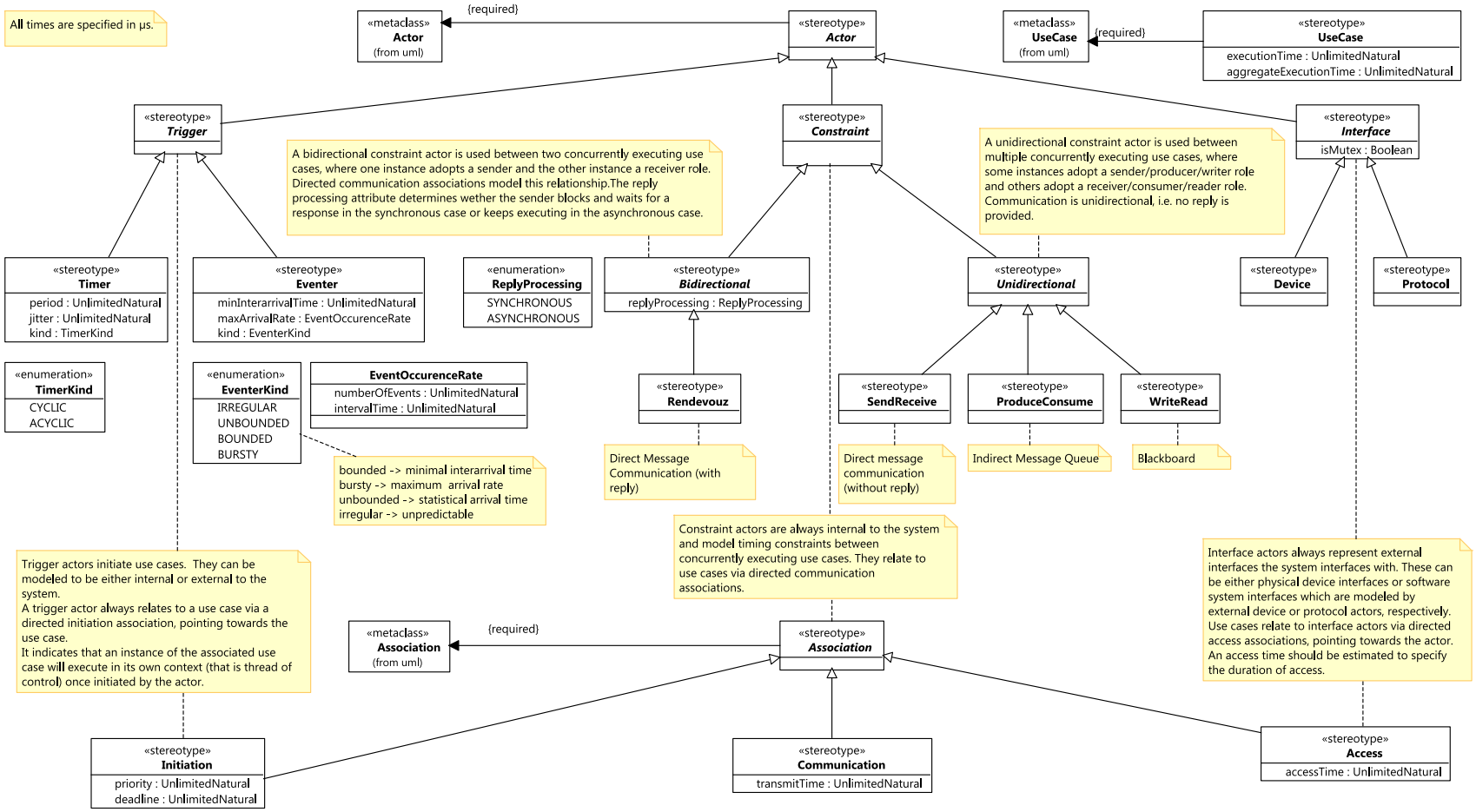


Figure 7.5: MeDUSA Requirements Profile

Besides having the possibility to specify and document respective real-time related requirements within a *Requirements UML Model*, the *Requirements UML Profile* further enables the execution of a *Preliminary Real-Time Analysis*, as proposed by the homonymously named MeDUSA discipline. For such an analysis, the execution time of each use case, i.e. the worst case execution time of all scenarios subsumed by it, has to be estimated and specified. A real-time analysis may then be performed as described within Section 6.2.2.6 (compare also [Rit08]).

7.2.2 Analysis UML Profile

Similar to how the MeDUSA *Requirements UML Profile* supports the classification of actors as specified by the *MeDUSA Actor Taxonomy*, the *Analysis UML Profile* supports the classification of objects according to the *MeDUSA Object Taxonomy* by specifying a corresponding hierarchy of `Object` stereotypes, which is depicted by Figure 7.6.

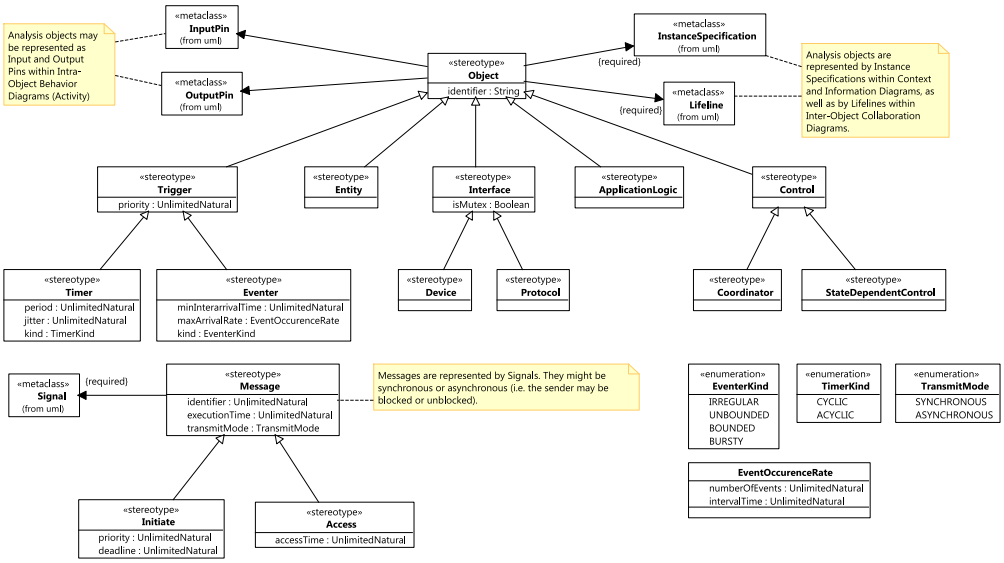


Figure 7.6: MeDUSA Analysis Profile

As there is - due to above outlined discrepancy in concepts between MeDUSA and the UML - no singular concept to represent an *analysis object* within an *Analysis UML Model*, the `Object` stereotype defines extensions to those UML meta-classes, which are used to depict *objects* within MeDUSA UML models, namely instance specifications, properties, input pins and output pins. To clarify that signals are used within a MeDUSA *Analysis UML Model* to denote *messages*, a respective `Message` stereotype is defined by the profile for the sake of clarity.

In contrast to the *Requirements UML Profile*, where a `Constraint Actor` stereotype is provided to address synchronization and interference issues, those issues are now addressed by means of respective messages. That is, reply messages, as subsumed

by a `Rendezvous Actor` can now be explicitly modeled as messages. To be able to specify the initiation of concurrently executing behavior and to denote shared access to resources, respective `Initiate` and `Access Message` stereotypes are nevertheless supported.

As the *Analysis UML Profile* has to enable an *Interim Real-Time Analysis*, execution times have to be provided as well. However, braking down the overall estimation for the `Use Cases`, they may now be attributed to individual `Messages`.

7.2.3 Design UML Profile

Similar to the *MeDUSA Analysis UML Profile*, the primary objective of the *MeDUSA Design UML Profile* is to support the classification of design objects according to the *MeDUSA Object Taxonomy*.

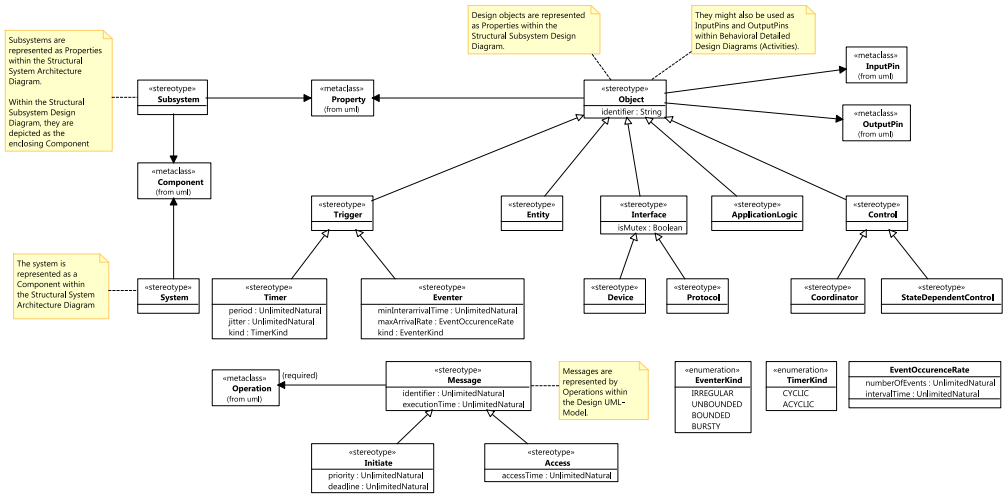


Figure 7.7: MeDUSA Design Profile

As indicated by Figure 7.7, unlike analysis objects, design objects are represented by properties (within the *Structural Subsystem Design Diagram*) rather than instance specifications. While design objects are further represented by lifelines within the *Behavioral Subsystem Design Diagrams*, the corresponding `Object` stereotype does not have to extend the UML meta-class `Lifeline`, as all lifelines specify a corresponding property, which they represent, and to which in turn the stereotype may be applied. That is, the property itself rather than the `Object` stereotype serves to integrate the different representations (within the *Analysis UML Model* this is objective of the respective `Object` stereotype, as properties are not specified within it). As *messages* are represented by operations within a *Design UML Model*, the respective `Message` stereotype is accordingly defined as an extension to the meta-class `Operation` rather than the meta-class `Signal`, as it is the case within the *Analysis UML Profile*.

Besides this, the *Design UML Profile* pretty much reflects the same properties than the *Analysis UML Profile*, as far as information, relevant to a *Consolidated Real-Time Analysis* is concerned. This is because both, the *Interim* as well as the *Consolidated Real-Time Analysis*, are technically performed in the same manner, only differentiated by the accuracy of the estimated values, which are used within the computations.

7.3 MeDUSA UML-to-ANSI-C Code Generation Schema

Only few methodical approaches explicitly address the transformation of models into source code. COMET [Gom00] for example, the direct predecessor of MeDUSA, does not address code generation at all. Similarly ROOM, the second major influencing approach, only provides a small example on how the implementation of a ROOM model may be performed within C++ [SGW94]. In fact, as far as publications related to model-based code generation can be found, they can mostly be related to the technical documentation of some modeling tool or some technical code generation framework. As a profound methodological integration between todays available methods and tools can mostly not be found (cf. Section 3.4), a discrepancy between what is offered by currently available modeling tools in terms of code generation and what is actually required from the viewpoint of the employed development method, is often observable.

In the context of UML-based ANSI-C code generation, as it is addressed within this methodology, the major problem is not only above quoted discrepancy, but indeed the fact, that the respective ANSI-C target language is to the very most extend simply not regarded at all by most of the available UML modeling tools. Java and C++ are the predominant programming languages, which have to be covered from a marketing perspective, and ANSI-C indeed seems to be of interest only in some marginal domains (even if the state-of-the-practice, as quoted within Section 3.3 seems to speak another language). In fact, out of the set of currently available commercial UML modeling tools, those actually supporting the generation of ANSI conformant C code, may just be a fistful. Telelogic's top dog *Rhapsody in C* has to be named, and *Poseidon Embedded Edition*, a tool jointly developed by Gentleware and Method Park may also be quoted as a positive example with its built-in *OO>C* code generator. However, besides the two, there does not seem to be much that is worth to be mentioned (cf. [FNL08]).

Having the defined goal of methodological integration and completeness, it is clear that within the MeDUSA-ViPER methodology, code generation has to be explicitly addressed in a way that no discrepancy between method and supporting tool, but instead a close integration is resultant. From a logical viewpoint, this involves the definition of a code transformation schema, which adequately reflects MeDUSA's requirements. From a technical perspective, it subsumes the realization of respective code generation tool as part of ViPER. While the latter will be accordingly addressed within the context of the ViPER tool in Chapter 8, the logical transformation schema, which is based on the code generation schema underlying the *OO>C* code generator (cf. [Gei02] and [Gen05]), as well as on own experiences (cf. [Fun06] and [Kev07]), will be outlined in the following.

7.3.1 General Transformation Strategy

The logical transformation of UML design models into ANSI-C source code may conceptually be regarded to involve two main concerns. On the one hand, the (possibly context dependent) transformation of UML model elements into syntactical ANSI-C language elements has to be addressed, and on the other hand, the logical organization of the resulting syntactical language elements into translation units, i.e. source code files. A seamless transition, as it is intended here, has to ensure that traceability is preserved in terms of both aspects. That is, a developer has to be enabled to easily follow the logical mapping of the individual model elements to syntactical language constructs, and it furthermore has to be ensured that the resulting source code fragments can be easily located within the translation unit structure.

Concerning both aspects, a general conformity between the *Type* concept, which is incorporated into both languages, can be taken advantage of. That is, within a UML input models, classifiers (which are actually types), as components, classes, interfaces, or associations, as well state machines, or interactions are the central model elements, around which a transformation may be logically organized. Transforming each classifier, contained in a *Design UML Model*, into a corresponding ANSI-C type, these types may then be also chosen to infer the logical organization of the resulting translation units.

While transforming classifiers into types seems to be a quite straightforward approach from a broad perspective, the specification of a generally applicable concrete transformation strategy, which is suited for arbitrary UML input models is a hard and tedious work, due to the overall expressiveness and complexity of the UML. Even when limiting transformation to a certain subset of classifiers, e.g. to the structurally relevant ones, as done in [Fun06], a generic transformation approach is only hard to manage. That is, due to the large number of different *usage scenarios*, a classifier may be employed in, either the transformation gets arbitrary complex, as it has to handle a lot of distinct cases, or the resultant source code gets quite complex, in case it is intended to be unaware of a concrete *usage scenario* and thus generally applicable. If for example the source code for a class is intended to be generated independent on certain assumptions about its usage in a respective input model, it has to reflect that a class may be used as simple type of an attribute or operation parameter, as type of a port, or even as a structured and encapsulated classifier with a complex internal structure.

Nevertheless, as the structure of a MeDUSA *Design UML Model* has been clearly defined (cf. Section 7.1), a transformation strategy may take this into account, so that neither the transformation schema, nor the resultant source code gets unnecessary complex. That is, by reflecting only the applied usages of classifiers within a MeDUSA *Design UML Model*, the transformation strategy may be restricted so that it is easier to understand and manage, and the resultant generated source code in turn gains an enhanced quality in terms of readability as well as performance, because it does not have to preserve a general applicability.

7.3.2 Classifiers within a MeDUSA Design UML Model

As already stated before, the code generation may be organized around the classifiers, which are employed in a MeDUSA *Design UML Model*. As a starting point, Figure 7.8 outlines all classifiers[5] defined by the UML, and highlights those, which are used in the context of a MeDUSA *Design UML Model*.

Out of these, the concrete (i.e. non abstract) classifiers that have to be transferred are:

- `Primitive Type`
- `Enumeration`
- `Data Type`
- `Class (from Kernel, from Communications)`
- `Interface`
- `Association`
- `Component`
- `State Machine`
- `Protocol State Machine`
- `Activity`
- `Interaction`
- `Opaque Behavior`

The differentiation of the *class* concept into its three merge increments, as they are defined within the UML language units *Structures*, *Communications*, and *Composite Structures*, is of course artificial, as these increments are only used internally for the definition of the single `Class` concept, which then subsumes the properties and characteristics of all three. However, the transformation of the internal structure and the externally visible properties of a structured and encapsulated (structured) class requires a dedicated transformation, which is avoidable in case of a non-structured class, whose transformation is pretty much comparable to that of a simple data type, only that additional information from potentially specified owned behavior has to be evaluated. As within a MeDUSA *Design UML Model*, (structured) classes are never used, as in fact, components are the single structured and encapsulated classifiers employed by the method, having the possibility to refer to non-structured classes explicitly, as done in the following, thus helps to simplify the explanation of the transformation significantly.

According to the comments provided in Section 7.1.3, within a *Design UML Model* above mentioned classifiers are used as depicted by Table 7.1. The list of course implies that a classifier is not used in different contexts simultaneously, which is a reasonable assumptions in case of a MeDUSA *Design UML Model*. The usage of a (non-structured) class as the type of a part within a subsystem's internal decomposition and at the same time as the type of a port for example, is neither necessary nor practical. The herein presented transformation strategy is thus based on the assumption that classifiers are only used in one of the outlined usage scenarios at a time.

[5] As the definition of the UML meta-model is heavily based on the concept of `Package Merge`, which - according to [OMG07d] - is *"a directed relationship between two packages that indicates that the contents of the two packages are to be combined [...] in the sense that the source element conceptually adds the characteristics of the target element to its own characteristics resulting in an element that combines the characteristics of both."*, Figure 7.8 also outlines the individual merge increments of a respective `Classifier`.

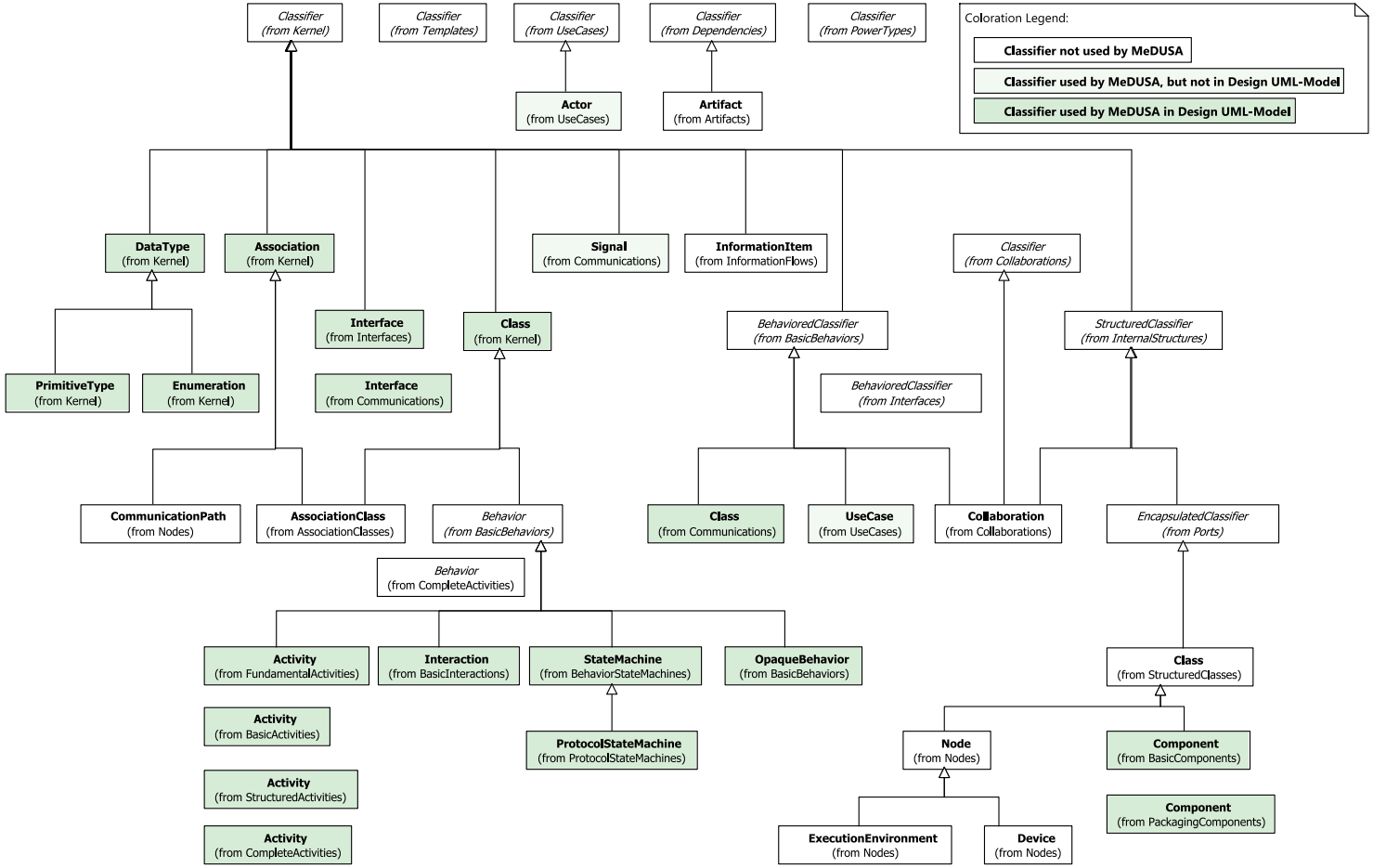


Figure 7.8: UML Classifiers Hierarchy, according to [OMG07d]

Component	• Specification of the system, singleton Structured and Encapsulated Classifier with internal structure and explicit external interfaces, directly contained by the system model fragment • Specification of a subsystem, singleton Structured and Encapsulated Classifier with internal structure and explicit external interfaces, directly contained by the subsystem model fragment, and as Type of a Part, representing a subsystem (instance) in the internal decomposition of (Non-Structured)the system Component
(Non-Structured) Class	• Type of a Part, representing an object in the internal decomposition of a subsystem Component • Type of Port, to specify (part of) the external interface of a subsystem Component • Type of an Attribute or Operation Parameter
Interface	• Exposed required or provided Interface of a Port, i.e. realized or used interface of a Port's typing (Non-Structured) Class
Association	• Type of a Connector in the internal decomposition of a subsystem Component, i.e. between (Non-Structured) Classes used as types of internally composed Parts and Ports • Type of a Connector in the decomposition of the system Component
Primitive Type, Enumeration, Data Type	• Type of an Attribute or Operation Parameter
Protocol State Machine	• Owned Behavior of a subsystem Component or a Port's typing (Non-Structured) Class, as behavioral specification of (parts of) the externally visible interfaces of a subsystem Component
Interaction	• Owned Behavior of a system or subsystem Component, as behavioral specification of its decomposition
State Machine, Activity	• Owned Behavior of a Part's typing Class, as specification of (internal) object behavior.
Opaque Behavior	• Effect of a Transition in the State Machine based specification of (part of the) object behavior. • Behavior of Call Behavior Action in the Activity based specification of (part of the) object behavior.

Table 7.1: Usage of Classifiers within a MeDUSA Design UML Model

7.3.3 Generating Folders and Translation Units

As already stated in [Fun06], it is practical to arrange the generation of a file system folder structure according to the coarse structure of the input model itself, as this way, ANSI-C representations of the UML model artifacts can be easily identified. As it can be inferred from the instance specifications provided in Section 7.1, a MeDUSA *Design UML Model* usually manifests itself as a set of components, representing the system as well as its subsystems, which are either contained in an integrated, single model file or within a set of distributed model fragments. Note that in either case, a direct containment relationship between the system component, which is in either case contained by the root model, and the subsystem components is not established, as the logical containment between the system and its subsystems is indeed modeled indirectly by specifying a composite internal structure for the system in terms of parts, which are typed by the respective subsystem components (cf. Figure 6.27). As the coarse structure of a *Design UML Model* is thus organized along the set of system and subsystem components, and as those components - in their role as *packaging components* - directly or indirectly contain all other elements, a file system folder structure may be best organized along the contained components.

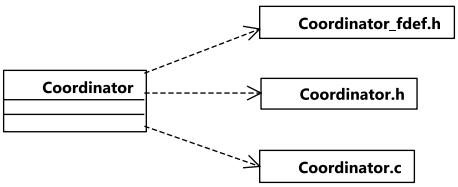


Figure 7.9: General Transformation of Classifiers into Translation Units

Based on it, all classifiers, including above mentioned components, but with the exception of primitive types and enumerations, which can be directly mapped to respective ANSI-C constructs, may then be transferred into a set of three *translation units*, containing the declaration of a respective ANSI-C type, related functions and macros (cf. [Gen05] and [Gei02]).

As depicted by Figure 7.9, those are a *forward declaration file* (`_fdef.h`), a *header file* (`.h`), and a *source file* (`.c`). The *header file* contains the declaration of the ANSI-C type, the classifier is transformed into, as well as the declaration of related macros and functions. The *forward declaration file* contains a forward declaration of the respective type, which is needed in case a recursive reference is required within its declaration. The *source file* finally contains the implementation of all functions, which of course depends on the respective classifier that is transformed and will thus be outlined individually in the following.

In this context, it has to be mentioned that, while the generation of a file system folder structure along the contained components is basically pretty much straight forward, the namespace-related characteristics of a component, namely the ability to provide a means for resolving composite names for its contained named elements (cf.

[OMG07d]), cannot be preserved, due to a lack of the concept in the ANSI-C language. Therefore the names of all syntactical ANSI-C elements have to be a fully qualified names, being prefixed by the concatenated names of all enclosing namespaces.

7.3.4 Generation of Syntactic Elements

The generation of syntactical ANSI-C language elements for the respective classifiers will be investigated in detail in the following. While it has already been mentioned that primitive types and enumerations somehow play an outstanding role within the transformation, what is why their transformation is addressed directly in advance, it has to be further anticipated that indeed not all of the aforementioned classifiers are actually transformed into a corresponding ANSI-C type. That is, due to reasons of performance and simplicity, associations and interfaces are for example only transformed indirectly, in the context of those classifiers, being affected by them. Interfaces are thus only indirectly transformed in the context of those classes, which are used as types of ports, associations are accordingly only transformed indirectly in the context of the associated classes. The same holds for opaque behaviors, which are used within a MeDUSA *Design UML Model* to specify guards and effects of a transition within a state machine, or to depict a set of instructions within an activity. They also do not have to be transferred into an own ANSI-C type but can here be transferred into code implicitly in the context of the state machines or activity respectively.

Transformation of Primitive Types

Both, the UML as well as the ANSI-C language support the concept of *primitive types*, which suggests that a transformation of UML primitive types into their ANSI-C equivalents is pretty much straight-forward. However, this is not the case. In fact, the UML supports only four predefined primitive types, namely `Integer`, `Boolean`, `String`, and `Unlimited Natural`, which were designed for its internal use, and which thus do not directly correspond to those primitive types of ANSI-C. Unlimited naturals are for example not covered within ANSI-C, while concepts to represent floating point numbers or individual characters are for instance not covered by the UML.

Therefore, using the predefined primitive types of the UML within a MeDUSA *Design UML Model* does not seem to be appropriate, as important implementation details could not be specified this way. Instead, as proposed by the UML specification itself, it is desirable to *"provide [...] own libraries of data types to be used when modeling with the UML"* [OMG07d]. What was already proposed in terms of a library of domain specific data types when describing the structure of MeDUSA's UML models in Section 7.1, can thus be as well applied to the handling of primitive types. To be concrete, a UML library as depicted in Figure 7.10, which defines UML equivalents of the built-in ANSI-C primitive types, may be rather used during modeling, and those primitive types, predefined by the UML, may thus be omitted.

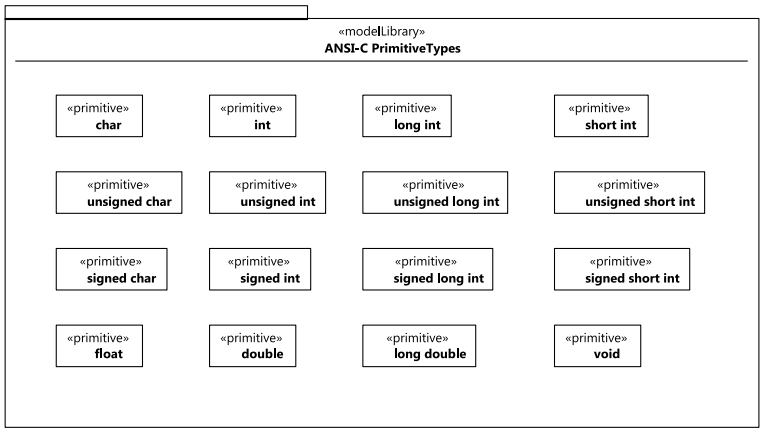
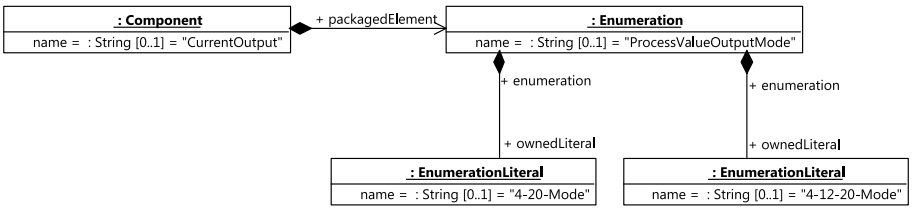


Figure 7.10: ANSI-C Primitive Types UML-Library

The transformation of those primitive types, defined by above sketched library is then pretty much straight-forward. In fact, as all these primitive types actually directly correspond to built-in ANSI-C types, no generation of types is needed at all. Instead, the name of the type only has to be used when generating the ANSI-C representation of each typed element, that is attribute, operation, or parameter, which specifies it.

Transformation of Enumerations

As the ANSI-C language supports `enumeration constants`, mapping enumerations is pretty much straight-forward. That is, as denoted by Figure 7.11b for the `ProcessValueOutputMode` example depicted by Figure 7.11a, an enumeration may be directly transformed into a respective `enum` declaration, where all UML enumeration literals are directly mapped to corresponding enumeration values.



(a) Instance Specification: Enumeration with Owned Literals

Figure 7.11: Example - Transformation of Enumeration

Because no recursive declaration is allowed and because no additional definitions are subsumed by the transformation of an enumeration, a *forward declaration* and *source file*, as generally needed for all other classifiers, is indeed not needed in case of an enumeration, so they play a rather outstanding role to this extend.

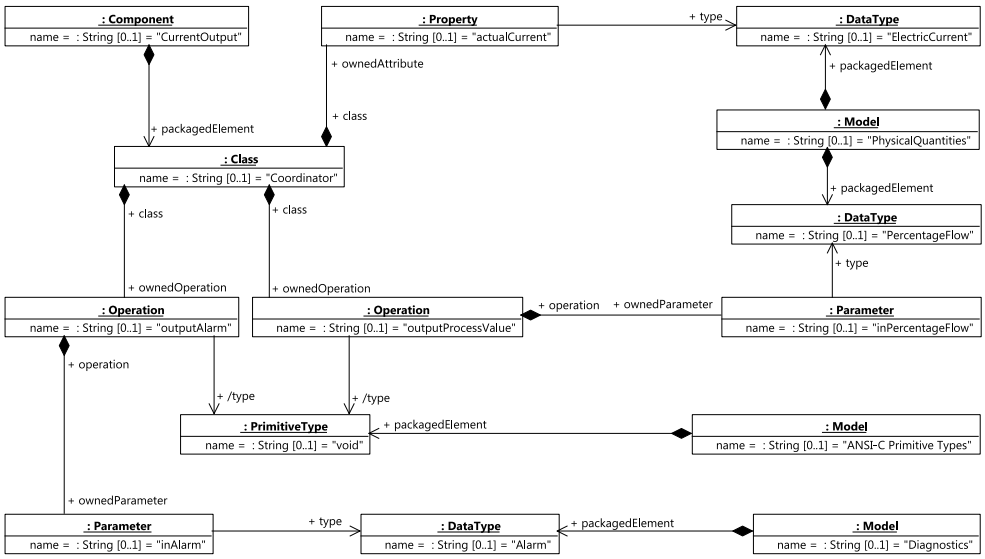
```
/************************************************************
* File: ProcessValueOutputMode.h
************************************************************/
...

/*  Enumeration Declaration */
enum CurrentOutput_ProcessValueOutputMode {
  4-20-Mode,
  4-12-20-Mode
};

...
```

(b) Generated Code: Corresponding Struct Declaration

Figure 7.11: Example - Transformation of Enumeration

General Transformation of Data Types and (Non-Structured) Classes

The ANSI-C equivalent chosen for all classes and data types, as well as for all other classifiers, is a `struct`, whose name has to match the fully qualified name of the classifier. In case of the `Coordinator` example `Class`, depicted by Figure 7.12a, a `struct` could for instance be generated as depicted by Figure 7.12b.



(a) Instance Specification: Simple Class with Attributes and Operations

Figure 7.12: Example - Transformation of Data Type and (Non-Structured) Class

The creation and destruction of `struct` instances is realized - inspired by how its is done in the context of object-oriented languages - by dedicated constructor and destructor functions, which are parameterized for this purpose with a respective `struct pointer`, as depicted by Figure 7.12b. Even if not explicitly specified, a classifier is assumed to have at least an implicit default constructor and destructor, so respective functions are always created together with the struct declaration.

```
/************************************************************
* File: Coordinator_fdef.h
************************************************************/
...

/* Classifier Struct Forward Declaration */
struct _CurrentOutput_Coordinator;

#define Example_System_CurrentOutput_Coordinator \
  struct _CurrentOutput_Coordinator
...
```

```
/************************************************************
* File: Coordinator.h
************************************************************/
...

/* Classifier Struct Declaration */
struct _CurrentOutput_Coordinator {
  ...
};

/* Constructors and Destructors */
void CurrentOutput_Coordinator_create(CurrentOutput_Coordinator* self);

void CurrentOutput_Coordinator_destroy(CurrentOutput_Coordinator* self);

...
```

```
/************************************************************
* File: Coordinator.c
************************************************************/
...

void CurrentOutput_Coordinator_create(CurrentOutput_Coordinator* self) {
  ...
}

void CurrentOutput_Coordinator_destroy(CurrentOutput_Coordinator* self) {
  ...
}

...
```

(b) Generated Code: Corresponding Struct and Constructor & Destructor Function Declarations

Figure 7.12: Example - Transformation of Data Type and (Non-Structured) Class

Within the declaration of the `struct`, the attributes of a classifier are directly transformed into nested declarations of `struct members`, as depicted by Figure 7.12c. While the type and multiplicity of the attributes can be directly represented by declaring an `array member` of a corresponding type, the visibility of an attribute cannot be directly transferred, as access to the members of a struct is always public. It can thus only be *emulated* by some sort of usage convention. That is, for each attribute that is publicly visible[6], a respective *selector* function can be defined, which - by convention - guards access to the property. That is, all attributes should only be accessed via the respective selector function and not directly by accessing the struct.

The transformation of the classifier's operations is done in a straight-forward manner by transferring them into respective functions, as shown in Figure 7.12d. In contrast to object-oriented programming languages, where the encapsulation principle is directly incorporated into the *class* concept, so that operations and related variables are en-

[6] A clean differentiation of the different UML defined visibility kinds, i.e. *public*, *protected*, *package*, or *private* is not realizable, so all attributes of other than private visibility are regarded to be public

```
/************************************************************
* File: Coordinator.h
************************************************************/
...

/*  Classifier Struct Declaration */
struct _CurrentOutput_Coordinator {
  /* Owned Attributes */
  PhysicalQuantities_ElectricCurrent actualCurrent[1];
  ...
};

/* Attribute Selectors */
PhysicalQuantities_ElectricCurrent*
  CurrentOutput_Coordinator_actualCurrrent(CurrentOutput_Coordinator* self);

...
```

```
/************************************************************
* File: Coordinator.c
************************************************************/
...

/* Attribute Selectors Implementation*/
PhysicalQuantities_ElectricCurrent*
  CurrentOutput_Coordinator_actualCurrent(CurrentOutput_Coordinator* self) {
    return *self->actualCurrent;
}

...
```

(c) Generated Code: Member and Selector Function Declarations, corresponding to Owned Attributes

```
/************************************************************
* File: Coordinator.h
************************************************************/
...

/* Owned Operations */
ERROR_CODE CurrentOutput_Coordinator_outputProcessValue(
  CurrentOutput_Coordinator* self,
  PhysicalQuantities_PercentageFlow inPercentageFlow);

ERROR_CODE CurrentOutput_Coordinator_outputAlarm(
  CurrentOutput_Coordinator* self,
  Diagnostics_Alarm inAlarm);

...
```

```
/************************************************************
* File: Coordinator.c
************************************************************/
...

/* Owned Operations Implementation */
ERROR_CODE CurrentOutput_Coordinator_outputProcessValue(
  CurrentOutput_Coordinator* self,
  PhysicalQuantities_PercentageFlow inPercentageFlow) {
  ...
}

ERROR_CODE CurrentOutput_Coordinator_outputAlarm(
  CurrentOutput_Coordinator* self,
  Diagnostics_Alarm inAlarm) {
  ...
}

...
```

(d) Generated Code: Function Declarations for Owned Operations

Figure 7.12: Example - Transformation of Data Type and (Non-Structured) Class

capsulated in a single unit, in the context of a procedural programming language as C, functions and related variables are pretty much unrelated. As a consequence, each

generated `function` has to be parameterized with the `struct` instance, whose data is being accessed, as only thereby the context of the `function call` can be clearly determined. Each non static `Operation` is thus transformed into a `function`, which defines its owning `Classifier`'s `struct` representation as a first parameter. All other parameters of the corresponding `function` are then created to match those of the represented `Operation`. In case of non primitive `Parameter Types`, `pointers` have to be used, as shown in Figure 7.12d, as a *call-by-reference* concept is not supported in other cases, and parameters would otherwise be passed in *by-value* instead. Similar to attributes, the visibility of an operation cannot be adequately represented within ANSI-C. However, a differentiation of public and private operations can be realized by declaring the corresponding functions within the *header file* or *source file* respectively, again treating all non-private operations implicitly as public.

The transformation of classifiers usually also leads the generation of a number of private as well as technically motivated support functions, which are actually not accounted as part of the transformed classifier's public interface. To overcome this, a general programming interface that matches the classifier's public interface, has to be offered. A *facade*, which only contains the *interface related* functions and hides all technical support functions, as well as those functions, representing non public operations, is desirable. As outlined by Figure 7.12e, such a facade can be easily realized as a *macro facade* (cf. [Gen05] and [Gei02]), which consists of macro definitions, serving as shortcuts to the publicly visible functions, and thus ensures that no resources are wasted, as the pre-processor will replace each macro usage with the respective call to the referenced public function. The macros can be generally split into three groups, namely shortcuts to the constructors and destructors, which are prefixed `CREATE` and `DESTROY`, shortcuts to the selectors of the public attributes, being prefixed as `ATTRIBUTE`, as well as shortcuts to the public operations, prefixed as `OPERATION`.

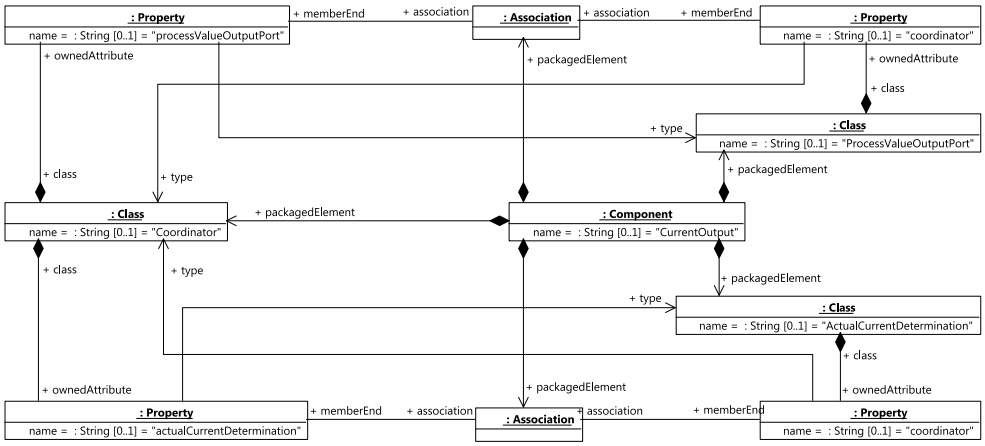
```
/*************************************************************
* File: Coordinator.h
*************************************************************/
...

/* Macro Facade */
#define CREATE_CurrentOutput_Coordinator(self) \
  CurrentOutput_Coordinator_create(self);

#define DESTROY_CurrentOutput_Coordinator(self) \
  CurrentOutput_Coordinator_destroy(self);

#define ATTRIBUTE_CurrentOutput_Coordinator_actualCurrent(self) \
  CurrentOutput_Coordinator_actualCurrent(self);

#define OPERATION_CurrentOutput_Coordinator_outputProcessValue(self, inPercentageFlow) \
  CurrentOutput_Coordinator_outputProcessValue(self, inPercentageFlow)

#define OPERATION_CurrentOutput_Coordinator_outputAlarm(self, inAlarm) \
  CurrentOutput_Coordinator_outputAlarm(self, inAlarm)

...
```

(e) Generated Code: Macro Facade as Public Classifier Interface

Figure 7.12: Example - Transformation of Data Type and (Non-Structured) Class

Additions for (Non-Structured) Classes typing Parts, Implicit Transformation of Associations as Part of a Subsystem's Internal Decomposition

In contrast to those classes, which are used as types of simple attributes or operation parameters, those used to type parts within the internal decomposition of a subsystem component, which do thus indeed represent objects and not just simple values, furthermore have to reflect that their instances are structurally related to other parts or ports. In correspondence to the general transformation principles, which were outlined before, those associations, which are used as types of connectors within a subsystem's internal decomposition, are not transferred into corresponding ANSI-C types, but are only indirectly created in the context of the respective associated classes, due to reasons of simplicity and performance.



(a) Instance Specification: A Class typing a Part (Subsystem Decomposition)

```
/************************************************************
* File: Coordinator.h
************************************************************/
...

/* Classifier Struct Declaration */
struct _CurrentOutput_Coordinator {
 ...
 /* Association Ends */
 CurrentOutput_ProcessValueOutputPort processValueOutputPort[1];
 CurrentOutput_ActualCurrentDetermination actualCurrentDetermination[1];
 ...
};

...
```

(b) Generated Code: Member Declarations corresponding to Association Ends

Figure 7.13: Example - Transformation of (Non-Structured) Class typing Part

As depicted by Figure 7.13 in case of the `Coordinator` example class, this is realized by adding respective `member` declarations, representing the respective ends of all outgoing associations, to the struct declarations of the associated classes. The initialization of these `members`, that is the *wiring* of the interconnected parts and ports, has to be done from within the initialization of the enclosing subsystem component, and will as such be described in detail in the context of the transformation of subsystem components.

Additions for (Non-Structured) Classes typing Ports, Implicit Transformation of External Interfaces and Associations within the System Internal Decomposition

As required interfaces are directly *realized* by the port's typing class, that is the class offers corresponding operations to that of the interface, all interface operations can be directly transformed into respective functions within the port's type. The implementation of these functions, as depicted by Figure 7.14, can be generated to simply delegate all calls to either the enclosing component's instance or to an internal part, dependent on whether the port is a behavior port or not[7].

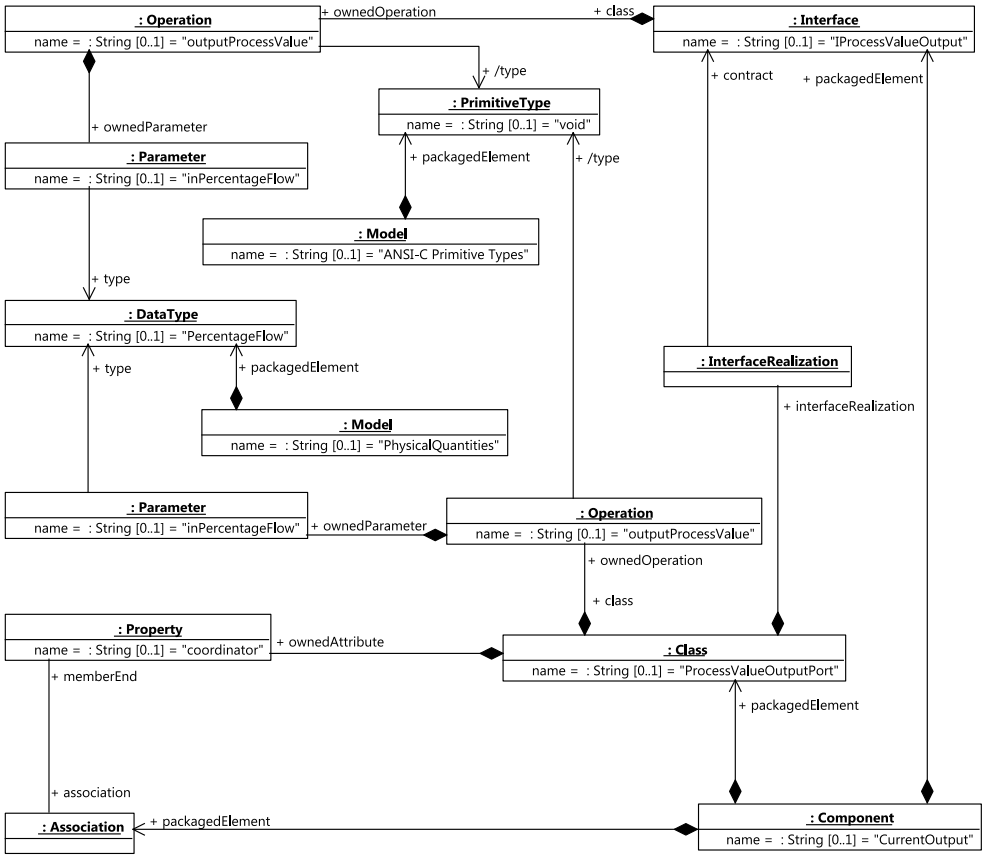
As those classes, being used as types of ports can - as far as associations towards the internal decomposition of the owning subsystem component is concerned - be transferred into ANSI-C code in the same manner as those classes typing parts, this can - in case of non-behavior ports - be realized by accessing the respective `struct members`, which represent outgoing association ends. In case a port is a behavior port, i.e. it forwards any requests to its enclosing subsystem component, the class used as its type does not have any associations to parts within the internal decomposition. In such a case, the `struct` generated for the port's typing class instead has to declare a `member` to refer to the enclosing subsystem component instance.

Similar as in case of provided interfaces, all operations defined by a required interface, have to be offered by the port's typing class as well. While this sounds astonishing at first, it gets rather obvious when envisioning that the ports serve as sort of a *facade*, which guards all calls from the internal decomposition of a subsystem component to its external environment, as due to this, the port's type has to offer respective operations that can be called from internal parts for this purpose.

While each `Operation` of a required `Interface` may thus be transformed into a `function` in a way similar to as it is done for `Operations` of a provided `Interface`, the generated implementation of course has to differ, as it has to actually realize the *wiring* to a respective provided `Interface` in the external environment of the subsystem `Component`. Here, handling those associations, which are used as types of connectors within the system's internal decomposition in terms of above specified transformation (using struct members to represent the association ends) would not be adequate, as this would result in a strong coupling of the subsystem components, violating the self encapsulation property, which is indeed critical for their reuse.

Instead, as *wiring* between subsystems has to be performed only indirectly in terms of their required and provided `Interfaces`, the generated source code has to adequately represent this indirection as well. That is, the implementation of each required operation has to delegate its calls to a respective target implementation of the matching provided interface, as specified within the system's internal decomposition, thereby guaranteeing the needed indirection, which is necessary to preserve the self-encapsulation property of the subsystem `Component`.

[7] According to [OMG07d], *"a port has the ability to specify that any requests arriving at this port are handled by the behavior of the instance of the owning classifier, rather than being forwarded to any contained instances, if any."* Such ports are referred to as `Behavior Ports`.



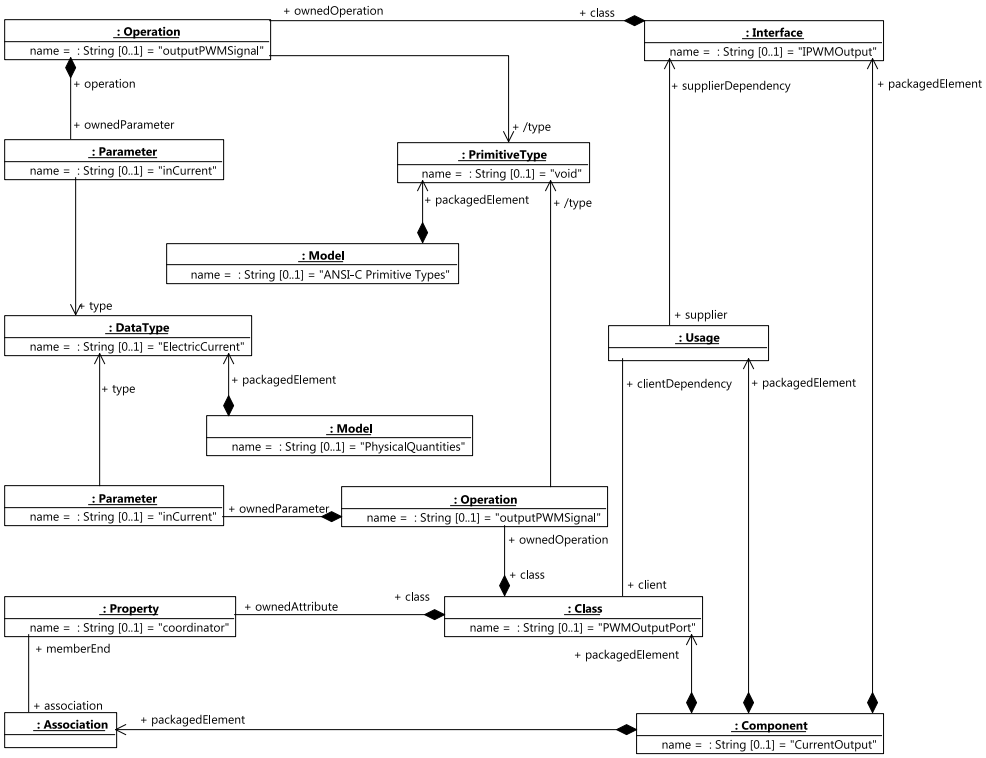
(a) Instance Specification: Class typing Port (Provided Interface)

```
/************************************************************
* File: ProcessValueOutputPort.c
************************************************************/
...

ERROR_CODE CurrentOutput_ProcessValueOutputPort_outputProcessValue(
  CurrentOutput_ProcessValueOutputPort* self,
  PhysicalQuantities_PercentageFlow* inPercentageFlow){
    return CurrentOuput_Coordinator_outputProcessValue(self->coordinator, inPercentageFlow);
}
...
```

(b) Generated Code: Function Declarations corresponding to Provided Interfaces

Figure 7.14: Example - Transformation of (Non-Structured) Class typing Port (Provided Interface)

For this purpose, a set of function pointers, representing the required interface operations, as well as a `pointer`, referencing the target implementation of the interface, on which the functions may be invoked, are additionally generated into the port's typing class, as depicted by Figure 7.15b. If these function pointers and the implementation pointer are correctly initialized to point to the respective target implementations, which is done in the context of the enclosing system component's initialization and will thus be documented in the context of transformation of components, the implementation of the interface functions within the port's class may then delegate their calls by using the function pointers, as depicted by Figure 7.15b.



(a) Instance Specification: Instance Specification: Class typing Port (Required Interface)

```
/*************************************************************
* File: PWMOutputPort.h
*************************************************************/
...
/* Classifier Struct Declaration */
struct _CurrentOutput_PWMOutputPort {
  ...
  /* Required Interfaces */
  void* iPWMOutput;
  ERROR_CODE (* iPWMOutput_outputPWMSignal)(void* iPWMOutput, PhysicalQuantities_ElectricCurrent inCurrent);
  ...
};

...
```

```
/*************************************************************
* File: PWMOutputPort.c
*************************************************************/
...
ERROR_CODE CurrentOutput_PWMOutputPort_outputPWMSignal(
  CurrentOutput_PWMOutputPort* self,
  PhysicalQuantities_ElectricCurrent* inCurrent){
  return (self->iPWMOutput_outputPWMSignal)(self->iPWMOutput, inCurrent);
}

...
```

(b) Generated Code: Function Pointer and Struct Pointer Member Declarations corresponding to Required Interfaces

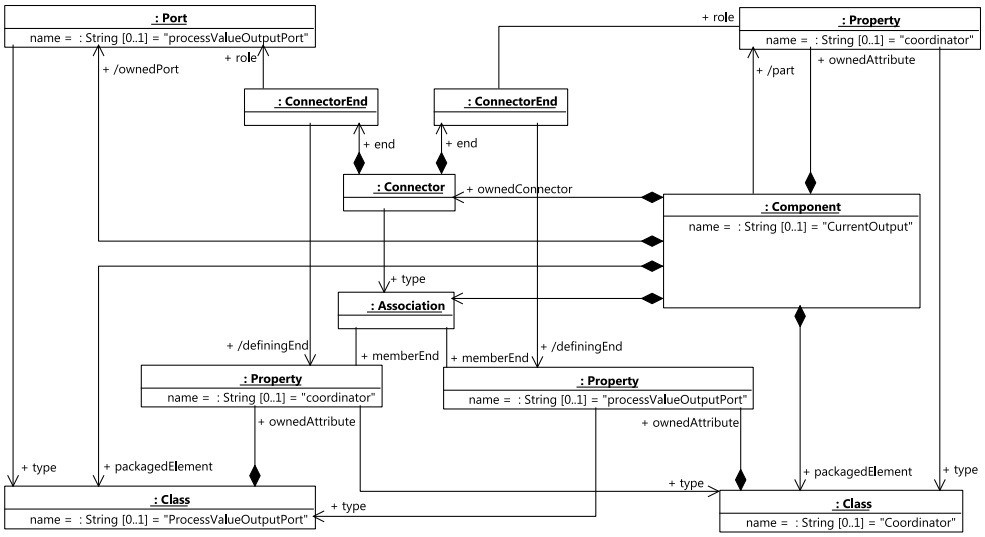
Figure 7.15: Example - Transformation of (Non-Structured) Class typing Port (Required Interface)

Transformation of Components

Within a MeDUSA *Design UML Model*, components are used in two ways, namely to represent the system as well as its subsystems. While both scenarios have in common that components are used in their role as encapsulated and structured classifiers, there are slight differences. The system component does for example not own any ports, as the system's external interfaces are in fact encapsulated by respective *trigger* and *interface* objects (subsumed as parts by a respective subsystem component). Further, the system component's internal structure manifests itself in terms of parts, whose types are components, so they indeed represent subsystems rather than simple objects. In contrast to this, subsystem components own ports, which describe their externally visible interfaces, as well as parts, now indeed representing design objects, so they are typed by simple non-structured classes.

Besides having to represent the composed parts and ports, which is done by means of respective `struct members`, similar to as it is done with attributes in terms of non-structured classes, the code, generated for a component thus further has to reflect that the *wiring* of its composed properties is done differently, depending on whether the component represents the overall system or just one of its subsystems. That is, as outlined before, parts composed by a subsystem component are directly connected to each other via their respective association ends, while parts composed by the system component are indirectly wired via their exposed interfaces, as described before.

That is, in terms of a subsystem component, initializing the respective `struct members`, representing the association ends within the ports' and parts' struct, have to be initialized as depicted by Figure 7.16. In case of the system component, the situation is comparable but yet different. Again, `struct members`, which represent its composed parts, which are now component instances, have to be initialized. However, *wiring* now has to ensure that the `function pointers` and the implementation `pointer`, representing the required interfaces of the components ports, have to be initialized with the respective `struct` instances and `functions` of the respective opposite ports, which offer the compatible provided interfaces, as outlined by Figures 7.17a and 7.17b.



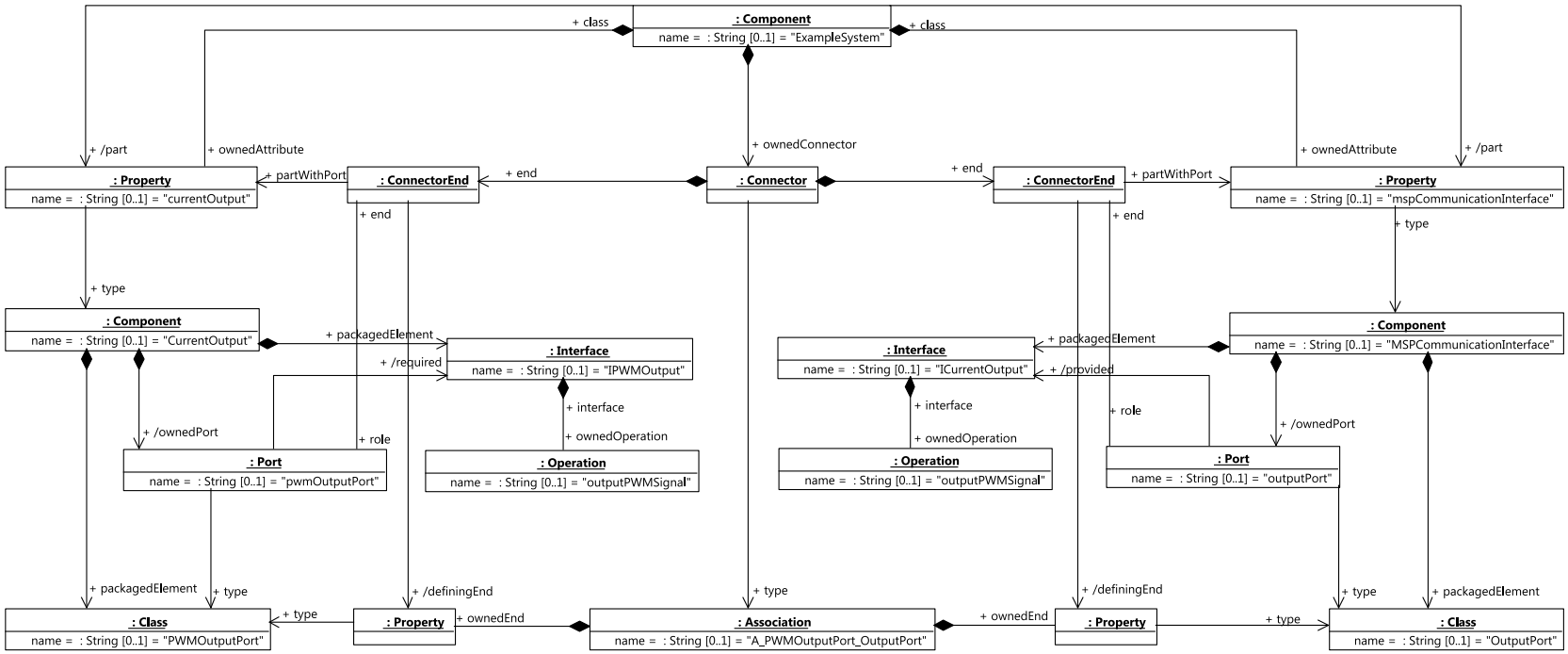
(a) Instance Specification: Subsystem Component

```
/************************************************************
* File: CurrentOutput.h
************************************************************/
...

/* Classifier Struct Declaration */
struct _CurrentOutput {
  /* Parts */
  CurrentOutput_Coordinator coordinator[1];
  ...
  /* Ports */
  CurrentOutput_ProcessValueOutputPort processValueOutputPort[1];
  ...
};

...
```

```
/************************************************************
* File: CurrentOutput.c
************************************************************/
...

void CurrentOutput_create(CurrentOutput_Coordinator* self) {
  /* Create Parts */
  CREATE_CurrentOutput_Coordinator(&self->coordinator[0]);
  ...
  /* Create Ports */
  CREATE_CurrentOutput_ProcessValueOutputPort(&self->processValueOutputPort[0]);
  ...
  /* Wiring */
  self->coordinator[0].processValueOutputPort[0] = &self->processValueOutputPort[0];
  self->processValueOutputPort[0].coordinator[0] = &self->coordinator[0];
  ...
}

...
```

(b) Generated Code: Struct Member Declarations for Parts & Ports, Wiring

Figure 7.16: Example - Transformation of subsystem Component



(a) Instance Specification: System Component

Figure 7.17: Example - Transformation of system Component

```
/*************************************************************
* File: ExampleSystem.h
*************************************************************/
...

/* Classifier Struct Declaration */
struct _ExampleSystem {
 /* Parts (Subsystems) */
 CurrentOutput currentOutput[1];
 MSPCommunicationInterface mspCommunicationInterface[1];
 ...
};

...
```

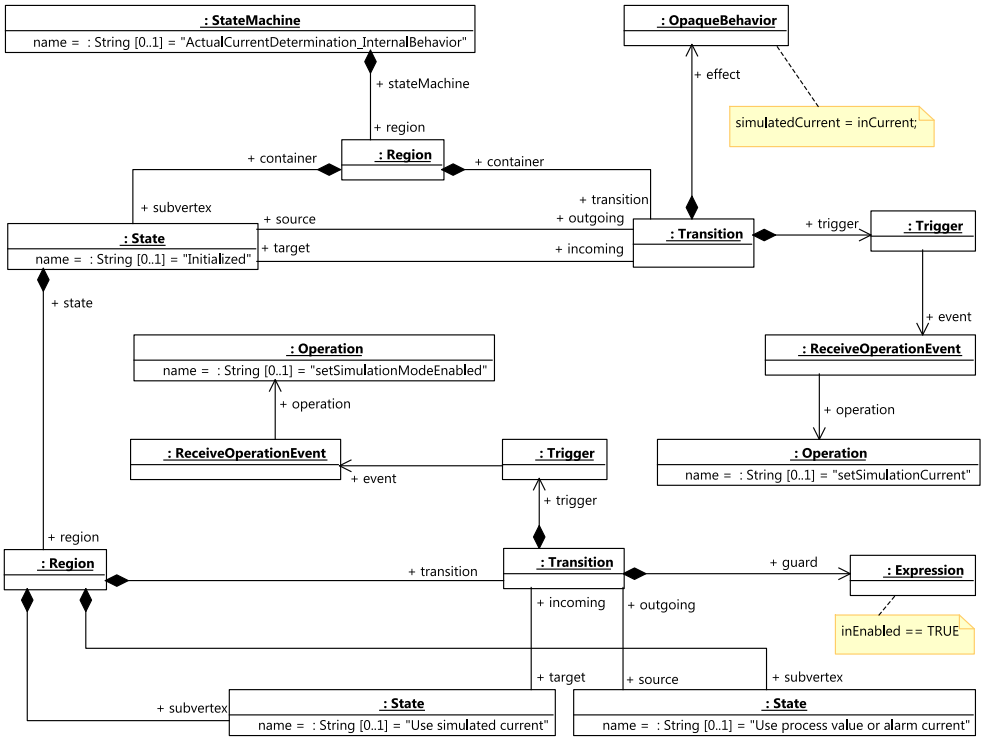
```
/*************************************************************
* File: ExampleSystem.c
*************************************************************/
...

void ExampleSystem_create(ExampleSystem* self) {
  /* Create Parts (Subsystems) */
  CREATE_CurrentOutput(&self->currentOutput[0]);
  CREATE_MSPCommunicationInterface(&self->mspCommunicationInterface[0]);
  ...

  /* Wiring (Required Interfaces) */
  self->currentOutput[0].pwmOutputPort[0].iPWMOutputPort =
    &self->mspCommunicationInterface[0].outputPort[0];
  self->currentOutput[0].pwmOutputPort[0].iPWMOutputPort_outputPWMSignal =
    &MSPCommunicationInterface_OutputPort_outputPWMSignal();
  ...
}

...
```

(b) Generated Code: Struct Member Declarations for Parts, Wiring of Parts within Constructor Implementation

Figure 7.17: Example - Transformation of system Component

General Transformation of State Machines

There are several possibilities to transform a state machine into respective ANSI-C code, reaching from simple solutions using nested `switch` statements up to profound, optimized solutions based on compressed `function pointer tables` (cf. [Gei02]). While there are significant differences in the performance and readability of the generated source code, the basic principle behind all of them is to generate a data structure to hold the current state of the state machine, together with a set of functions, which represent the state machine's transitions as well as their guards and effects.



(a) Instance Specification: State Machine

Figure 7.18: Example - Transformation of a State Machine

Based on this, a simple transformation of a state machine may thus be performed as depicted by Figures 7.18a, 7.18b and 7.18c. That is, as with all other previously outlined classifiers, a `struct` is defined, containing a `member` to hold the current state of the state machine. All states, as well as all events, being referenced from triggers of the state machine's transitions are defined as `symbolic constants` within the *header file*. Further, a single function is defined to trigger a transition of the state machine via a respective event.

The actual state machine logic is captured in terms of a *transition table*, which might be realized as a two-dimensional array of function pointers. It specifies which transition can be taken in a respective state on the occurrence of a respective event by containing a function pointer to a respective *transition function* (one `function` is created for each

```
/************************************************************
* File: ActualCurrentDetermination_InternalBehavior.h
************************************************************/

/* (Flattened) States */
#define CurrentOutput_ActualCurrentDetermination_InternalBehavior_\
       STATE_Initialized_UseProcessValueOrAlarmCurrent  0x001
#define CurrentOutput_ActualCurrentDetermination_InternalBehavior_\
       STATE_Initialized_UseSimulatedCurrent      0x002
...

/* Events */
#define CurrentOutput_ActualCurrentDetermination_InternalBehavior_\
       RECEIVE_setSimulationModeEnabled           0x001
#define CurrentOutput_ActualCurrentDetermination_InternalBehavior_\
       RECEIVE_setSimulationCurrent               0x002
...

/* Classifier Struct Declaration */
struct _CurrentOutput_ActualCurrentDetermination_InternalBehavior{
  /* Current State */
  int state;
  ...
}

ERROR_CODE CurrentOutput_ActualCurrentDetermination_InternalBehavior_onEvent(
  CurrentOutput_ActualCurrentDetermination_InternalBehavior* self,
  int event,
  void* parameters[]);
```

(b) Generated Code: Macro Definitions corresponding to States and Events, Corresponding Struct Declaration

Figure 7.18: Example - Transformation of a State Machine

transition), or `null`, if no transition is triggered by a respective event in a certain state. The transition function being referenced in turn has to call the related *guard* and *effect functions* and has to change the state of the state machine, if necessary. Therefore it has to be parameterized with a pointer to the state machine struct, in which context it is to be executed.

Querying a state machine's current state is possible via access to the respective struct member, declared for this purpose, similar to how an attribute may be accessed by its generated *selector function*. Initialization of the state machine is performed in its generated constructor, as in case of all other classifiers.

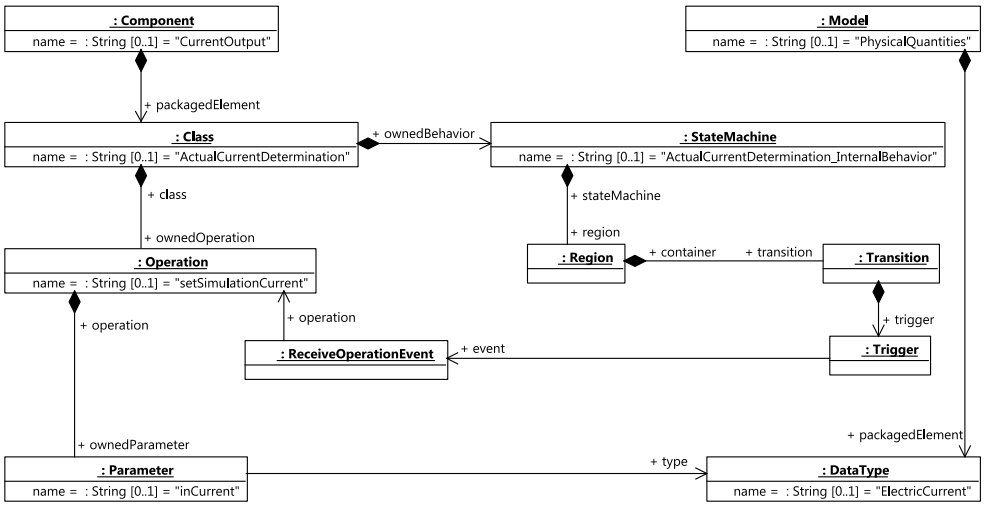
```
/************************************************************
* File: ActualCurrentDetermination_InternalBehavior.c
************************************************************/

/* Guards */
ERROR_CODE GUARD_1(void* parameters[]){
  // inEnabled == TRUE
  ...
}
...

/* Effects */
ERROR_CODE EFFECT_1(void* parameters[]){
  // simulated current = inCurrent;
  ...
}
...

/* Transitions */
ERROR_CODE TRANSITION_1(
  CurrentOutput_ActualCurrentDetermination_InternalBehavior* self, void* parameters[]){
  /* Check Guard */
  // no guard

  /* Execute Effect */
  EFFECT_1(parameters);

  /* Change State */
  //no state change

  return EVENT_CONSUMED;
}
...

ERROR_CODE TRANSITION_5(
  CurrentOutput_ActualCurrentDetermination_InternalBehavior* self, void* parameters[]){
  /* Check Guard */
  if(!GUARD_1(parameters))
    return GUARD_NOT_PASSED;

  /* Execute Effect */
  // no effect

  /* Change State */
  self->state =
  CurrentOutput_ActualCurrentDetermination_InternalBehavior_STATE_Initialized_UseSimulatedCurrent;

  return EVENT_CONSUMED;
}
...

/* Transition Table */
ERROR_CODE (* transitions)(
  CurrentOutput_ActualCurrentDetermination_InternalBehavior* self,
  void* parameters[]) [NUM_STATES][NUM_EVENTS] = {
    { TRANSITION_1, TRANSITION_2, TRANSITION_3, TRANSITION_4},
    { TRANSITION_5, TRANSITION_6, TRANSITION_7, TRANSITION_8}
}

void CurrentOutput_ActualCurrentDetermination_InternalBehavior_create(
  CurrentOutput_ActualCurrentDetermination_InternalBehavior* self){
    self->state =
    CurrentOutput_ActualCurrentDetermination_InternalBehavior_STATE_Initialized_UseProcessValueOrAlarmCurrent;
}

ERROR_CODE CurrentOutput_ActualCurrentDetermination_InternalBehavior_onEvent(
  CurrentOutput_ActualCurrentDetermination_InternalBehavior* self,
  int event, void* parameters[]){
   if(transitions[self->actualState][event] != NULL){
     return transitions[self->actualState][event](self, parameters);
   }
   else{
     return NO_TRANSITION;
   }
}
...
```

(c) Generated Code: Function Implementations corresponding to Guards, Effects, and Transitions, Declaration of State-Transition Table as Function Pointer Array and Declaration of State Transition Function

Figure 7.18: Example - Transformation of a State Machine

Additions for State Machines used as Behavioral Specification of a Design Object

If a state machine is employed as complete specification of the internal behavior of a design object, additional code has to be generated to integrate the state machine code with the one, generated for the class, which types the respective object. That is, as the state machine serves as a complete behavioral specification of the respective class, the implementation of all its operations may be generated so that each call is indeed delegated to the associated state machine. In turn, the state machine has to understand all these call events, indicating the receipt of a respective operation, which are offered by the associated class.



(a) Instance Specification: State Machine as Internal Behavior

Figure 7.19: Example - Transformation of a State Machine as Internal Behavior Specification

As indicated by Figure 7.19b for the example outlined in Figure 7.19a, a `member`, pointing to the state machine implementation has to be added to the struct of the class, whose internal behavior is being specified, and this member has to be initialized from within the constructor function for that class. The implementation of each function, generated for the operations of the class, then has to delegate all calls to the state machine, as indicated by the Figure 7.19b, passing over the corresponding event, as well as all parameter values of the call.

In case structural or behavioral features of the owning class instance have to be accessed from within the guard and effect functions of the state machine, the state machine instance in turn has to be (reverse) linked to the owning class instance. The struct generated for the state machine therefore has to contain a respective `member` as well. Its initialization may also be done from within the constructor of the owning class.

```
/************************************************************
* File: ActualCurrentDetermination.h
************************************************************/
...

/* Classifier Struct Declaration */
struct _CurrentOutput_ActualCurrentDetermination {
  ...
  /* Owned Behavior */
  CurrentOutput_ActualCurrentDetermination_InternalBehavior ownedBehavior[1];
}
...
```

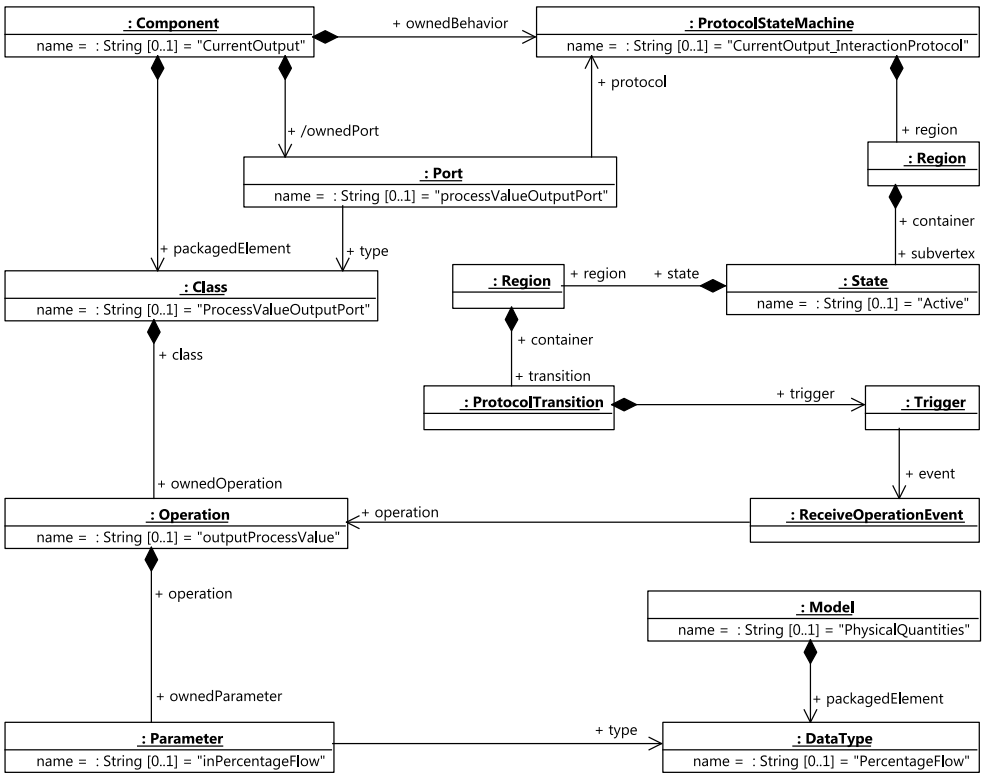
```
/************************************************************
* File: ActualCurrentDetermination.c
************************************************************/
...

void CurrentOutput_ActualCurrentDetermination_create(
  CurrentOutput_ActualCurrentDetermination* self){
  ...

  /* Initialize Owned Behavior */
  CurrentOutput_ActualCurrentDetermination_InternalBehavior_create(&self->ownedBehavior[0]);
}

ERROR_CODE CurrentOutput_ActualCurrentDetermination_setSimulationCurrent(
  CurrentOutput_ActualCurrentDetermination* self,
  PhysicalQuantities_ElectricCurrent* inCurrent){

  /* Call Owned Behavior */
  void* parameters[1];
  ... // allocate
  parameters[0] = inCurrent;

  CurrentOutput_ActualCurrentDetermination_InternalBehavior_onEvent(
    self->ownedBehavior[0],
    CurrentOutput_ActualCurrentDetermination_InternalBehavior_RECEIVE_setSimulationCurrent,
    parameters);
}

...
```

(b) Generated Code: Struct Member Declaration and Function Implementation within Owning Behaviored Classifier

Figure 7.19: Example - Transformation of a State Machine as Internal Behavior Specification

Additions for Protocol State Machines as Behavioral Specification of a Subsystem's External Interface

A protocol state machine is used to specify the valid interactions of a subsystem component from an external perspective. It may be modeled as either owned behavior of the overall subsystem component (in case several ports are affected), for a single port, or even for a single interface.



(a) Instance Specification: Protocol State Machine as Interaction Protocol

Figure 7.20: Example - Transformation of a State Machine as Interaction Protocol Specification

While the source code generated for a protocol state machine is more or less identical to that of a behavior state machine (a protocol state machine however does not specify any effects), its integration with the port's class is of course different. That is, as shown in Figure 7.20, the implementation of those functions, generated to represent the operations of the port's class (or more precise those specified by all interfaces, realized by that class) accordingly have to be generated so that protocol conformance is checked before delegating all externally arriving calls to the subsystem component's internal decomposition. This also holds in case a protocol state machine is specified as protocol of a single (provided) interface, as those are only indirectly transformed in the context of the class, which is used as type of the port, as outlined before.

```
/************************************************************
* File: ProcessValueOutputPort.h
************************************************************/
...

/* Classifier Struct Declaration */
struct _CurrentOutput_ProcessValueOutputPort {
  ...
  /* Protocol State Machine */
  CurrentOutput_InteractionProtocol protocol[1];
};

...
```

```
/************************************************************
* File: ProcessValueOutputPort.c
************************************************************/
...

ERROR_CODE CurrentOutput_ProcessValueOutputPort_outputProcessValue(
  CurrentOutput_ProcessValueOutputPort* self,
  PhysicalQuantities_PercentageFlow* inPercentageFlow){

  /* Check Protocol State Machine */
  void* parameters[1];
  ... // allocate
  parameters[0] = inPercentageFlow;

  int protocolConformance = CurrentOutput_InteractionProtocol_onEvent(
    self->protocol[0],
    CurrentOutput_InteractionProtocol_RECEIVE_outputProcessValue,
    parameters);

  /* Delegate Call */
  if(protocolConformance == EVENT_CONSUMED){
    return CurrentOuput_Coordinator_outputProcessValue(self->coordinator, inPercentageFlow);
  }
  else{
    return protocolConformance;
  }
}

...
```

(b) Generated Code: Struct Member Declaration and Function Implementations within Port's Type

Figure 7.20: Example - Transformation of a State Machine as Interaction Protocol Specification

Initialization of the respective `protocol` struct member within the port's class has to be handled either by the port's class or by the enclosing component, dependent on how the protocol state machine is used. That is, in case the protocol state machine affects several of the subsystem's ports, it is modeled as owned behavior of the component, so a reference to the single shared protocol state machine instance, associated to the subsystem, has to be passed over to the struct instances, which represent its ports. In case the protocol state machine is specified as protocol of a single port or interface, it may be initialized from within the constructor of the port's typing class instead.

Transformation of Activities

While a state machine is used for several purposes, activities, being modeled via the *Behavioral Detailed Design Diagrams*, are used within a MeDUSA *Design UML Model* only to specify internal object behavior. That is, to be precise, they are as-

sociated as *method* to the operations of an object's class. While the UML allows to specify an activity at an abstraction level comparable to that of source code, so detailed that an operation may be fully specified by it, within MeDUSA, this is not employed. That is, activities are usually only used to denote the basic control flow of a certain algorithm (inside an *application-logic* object), not specifying each individual instruction in detail.

While generating complete functional code for an activity would be rather straightforward, as most of the comprised actions are directly transferable into ANSI-C statements, because of the incompleteness of such a specification, its generation does not make sense within MeDUSA. That is, while it would be easily possible, it is yet impractical. The reason therefore is that, to guarantee robustness of the generated code against re-generation, it would have to be anticipated, where a *Subsystem Implementer* would have to manually add code (those section would have to be guarded to ensure they are not overwritten during re-generation; compare [Fun06] for details). As this, due to the incompleteness of the specification, could be an arbitrary location within the generated statements, the overall generated code would have to be *polluted* with a lot of such guards, and would thus be only hard to read and maintain.

Therefore, in the context of MeDUSA, code generation from activities is not covered and those being developed during *Detailed Behavioral Design Modeling* are thus regarded to serve for specification and documentation purposes, rather than as input to an automatic code generation tool.

Generation of Interactions

The same holds for interactions, which are developed within MeDUSA to depict collaborative behavior within the system and subsystem decompositions respectively. For the same reasons as in case of activities, that is because the specification of individual object behavior, which could be inferred from them, is not complete (as only operation calls are reflected, but no additional details), they are thus regarded to be used solely for specification and documentation purposes as well. In this role, they further serve as some sort of *proof of concept*, as they depict that the system respectively subsystem immanent behavior corresponds to the respective structural specifications. A generation of source code from the interactions contained in a *Design UML Model* is thus also not covered within MeDUSA.

Chapter 8

Tool - ViPER

ViPER (*Vi*sual Tooling *P*latform for Model-Based *E*ngineer*r*ing) [ViPERc] is a tooling platform to leverage model-based engineering. It is based on *Eclipse* [Eclipse] technology and offers support for UML-based visual modeling, UML-based ANSI-C code generation, extended support for editing and simulating of detailed narrative use case descriptions, as well as built-in dedicated methodical support for MeDUSA. ViPER thus forms the third integral part of the herein presented methodology (cf. Section 5.1).

Subsequently, the tool will be investigated in detail. This will be done first from a coarse-grained architectural perspective, introducing all major components[1] and naming their respective end-user visible features. Second, *ViPER MetiS*, the feature, which delivers dedicated methodical support for MeDUSA, will be investigated in more detail, including an in-depth discussion on some of its technical realization aspects as well.

8.1 The ViPER Integrated Development Environment

ViPER, as already outlined, is a model-based engineering platform, offering support for visual modeling, model transformation and code generation. It is realized as an *Integrated Development Environment (IDE)*, which amongst others comprises a complete *Eclipse SDK* [Eclipse]. As outlined by Figure 8.1, the *Eclipse SDK* - and thus ViPER as well - contains the *Eclipse Platform*, as well as the *Java Development Tooling (JDT)* and *Plug-in Development Environment (PDE)* features, which deliver an integrated development environment for Java-based development in general, as well as Eclipse plug-in development in particular.

[1] The Eclipse plug-in model [GB03], on which the ViPER IDE is based, refers to those top-level components as *features*, so this term will be rather used in the following.

As the ViPER IDE additionally bundles the *Eclipse C/C++ Development Tooling (CDT)* [CDT] feature, it can further be regarded as a fully-featured integrated development environment for C/C++ development, including dedicated editors, compilers, and debuggers. Furthermore, tool support for generic model-based engineering, including model validation, model-to-model transformation, as well as model-to-text generation is offered by integrating the contributions of the the *Eclipse Generative Modeling Technologies (GMT)* project [GMT].

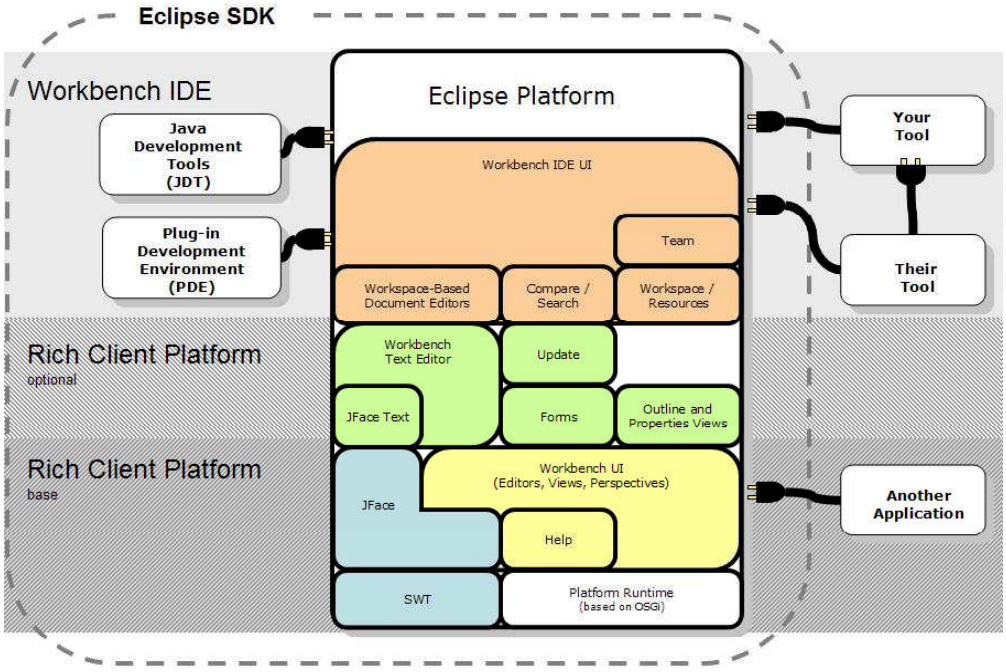


Figure 8.1: Eclipse SDK Architecture (cf. [DFK+04])

Besides these and some other third-party integrations, the ViPER IDE itself contributes graphical UML modeling support, UML-based ANSI-C code generation, editing and simulation of narrative use case descriptions, as well as methodical support for MeDUSA. This is realized by respective features[2] of the ViPER IDE, namely:

- *ViPER UML2*
- *ViPER NaUTiluS (Narrative Use Case Description Toolkit for Evaluation and Simulation)*
- *ViPER MetiS (MeDUSA Methodical Support)*

Additionally contained is the *ViPER Platform* feature, which does not deliver end-user visible features but provides common base classes and frameworks, needed for the realization of above outlined features.

[2] In the context of the Eclipse plug-in model, a *feature* bundles a set of *plug-ins*.

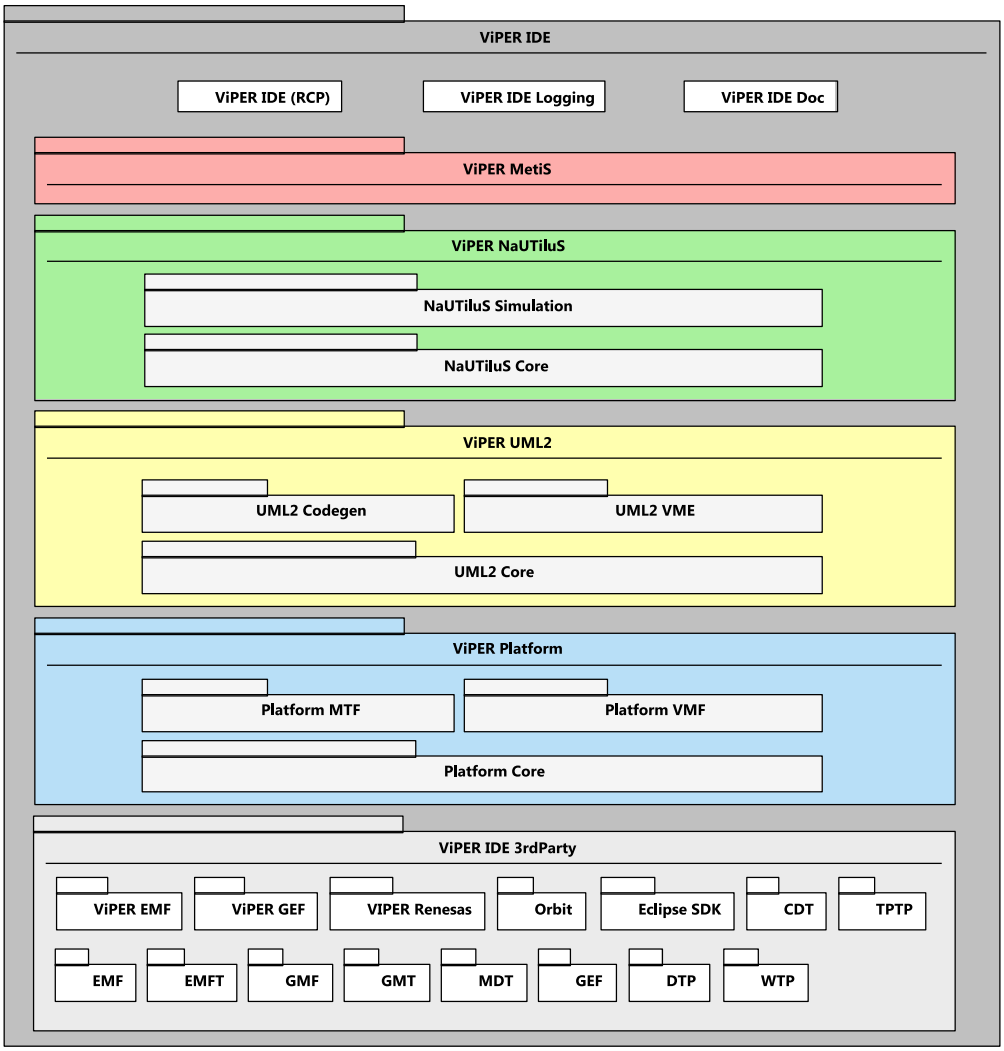


Figure 8.2: ViPER IDE Architecture

The resulting overall ViPER IDE architecture is depicted by Figure 8.2. It shows that additionally to those four features, the ViPER IDE bundles a plug-in, which is used to define the ViPER IDE Rich Client Product (RCP) [ML05], plug-ins to centralize logging and to provide IDE-wide documentation, as well as a feature to bundle all third-party contributions, including the Eclipse SDK.

ViPER Platform The *ViPER Platform*, whose plug-in architecture is depicted by Figure 8.3, forms the basis of the ViPER IDE by offering basic functionality, needed for the realization of all *higher-level* IDE features. It does not offer any directly end-user visible functionality, but it delivers basic utility classes, as well as two white-box frameworks, namely the *ViPER Platform Visual Modeling Framework (VMF)* and the *ViPER Platform Model Transformation Framework (MTF)*.

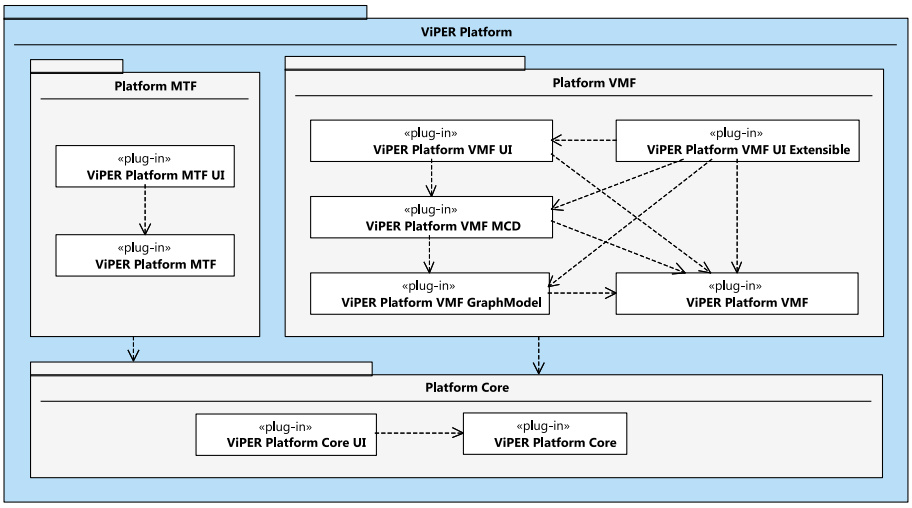


Figure 8.3: ViPER IDE Platform Plug-in Architecture

ViPER Platform VMF is a framework to support the development of graphical editors for arbitrary data models, based on technology offered by the *Eclipse Graphical Editing Framework (GEF)* [GEF] and the *Eclipse Modeling Framework (EMF)* [EMF]. It is as such the underlying framework upon which the graphical UML editing capabilities of the ViPER IDE are realized. To some extent, it is comparable to the *Eclipse Graphical Modeling Framework (GMF)* [GMF]. However, *Eclipse GMF* uses some sort of black-box approach, while *ViPER Platform VMF* was explicitly designed as a white-box framework.

ViPER Platform MTF in turn supports the development of model-to-model transformation and model-to-text generation wizards based on technology offered by the *openArchitectureWare* project [GMT]. It thus forms the basis, upon which the UML-to-ANSI-C code generation, built into the ViPER IDE, is realized.

ViPER UML2 The *ViPER UML2* feature bundles all UML-related capabilities of the ViPER IDE. Besides the *UML2 Core* feature, which similar to the *Platform Core* feature delivers common base and utility classes, it comprises, as depicted by Figure 8.4, the *ViPER UML2 Visual Modeling Environment (VME)*, which supports the graphical editing of UML diagrams and models, as well as the *ViPER UML2 Code Generators (CodeGen)*.

The *ViPER UML2 VME* feature is the reference implementation of the *ViPER Platform VMF* framework, using an EMF-based implementation of the UML meta-model, as provided by the *Eclipse UML2* project [MDT]. It thus provides an editor and related model and diagram creation wizards to graphically edit UML diagrams, conformant to the current language standard version ([OMG07c]). Currently, editing of *package*, *class*, *object*, *component*, *composite structure*, as well as *use case*, *state machine*, *activity*, *communication* and *sequence* diagrams is supported, while not in all cases, the

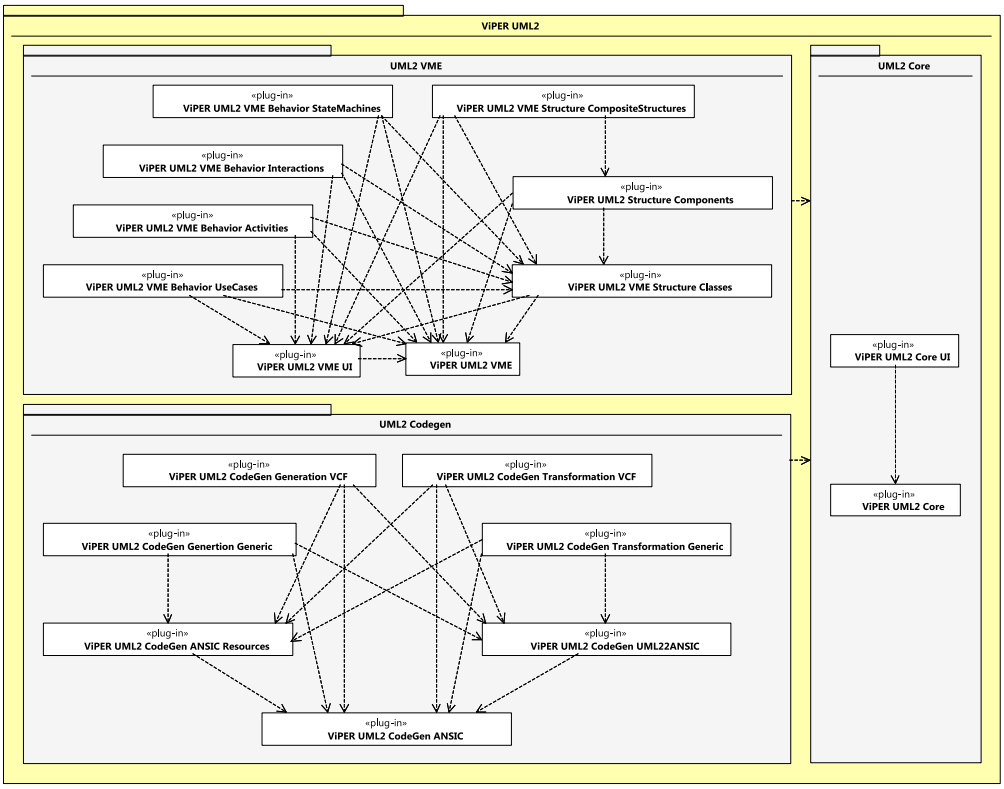


Figure 8.4: ViPER IDE UML2 Plug-in Architecture

full set of modeling capabilities, as defined by the UML, is already offered. The general look & feel, as well as the user guidance of the provided UML editor, which is depicted by the screenshot provided in Figure 8.5, is pretty much determined by the underlying GEF framework. That is, it is quite comparable to other GEF-based editors, like the Rational IBM Software Development Platform for instance.

The *ViPER UML2 CodeGen* feature offers UML-to-ANSI-C code generation. It is realized as the reference implementation of the *ViPER Platform MTF* framework and delivers generation wizards, out of which one is exemplarily depicted by Figure 8.6. Currently, a generic, platform-independent code generation, which evaluates only the structural information, captured in a UML model (cf. [Fun06]), as well as a more sophisticated, platform-specific code generation (for the Renesas M16C family), which additionally offers basic support for timing and concurrency issues (cf. [Kev07]).

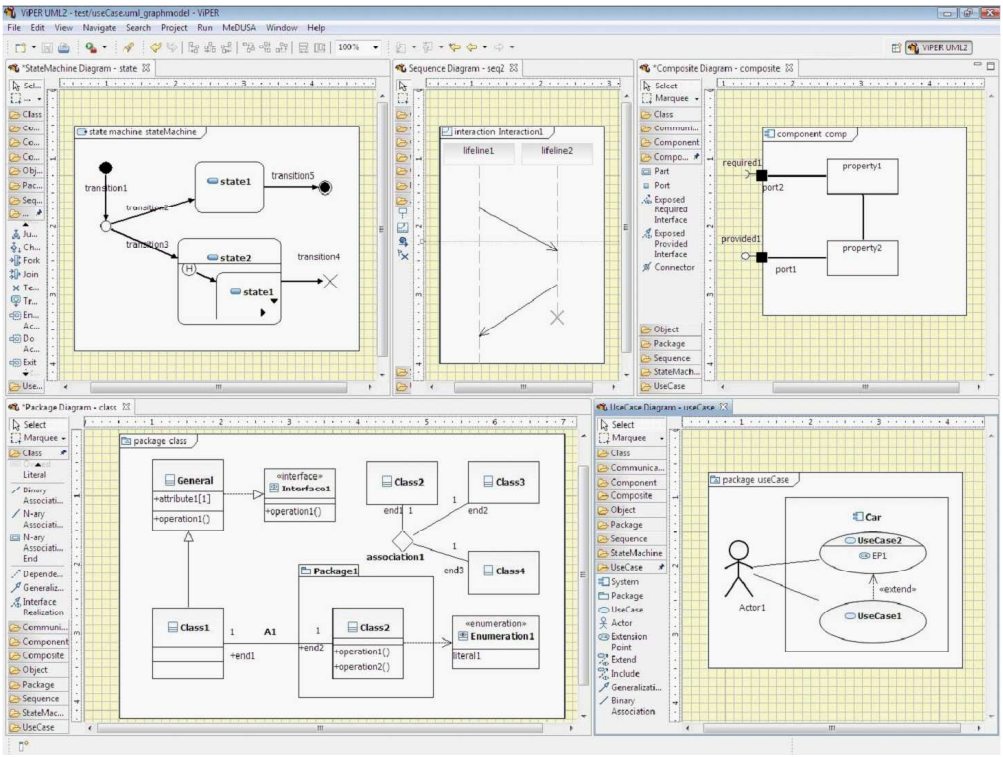
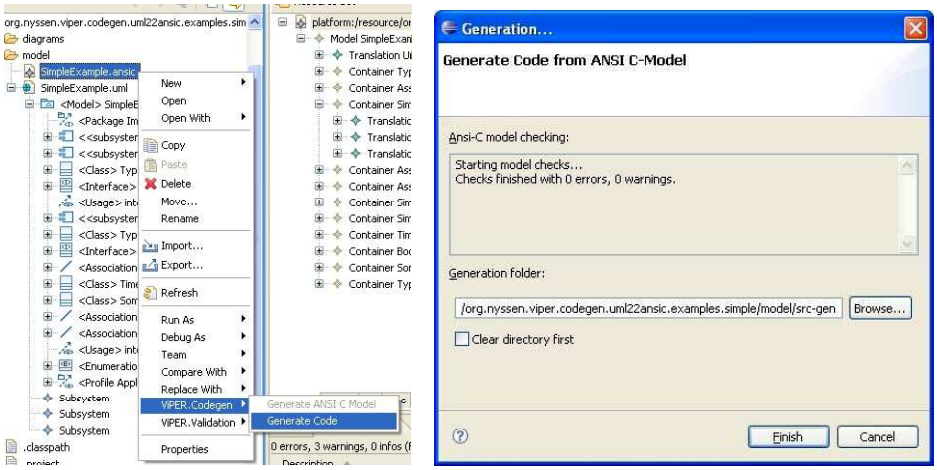


Figure 8.5: ViPER UML2 VME (Screenshot)



(a) Code Generation Context Menu (b) Code Generation Wizard

Figure 8.6: ViPER UML2 CodeGen (Screenshot)

ViPER NaUTiluS *ViPER Narrative Use Case Description Toolkit for Evaluation and Simulation (NaUTiluS)* is a toolkit to support the modeling and simulation of narrative use case descriptions. It is organized, as depicted by Figure 8.7, into two features, namely *ViPER NaUTiluS Core* and *ViPER NaUTiluS Simulation*.

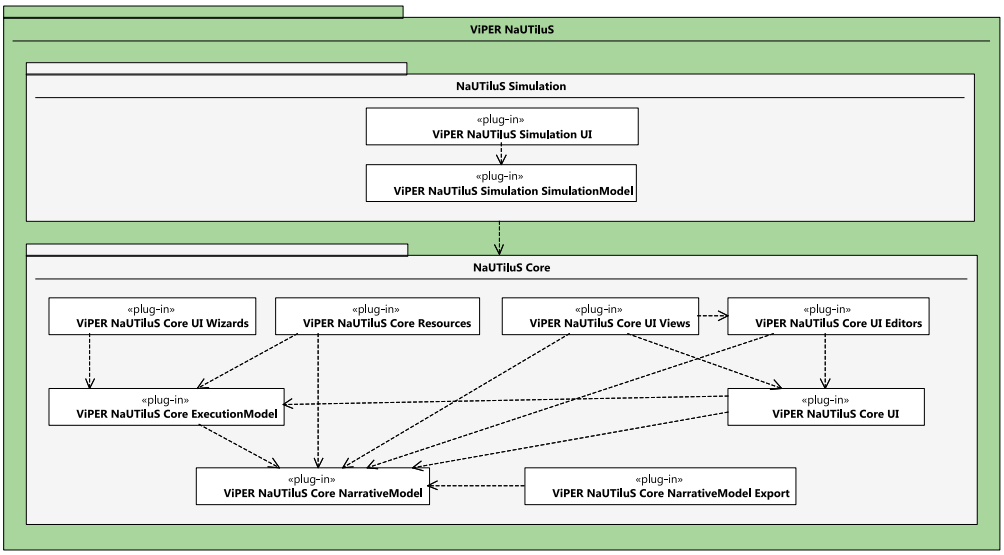


Figure 8.7: ViPER IDE NaUTiluS Plug-in Architecture

The *NaUTiluS Core* offers support for creation and manipulation of narrative use case descriptions based on a notation, initially proposed by Bittner & Spence [BS03] and refined and improved at the Research Group Software Construction, which captures use case details by describing them as *flow of events* (cf. [WNHL08]). The *NaUTiluS Core* feature offers a form-based editor together with some supporting wizards and views, depicted by Figure 8.8, to support this.

In terms of *ViPER NaUTiluS Simulation*, a prototypical simulation environment for user-system interactions, based on narrative use case descriptions is provided [Leh08]. It allows to interactively step through the different scenarios, subsumed in the detailed descriptions, by means of a simulation recorder, which is also capable playing back individual simulation traces, and which offers a set of additional supporting views to provide detailed information about individual simulation traces, as well as on the overall simulation capabilities of a set of narrative descriptions.

In the future, it is further planned to enrich ViPER NaUTiluS with capabilities (*ViPER NaUTiluS Evaluation*) to allow a detailed analysis and evaluation of use case models (including detailed narrative descriptions). This should include the evaluation of quality properties by means of dedicated metrics, as well as support for refactorings on use case models based on those evaluation results.

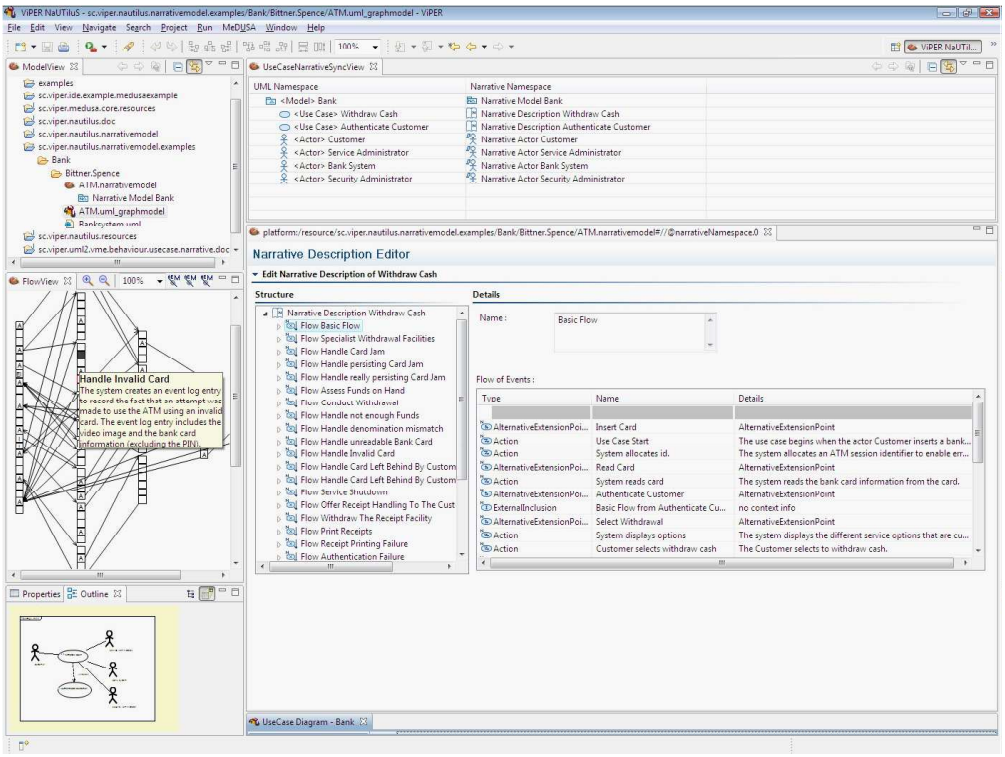


Figure 8.8: ViPER NaUTiluS (Screenshot)

8.2 ViPER MetiS- MeDUSA Methodical Support

ViPER MetiS offers dedicated methodical support for MeDUSA. This includes a browsable hypertext documentation of the MeDUSA definition, being integrated into the help system of the ViPER IDE, and wizards to support the execution of individual MeDUSA tasks. Integration between those task wizards and the browsable hypertext documentation is provided by so called *cheat sheets* (cf. Figure 8.16), which provide guidance in the step-by-step execution of the MeDUSA workflow (or parts of it) within the context of a concrete development project. A cheat sheet may be understood as some sort of instruction sheet, which outlines all MeDUSA tasks in their respective timely order. It offers context-sensitive help by referencing respective entries within the hypertext definition and additionally offers the possibility to directly invoke respective task wizards.

ViPER MetiS further offers implementations for the MeDUSA UML profiles, outlined in Section 7.2, together with a set of MeDUSA specific model validation rules, used to ensure conformance of MeDUSA UML models with respect to the model structures outlined in Section 7.1

8.2.1 ViPER MetiS Plug-in Architecture

According to the realized end-user visible features, the plug-in architecture of ViPER MetiS comprises the plug-ins *MetiS Profiles*, *MetiS ModelConstraints*, *MetiS Definition*, *MetiS TaskWizards*[3], and *MetiS Cheatsheets*.

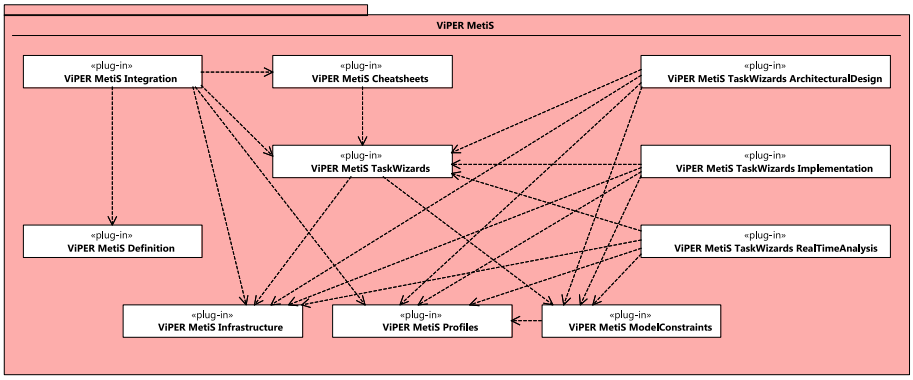


Figure 8.9: ViPER MetiS Plug-in Architecture

As outlined by Figure 8.9, the plug-ins *MetiS Infrastructure* and *MetiS Integration* have to be named to complete the picture. As denoted by its name, *MetiS Infrastructure* defines general infrastructure in terms of a MeDUSA project nature and MeDUSA specific preference settings. *MetiS Integration* serves to integrate the different MetiS plug-ins within the ViPER IDE workbench, by offering a respective workbench perspective, as well as a dedicated project creation wizard and a user interface to edit those preferences, defined by *MetiS Infrastructure*. However, as both plug-ins do indeed realize rather non innovative technical details, they are not investigated in detail in the following. The same holds for the *MetiS Profiles* plug-in, which simply bundles an Eclipse based realization of those MeDUSA profiles, introduced in Section 7.2, as well as the *MetiS Model Constraints* plug-in, which delivers a set of validation constraints and resolutions, realized by means of the EMF validation framework (cf. [EMF]).

The remaining plug-ins, namely *ViPER Definition*, *ViPER Task Wizards*, and *ViPER Cheatsheets* will be introduced adjacently, as they indeed realize dedicated methodical support. This introduction will include a detailed description of the user visible end features, each plug-in provides, as well as a short discussion on its technical realization. A more in-depth discussion on the conception and technical realization of the respective plug-ins may for example be found in [Her07] and [ViPERc].

[3] As outlined by Figure 8.9, the *MetiS Taskwizards* plug-in is actually split into four plug-ins, namely *MetiS TaskWizards*, which provides a white-box framework for the realization of dedicated task wizards, as well as one plug-in for each currently supported discipline of MeDUSA, namely *MetiS TaskWizards ArchitecturalDesign*, *MetiS TaskWizards Implementation*, and *MetiS TaskWizards RealTimeAnalysis*. Here, for the sake of simplicity, *MetiS TaskWizards* will be used to refer to this overall set of plug-ins.

8.2.2 ViPER MetiS Definition

As indicated by Figure 8.10, the *ViPER MetiS Definition* plug-in provides a browsable hypertext documenation of the complete MeDUSA definition, as captured in [NL08], quite similar to as it is done for the *IBM Rational Unified Process* within the *IBM Rational Software Development Platform* (cf. Section 5.2.2).

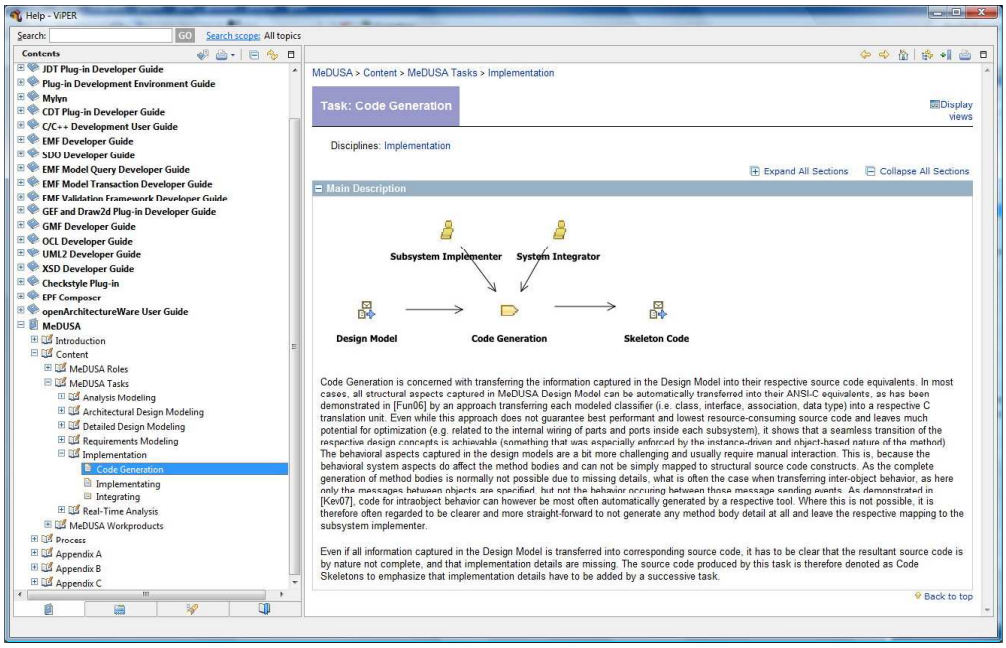


Figure 8.10: ViPER MetiS Definition (Screenshot) - Integration into Help System

MetiS Definition thus provides basic methodical support, as it enables a developer to always directly access the MeDUSA definition from within the ViPER IDE. Being realized in form of a hypertext documentation, the quick retrieval of information, as it is likely required when consulting the definition during active work with the tool, is pretty much facilitated by the enhanced navigation and search capabilities, which are offered by the Eclipse help system, into which the documentation is integrated.

Technical Realization

The hypertext documents, which contain the MeDUSA definition, are of course not handcraftet, but can be generated automatically from a MeDUSA UMA method library (cf. Section 6.2.1) by using the respective HTML publishing functionality of the *Eclipse Process Framework (EPF) Composer* tool [EPF]. The integration of the published HTML pages into the Eclipse respectively ViPER IDE help system is however not directly supported by the EPF Composer.

This can be achieved by offering a *table of contents* XML-file, which is exemplarily depicted by Figure 8.11. It has to be contributed to the respective `toc` extension point of the *org.eclipse.help* plug-in and registers the top-level HTML pages of the method definition within the contents browser of the help system, so that direct accessibility of the documentation is achieved.

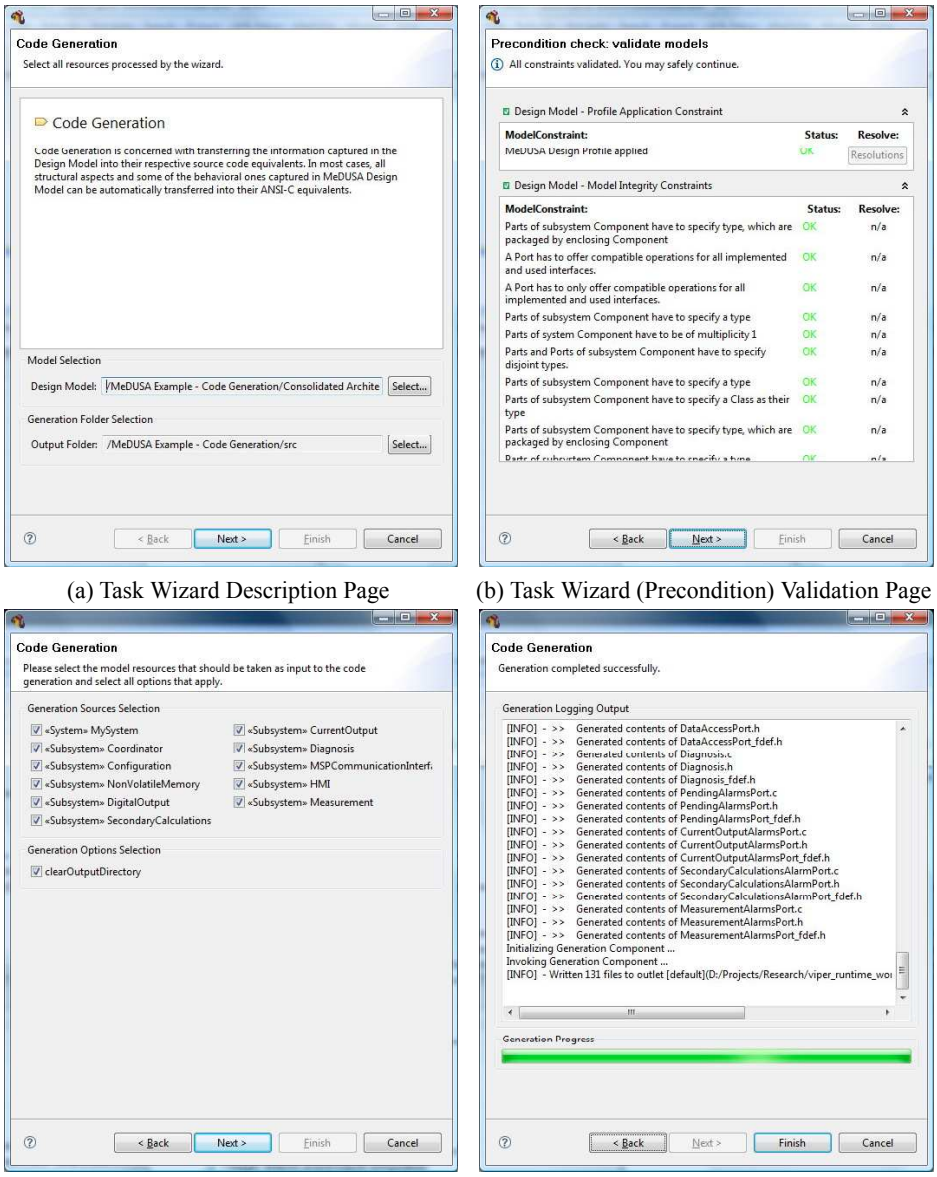
```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<?NLS TYPE="org.eclipse.help.toc"?>

<toc label="MeDUSA">
  <topic
    href="/sc.viper.metis.definition/html/medusa_concepts/customcategories/introduction_CA894658.html"
    label="Introduction">

    <topic
      href="/sc.viper.metis.definition/html/medusa_concepts/guidances/supportingmaterials/
            introduction_2BA636D3.html"
      label="Introduction" />

    <topic
      href="/sc.viper.metis.definition/html/medusa_concepts/guidances/examples/
            example_system_4D4F0766.html"
      label="MeDUSA Example System" />
  </topic>
  ...
</toc>
```

Figure 8.11: ViPER MetiS Definition - Help *toc* File (generated)

As MeDUSA is a living method, and as therefore its definition is likely to change over time, maintaining such a *toc* file manually would be impractical, and errorprone as well. Therefore, a ViPER specific extension to the *EPF Composer* tool was developed in terms of the *ViPER Method Support Framework (MSF)*. It extends the publishing functionality of the EPF tool by generating a respective *toc* file directly together with the HTML pages of the method definition, and - as it is not specifically dependent on MeDUSA - may be generically applied to generate respective files for arbitrary UMA-based method definitions. As it is not an essential part of the *ViPER MetiS* feature, but is only used within its development and maintenance, it will not be regarded further within this work. The interested reader may refer to [Her07] as well as [ViPERc] to find an in-depth documentation of *ViPER MSF*.

8.2.3 ViPER MetiS Task Wizards

The *ViPER MetiS TaskWizards* plug-ins deliver dedicated wizards to support the execution of selected MeDUSA tasks. Being a model-based construction method, all tasks defined by MeDUSA can generally be classified into tasks related to model manipulation, model-to-model transformation, as well as model-to-text generation. Modeling, i.e. model manipulation, is a rather creative activity, which is probably best supported by means of flexible graphical editors and additional supporting tools, as they are provided by the *ViPER UML2 VME feature*. Transformations as well as generations however offer an inherent automization potential and can - due to the linearity and goal-driveness of their execution - be probably best supported by means of dedicated wizards, which prescribe a general control flow and user-guidance in terms of their dialog pages.



(a) Task Wizard Description Page

(b) Task Wizard (Precondition) Validation Page

(c) Task Wizard Generation Sources And Options Selection Page

(d) Task Wizard Generation Processing Page

Figure 8.12: ViPER MetiS Code Generation Wizard (Screenshots)

In line with these insights, the *ViPER MetiS Task Wizards* aims at providing dedicated wizards for model transformation respectively model generation tasks[4]. Currently, wizards to support the *Subsystem Identification* as well as the *Code Generation* task (outlined by Figure 8.12) are provided, thus demonstrating exemplarily how dedicated support for each kind of wizard may be realized. Wizards to support the different

[4]It may be mentioned that from a technical viewpoint modeling may as well be regarded as a model transformation, having the same input and output models. To this extend, modeling tasks, which offer a certain automation potential, could as well be supported by a respective model transformation wizard.

kinds of *Real-Time Analysis* are currently being developed (cf. [Rit08]). As indicated by Figure 8.12 in case of the *Code Generation* task wizard, each wizard supports the execution of a respective MeDUSA task by guiding the user through its wizard pages. While some of the pages are of course specific to the respective task that is supported, some pages are indeed common to all wizards.

That is, all wizards offer a description page, as depicted by Figure 8.12a, as their starting page. It offers a short general description of the supported MeDUSA task, as well as a possibility to specify input and output models (in case of a transformation task), or output folders (in case of a generation task). This starting page is always followed by a *Validation Page*, as it is depicted by Figure 8.12b, which is used to check preconditions on the selected models, which have to be guaranteed to allow the execution of the respective wizard. The validation page does not only present the validation results to the end-user, but - where applicable - it also offers automatic resolutions in case conditions are violated.

Having evaluated all preconditions successfully, the user may then proceed to the task-specific wizard pages. Within the *Code Generation* wizard, a page to choose the generation input and to select certain generation options (cf. Figure 8.12c), as well as a page to output processing results (cf. Figure 8.12d) may be named. In case a wizard specifies postconditions, the wizard is always concluded by a *Validation Page* to check those postconditions. It has to be mentioned that this is mandatory in the case of transformation wizards, where - due to the possibility of their iterative execution on a set of models - at least all preconditions have to be fulfilled as postconditions as well.

Technical Realization

Due to technical implementation issues, the *ViPER MetiS Task Wizards* is not realized as a single plug-in, but actually split into the following:

- ViPER MetiS Task Wizards
- ViPER MetiS Task Wizards - Architectural Design
- ViPER MetiS Task Wizards - Implementation
- ViPER MetiS Task Wizards - Real-Time Analysis

While the three last mentioned plug-ins deliver those concrete task wizards, supporting the already named tasks of the respective MeDUSA disciplines, the *ViPER MetiS Task Wizards* plug-in itself delivers a white-box framework to support the implementation of arbitrary transformation respectively generation task wizards.

Amongst others, it provides generic support for the loading and saving of models, which serve as input and output of a respective task, for the evaluation of specified pre- and post-conditions on those models, for executing and reverting changes on them,

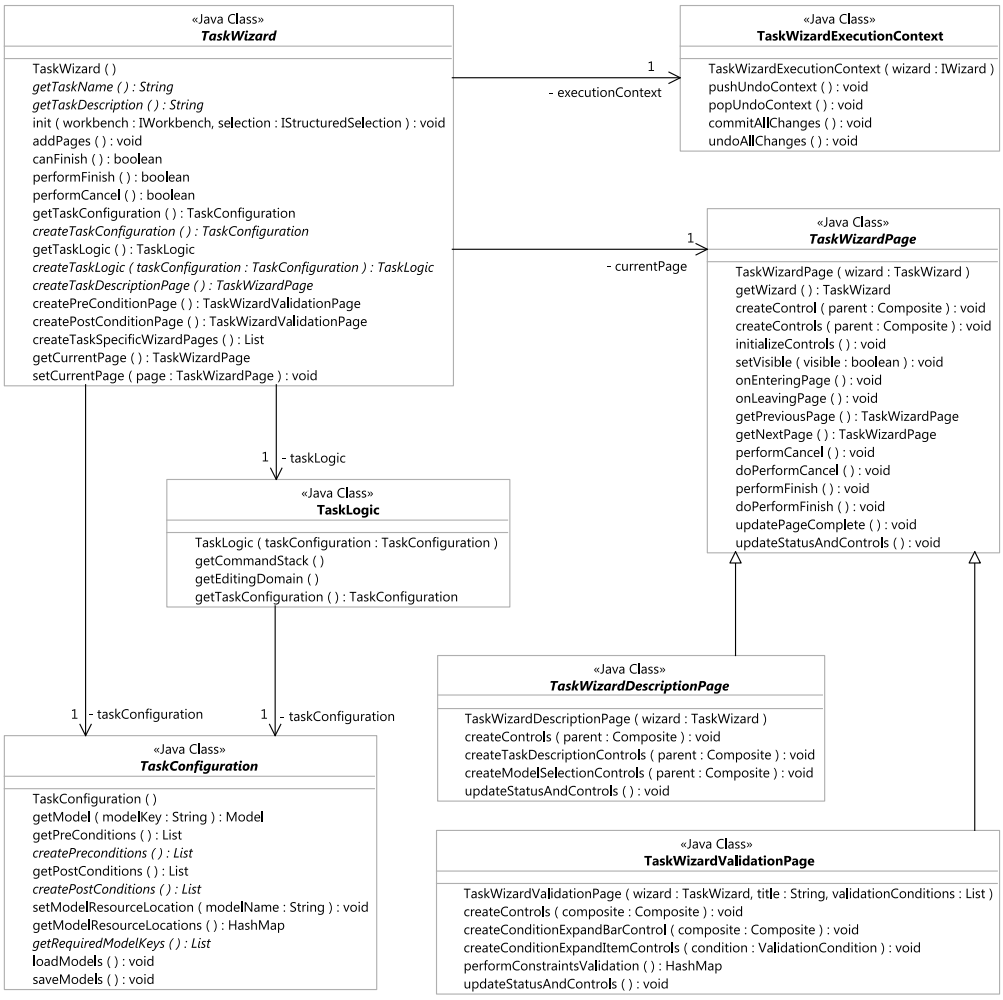


Figure 8.13: Detailed Class Design of the ViPER MetiS Task Wizard `core` Package

as well as for tracking all applied changes, which is needed to support the correct iterative execution of a respective wizard. It is split into three basic packages, namely `core`, `transformation`, and `generation`. The `core` package, which is depicted in Figure 8.13, defines the abstract `TaskWizard` and `TaskWizardPage` classes, which serve as the base classes for all task wizards and their respective wizard pages. The core package also provides concrete implementations of those wizard pages, which are commonly used in all wizards, namely the `TaskWizardDescriptionPage` and the `TaskWizardValidationPage`.

The abstract `TaskWizard` base class ensures that a common control flow in terms of a defined order of wizard pages is inherent to all MetiS task wizards. This is realized by means of the `addPages` template method, which defines a hook method, namely `createTaskSpecificWizardPages` to control that all task specific pages are added between the validation pages used to check the pre- and postconditions. The `TaskWizard` class further implements control of the *back*, *next*, *finish*, and *cancel* buttons, of-

fered on each wizard page. While the enabled state of those buttons may be influenced by a respective wizard page via its *page status*, the application and reverting of all model changes, being performed through an individual wizard page, does not have to be handled by the page itself, but is transparently done within the `performCancel` and `performFinish` template methods within the `TaskWizard` class.

This is achieved, by enforcing that all model manipulations of a `TaskWizardPage` have to be executed as `Commands` on a `CommandStack`, which is shared by all `TaskWizardPages`. The `CommandStack` in turn delegates the execution of all commands to an underlying `OperationHistory`[5], which is controlled by a `TaskWizardExecutionContext`, associated to each `TaskWizard`. It maintains an own *undo context* for each wizard page so that all changes, which are applied from within a respective page, can be easily and transparently reverted, when navigating to a previous page or when performing a cancel operation, without any awareness of the respective `TaskWizardPages`.

Access to the `CommandStack`, shared by all `TaskWizardPages` is provided by a so called `TaskLogic`, associated to each `TaskWizard`, which is intended to be subclassed to provide the respective (application) logic of the supported task. That is, while the control logic is thus directly built into the `TaskWizard`, and while all user interface related aspects are covered by the `TaskWizardPages`, the task specific logic is meant to be separated. The `TaskLogic` therefore has direct access to an underlying `TaskConfiguration`, which keeps task specific configuration information, as the locations of all used input and output models, as well as the pre- and postconditions that have to be checked on those models. It also handles the loading and saving of those models.

Generation and transformation related specializations of the `TaskWizard` and `TaskWizardPage` base classes, as well as their related `TaskConfigurations`, are realized by respective subclasses within the `generation` and `transformation` packages, which are depicted by Figure 8.14.

As in terms of a `GenerationTaskWizard` a respective output folder has to be specified, a specialization of the `TaskWizardDescriptionPage`, denoted as `GenerationTaskDescriptionPage` is provided, which offers the possibility to specify a respective file system path for this purpose. Additionally, a specialization of `TaskConfiguration`, namely `GenerationTaskConfiguration` is provided, which realizes the backup of the folder contents prior to the wizard execution, as well as the restoring of this contents in case of a cancellation or failure of the generation.

A `TransformationTaskWizard` in turn needs to ensure traceability in terms of which model elements of a respective input model were transformed into which model elements of a related output model. This is necessary, as the iterative execution of a wizard on the same set of models (which may have changed in the meantime) has to reflect already committed user decisions of previous wizard executions. As an example consider the *Subsystem Identification* task, which is concerned with transferring analy-

[5]The `Operation History` is a concept built into the Eclipse platform to realize a unified undo and redo handling.

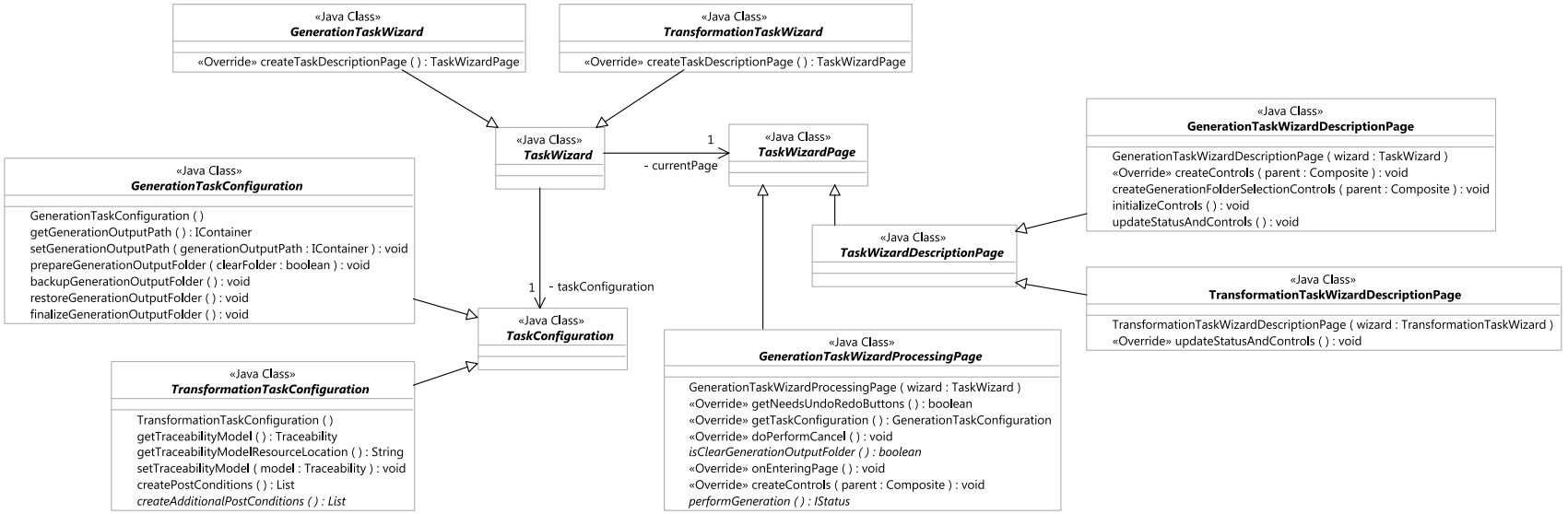


Figure 8.14: Detailed Class Design of ViPER MetiS Task Wizard - Generation & Transformation Wizard

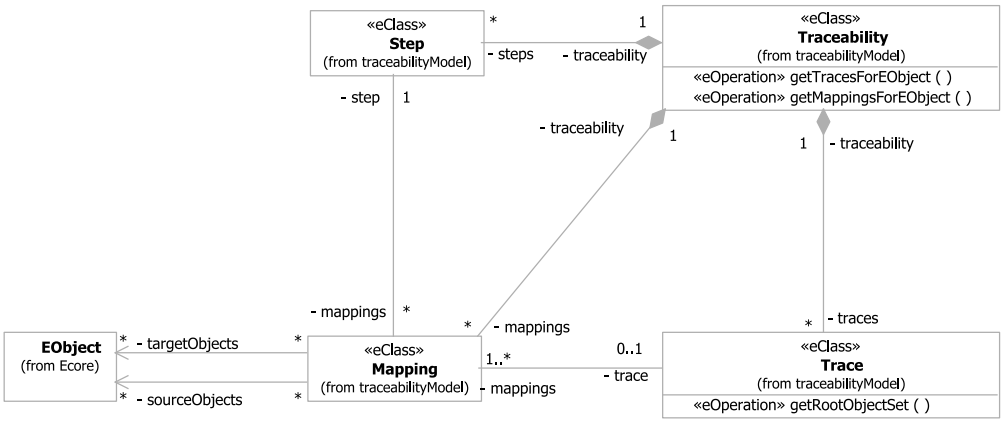


Figure 8.15: ViPER MetiS TaskWizards Traceability Meta-Model

sis objects into design objects, and with grouping those design objects into subsystems. While there is no direct model relationship between the *Analysis UML Model*, serving as input model, and the *Design UML Model*, serving as output model, the task wizard has to recognize, which analysis objects were already mapped to which design objects during an earlier execution of the wizard, so that the user does not have to redo any mappings already specified.

The abstract `TransformationTaskWizard` class accounts for this by offering support for tracing all model changes, applied during the execution of the wizard. it therefore maintains a respective *traceability model*, in which it can store all relevant traceability information. Such a traceability model is automatically created by the related `TransformationTaskConfiguration` of the wizard within a private folder of the respective project, in which the selected input and output models of the wizard reside[6], and is loaded and stored automatically, together with the specified input and output models.

The meta-model, defining the structure of all *traceability models*, is depicted by Figure 8.15. Its central concept is that of a *mapping*, which is understood to be an arbitrary relation between a set of source and target objects. *Mappings* may be grouped under two aspects. A *step* is used to group logically related *mappings*, which refer to possible unrelated sets of source and target objects but logically belong to the same model manipulation and thus occur at the same point in time. A *trace* in turn is used to group chronologically adjacent *mappings*. A *trace* thus allows to *trace* the course of a set of source objects through a temporal sequence of adjacent model manipulations. Due to limited space, the *traceability model* will not be further elaborated here. The reader may refer to [Her07] for a more in-depth discussion on the model, including a demonstrative example of its application within the *Subsystem Identification* task wizard.

[6]The `TransformationTaskWizardDescriptionPage` subclasses its `TaskWizardDescriptionPage` base class in order to confirm the restriction that all specified input and output models actually reside within the same project. This is needed, because traceability information between those models could otherwise not be consistently managed.

8.2.4 ViPER MetiS Cheatsheets Plug-In

While the *MetiS Definition* plug-in offers a broad but rather static view of the MeDUSA method definition, and while the *MetiS Task Wizards* plug-ins deliver support, which is rather dynamic, but limited to the execution of a single respective task, the *MetiS Cheatsheets* plug-in integrates both aspects by offering support for a guided step-by-step execution of the MeDUSA Workflow. That is, it provides so called *cheat sheets*, which are sort of non-modal interactive help dialogs that - according to [CR06] - *"are designed to walk you through a series of steps to complete a task and automatically launch any required tools."*.

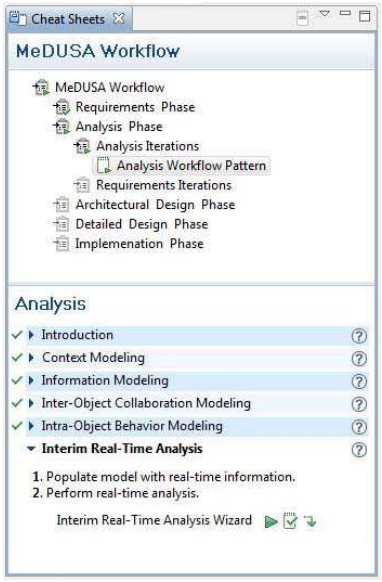


Figure 8.16: ViPER MetiS Cheatsheets (Screenshot)

As depicted by Figure 8.16, the *MetiS Cheatsheets* plug-in offers a composite cheat sheet that contains task groups and tasks according to what is defined by the MeDUSA method operations. It thus allows a user to step through the different tasks of the MeDUSA method, as defined by the *MeDUSA Workflow*, performing possible iterations as well. Integration with those help pages, offered by *MetiS Definition* plug-in is achieved by offering browsable links (the question marks, which are visible besides each task within Figure 8.16), via which the description of the respective task is directly accessible. Integration with those dedicated task wizards, offered by the *MetiS Task Wizards* plug-ins, is achieved by offering the possibility to start the execution of a respective wizard from within the cheat sheet, as exemplarily depicted by Figure 8.16, where the *Interim Real-Time Analysis* task wizard may be directly executed from within the respective cheat sheet entry.

Technical Realization

Technically, cheat sheets are specified by means of an XML-file, which has to be registered to the respective `cheatSheetContent` extension point of the *org.eclipse.ui.cheatsheets* plug-in. For illustration purposes, parts of the composite cheat sheet, which realizes the MeDUSA Workflow, is depicted by Figure 8.17.

```
<?xml version="1.0" encoding="UTF-8"?>
<compositeCheatsheet name="MeDUSA Workflow">
  <taskGroup kind="sequence" name="MeDUSA Workflow" skip="false">
    <taskGroup kind="sequence" name="Requirements Phase" skip="false">
      <taskGroup kind="sequence" name="Requirements Iterations" skip="false">
        <task kind="cheatsheet" name="Requirements Workflow Pattern" skip="false">
          <param name="path" value="requirements.xml">
          </param>
          <param name="showIntro" value="true">
          </param>
        </task>
      </taskGroup>
    </taskGroup>
    ...
  </taskGroup>
  ...
</taskGroup>
...
</compositeCheatsheet>
```

Figure 8.17: ViPER MetiS CheatSheets - MeDUSA Workflow Composite Cheatsheet

Of course, as in case of the MeDUSA definition, crafting those cheat sheets by hand would be tedious work. Therefore, the already introduced *ViPER MSF* framework was additionally enhanced with a generation capability, which offers support for the generation of respective cheat sheets for a given UMA method library (or, to be more precise, for each workflow or delivery process defined within it).

The *ViPER MSF* framework supports the integration between those cheat sheets and the electronic MeDUSA definition, by embedding links to the generated help pages directly into the generated cheat sheets. Integrating those wizards, offered by the *MetiS Task Wizards* plug-ins, is realized by specifying a unique wizard identifier as a *tool mentor* within the UMA definition of the method, which is then evaluated by the *ViPER VMF* framework's cheat sheet export functionality in a sense, that a corresponding *command* to start execution of the wizard, is generated directly into the corresponding cheat sheet file as well. Further information on the detailed technical realization within the *ViPER MSF* framework can be found in [Her07] as well as [ViPERc].

8.2.5 Adoptions & Innovations

As with MeDUSA, ViPER in general - as well as *ViPER MetiS* in particular - incorporates various already approved concepts, which have found their permanent place in todays modeling tools. Especially the UML related modeling and code generation capabilities, which are offered by *ViPER UML2* are based on commonly applied and industry approved technology and does only bear limited potential for actual novelties.

Regarding the methodical support that is built-in the situation is similar, as a single prominent outstanding novelty may most likely not be found. It is rather the combination of approved concepts together with some slight partial improvements that make up most of its value. Task wizards, as they are offered by *ViPER MetiS*, can be for instance be pretty much compared to those *Activity Agents* offered by the Jaczone Waypointer tool. However, the possibility to reflect already performed user decisions through improved traceability, which is essential for their usability within an iteratively executed method like MeDUSA, or the possibility to explicitly depict pre- and postconditions and the ability to automatically resolve most of the identified violations, may be regarded as innovations.

To draw a conclusion, one may thus state that the most significant contribution of the ViPER tool regarding its methodical support for MeDUSA may indeed not be seen in very innovative partial solutions. Indeed the real value lies within the seamless integration of methodical support within an overall integrated development environment, as this can - as already indicated in Section 5.2.2 - not be found in todays modeling tools. Even if the concepts of offering a navigable documentation of a method definition, or the offering of dedicated wizards to support certain tasks within the execution of a method can be found within existing tools (cf. for instance the *Method Browser* within the Rational Software Development Platform as well as the *Activity Agents* within the Jaczone Waypointer), their seamless integration, as it is demonstrated by *ViPER MetiS*, may be regarded as an innovation.

Part III

Evaluation & Conclusion

Chapter 9

Evaluation of MeDUSA

9.1 Continuous Evaluation - A Living Method

From its initial conception in early 2005, MeDUSA has gone a long way. Having started as a slight enhancement to the COMET method [Gom00], incorporating several experiences gained from the application of COMET in pilot projects, conducted at ABB Automation Products GmbH, MeDUSA has grown - at latest with its first publication in 2007 [NL07a] - into an independent and self contained method. Practical experiences as well as evaluation results have been continuously incorporated into the method since, what does in particular hold for the second, completely revised revision of MeDUSA, whose reference manual has been published in 2008 [NL08]. It is - with some slight changes, which have already been incorporated since its publication - also the one documented herein (cf. Chapter 6).

Before naming some general evaluation experiences, which are applicable to all of MeDUSA's revisions to some extend, and before quoting some recent evaluation results, which have been gathered with the application of MeDUSA's Second Edition in a pilot project, it is thus quite revealing to investigate MeDUSA's course from its initial conception up to the publication of its Second Edition, as most practical experiences and evaluation results, gathered over the time, have been directly incorporated into the method.

9.1.1 Initial (Pre-Published) Edition - 2005/2006

The application of the object-oriented COMET method, MeDUSA's direct predecessor, in some field studies within ABB Automation Products GmbH [NMSL04] demonstrated a general applicability of COMET, but also unveiled some severe shortcomings within the regarded application domain, which finally lead to the insight that a customized method would indeed be needed, capable to face the very special technical and organizational constraints within the industrial automation application area.

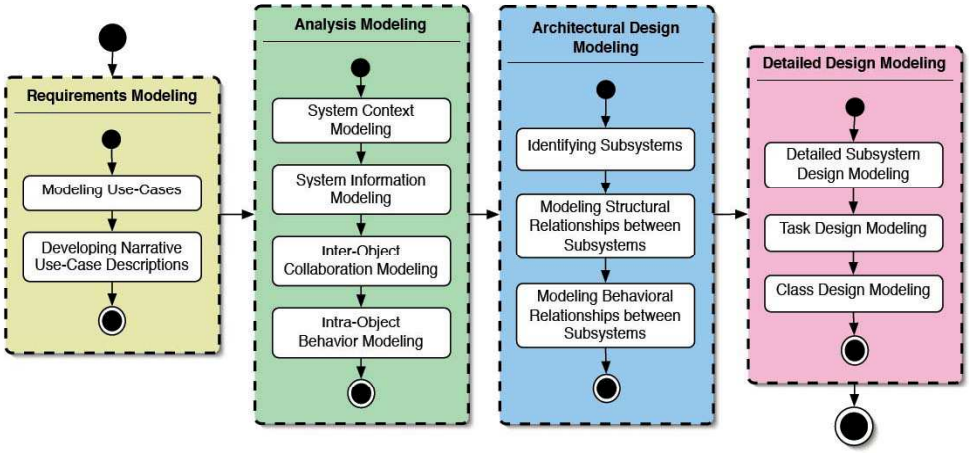


Figure 9.1: MeDUSA Pre-Published Edition - Workflow

Being itself the result of several iterations (cf. historic outline in Section 3.1), COMET was nevertheless considered to be a very concise and systematic method, which could serve as a good starting point for customizations and adoptions. Especially its use case driven approach seemed to be very promising and was directly incorporated as one of the key characteristics of MeDUSA. COMET's object-oriented nature however was considered to be a major hindrance for its application within the industrial automation domain, as a seamless transition from an object-oriented design into a procedural implementation is only hard to achieve.

Class-Based & Instance-Driven vs. Object-Oriented The initial, yet unpublished version of MEDUSA[1], whose general workflow is depicted in Figure 9.1, reflects this by what has been referred to in Section 6.3 as its instance-driven nature. That is, while COMET had employed static modeling on the classifier level to depict the context and data of the software system, MeDUSA made use of instance-based modeling by means of object diagrams for this purpose. Performing *Analysis Modeling* completely in terms of modeling objects, the instance-driven nature of MeDUSA thus ensured that classifiers were not employed earlier than during *Architectural Design Modeling*, so that object-oriented concepts were not enforced and a seamless transition into a procedural implementation could now be guaranteed.

Based on Second Generation of UML Closely related to this is the fact that, being based on UML 1.5 and thus employing class diagrams to specify the (structural) architectural design, COMET does not achieve a clear and explicit definition of the context dependencies of a respective subsystem, so that a distributed development of subsystems, as well as their reuse are hindered. Being based on the then upcoming UML 2.0

[1] Also not officially published, documentation about the initial (pre-published) version of MeDUSA, can for example be found in [Fun06].

standard [OMG05c], MeDUSA was designed to use the newly introduced composite structure diagrams for this purpose, which had been earlier identified as an adequate means to address those problems (cf. [NLS+05]). By explicitly specifying all required as well as provided interfaces, a subsystem can be specified with all its context dependencies, so that it can be independently developed - and reused. Furthermore, as composite structure diagrams allow to specify an internal subsystem decomposition in terms of aggregated objects (rather than classes), class modeling could now be postponed to the *Detailed Design Modeling* phase, thus contributing as well to the instance-driven nature of the method.

9.1.2 First (Published) Edition - 2007

With its initial publication [NL07a], as *Method for UML2-based Design of Embedded Software Applications*, the MeDUSA definition was formalized by means of the *SPEM* standard (cf. Section 6.2.1), splitting it into a specification of method content and method operations, which seems to be beneficial not only from documentation perspective but also facilitates the understandability and learnability of the method.

Real-Time Aware Taxonomies Regarding contents, the first of two major changes is the introduction of a revised *MeDUSA Actor Taxonomy*, which explicitly separates *trigger* actors from *interface* actors, as well as a related *MeDUSA Object Taxonomy*, which incorporates this division as well. While COMET had already known the concept of *timer* actors (and objects), *eventer* actors were not explicitly identified and represented, so that a clear separation of those event sources, triggering concurrent system behavior, and those external software and hardware interfaces, serving as mere (passive) communication interfaces, could not be achieved. With the clear division into active and passive actors respectively objects, incorporated into the *First Edition*, timing and concurrency constraints could then be specified already during *Requirements Modeling* and *Analysis Modeling*[2].

Early Task Design Closely related to this is the second major change, namely the shifting of former *Task Design Modeling* from *Detailed Design Modeling* into respective tasks in the *Analysis Modeling* and *Architectural Design Modeling* phases, as indicated by Figure 9.2. While this does not seem to be a major novelty from the first sight, it is very formative for MeDUSA, as it exposes the importance of real-time and concurrency requirements and stresses the necessity of addressing those aspects early (cf. [NL07b]). Even if no real-time analysis was performed during *Requirements Modeling* yet, as it is now incorporated into MeDUSA, addressing those constraints already during *Analysis Modeling* could be regarded as a major improvement with respect to MeDUSA's initial unpublished version, were *Task Design Modeling* was done as part of *Detailed Design Modeling*.

[2] Note that modeling of synchronization and interference issues within the *Requirements UML Model* was regarded to be an open issue at that time (cf. [NL07b]).

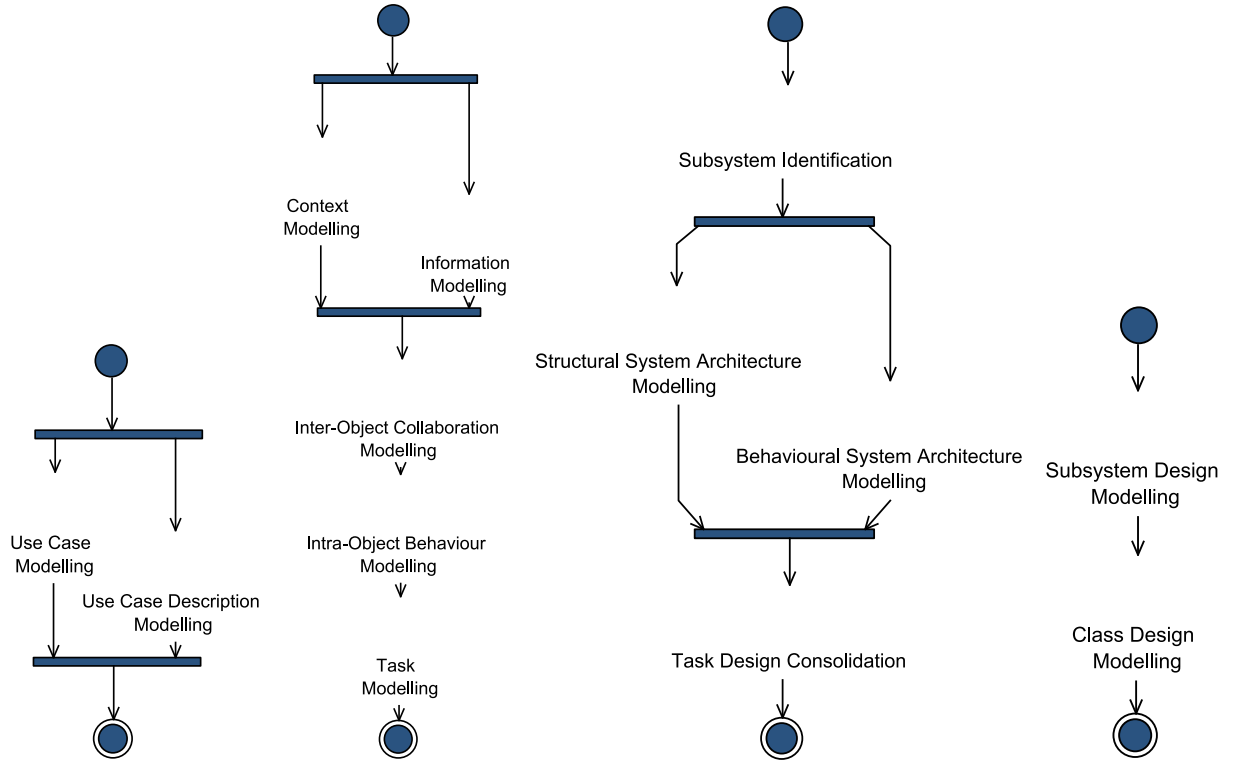


Figure 9.2: MeDUSA First Edition - Workflow Patterns

9.1.3 Second (Published) Edition - 2008

The *Second Edition* of MeDUSA actually is a complete revision rather than a simple bug-fixing update of the *First Edition*. Besides several minor changes, which were incorporated to improve the understandability and practical applicability of the method (which will not be discussed here in detail), the most immanent change is that of the MeDUSA acronym into *Method for UML2-based Construction of Embedded & Real-Time Software*. While the reformulation of the development target from *software applications* into *software* points to a mere clarification of terms[3], the other terminological differences actually symbolize some major changes that were incorporated.

Software Construction vs. Software Design Changing the denomination from *design method* into *construction method* indicates that - according to the goal of *Methodological Completeness*, as it is formulated in Section 4.3 - all tasks related to *Implementation* are now also explicitly covered by MeDUSA. That is, it was not only decided to include those tasks into the MeDUSA definition, but in particular the mapping of design concepts into ANSI-C source code equivalents is now being explicitly addressed by the *MeDUSA Code Generation Schema* (cf. Section 7.3).

Continuous Real-Time Analysis The explicit naming of *real-time* software as target domain indicates what has become one of the major features and sets MeDUSA apart from other software engineering methods in the domain, namely the explicit and continuous modeling and analysis of real-time constraints[4]. That is, in contrast to earlier versions, a real-time analysis is now continuously performed throughout the method execution, from the early *Requirements Modeling* up to the late *Architectural Design*, what can actually be regarded as a major novelty.

Narrative Modeling of Use Case Details Based on further research experiences, gained since MeDUSA's initial publication related to use case modeling, a new format for narrative, textual use case descriptions, which is based on a notation proposed in [WNHL08] and [HLNW09], has been adopted to *Use Case Details Modeling*.

Architectural vs. Detailed Design Further, the separation of concerns between *Architectural Design Modeling* and *Detailed Design Modeling* was redefined by merging the consolidation of the passive design objects, which was up to then subsumed by the *Subsystem Design Modeling* task within *Detailed Design Modeling*, with the consolidation of the active design objects, which was formerly performed in terms of *Task*

[3] In the regarded application domain, true application or true system software is rather unlikely and an actual software system is most likely be a mixture of both. The denomination was thus changed to indicate that both types of software (or even mixtures) are actually within MeDUSA's scope.

[4] Note that some research experiences related to modeling and analysis of timing and concurrency constraints, which are documented herein, have been incorporated after the publication of the Second Edition (cf. [Rit08])

Design Consolidation within *Architectural Design Modeling*. There are two major reasons for this rather impacting change.

At first, the result of the consolidation of active design objects was - according to the definition of the *Task Design Consolidation* - actually not documented within a distinct work product. Instead the *Task Design Consolidation* task prescribed a task and schedulability report to document the results of the subsequently performed real-time analysis as its single work products; the consolidating of active design objects was thus somehow subsumed by a re-iteration of the *Subsystem Identification*, which was initiated by the outcome of the real-time analysis, performed within *Task Design Consolidation*, so indeed, a clarification was pretty much required.

Second, the separation between consolidating the active (i.e. trigger) and passive (i.e. non-trigger) design objects seems to be artificial, after having gained experience with the practical application of the method. It was adopted directly from COMET and was actually retained within MeDUSA, motivated by the idea to develop as much as possible in a distributed, parallel manner. That is, having the consolidation of the passive design objects within the responsibility of the respective subsystem designers, this work could actually be performed in parallel, independent for each subsystem. A major drawback inherent to this separation however was that those tasks, which were performed between the *Subsystem Identification* and the consolidation of active and passive design objects within *Task Design Consolidation* and *Subsystem Design Modeling*, often had to be re-iterated after the subsystems internal decomposition had been consolidated, because this consolidation often led to changes within the subsystem division itself; it might for example be the case that a design object is decided to be split apart, moving one of the newly obtained fragments into a new subsystem, so that the subsystem interface may indeed be affected as well.

Reduced Modeling Overhead within Architectural Design Another aspect that contributes to the practical applicability of the method is the load of work products that has to be produced. In MeDUSA's *First Edition*, the *Subsystem Identification* task defined four distinct (intermediate) work products for each individual subsystem, and the *Subsystem Design Modeling* defined another two; while the outcome of consolidating the active objects was indeed not even documented, as stated above. In MeDUSA's recent version, where *Subsystem Consolidation* is performed directly after *Subsystem Identification*, the work results of consolidating the active and passive design objects is clearly and explicitly documented within four consolidated work products. On the other hand, the four intermediate work products, defined as outcome of *Subsystem Identification* might be omitted, if an experienced *System Architect* may decide to develop consolidated versions of the diagrams directly.

Improved Consistency of Detailed Design Besides having moved the consolidation of the passive design objects into *Subsystem Consolidation* within *Architectural Design Modeling*, *Detailed Design Modeling* was subject to another change. That is, while a structural detailed design had been developed as part of *Class Design Model-*

ing, a corresponding detailed design for the behavioral aspects had not been defined by MeDUSA's first edition. While internal object behavior has always been captured during *Intra-Object Behavior Modeling*, this had never been updated to incorporate any changes imposed by the consolidation of active and passive design objects, so no up-to-date detailed behavioral design specification could be given as input to adjacent code generation and implementation tasks. The second edition of MeDUSA reflects this by introducing a *Behavioral Detailed Design Modeling* task, while renaming *Class Design Modeling* into *Structural Detailed Design Modeling* respectively.

9.2 Practical Evaluation Results of the Second Edition

It is clear that - not least because of the timely closeness to its publication - a deep and intense evaluation of the current *Second Edition* of MeDUSA has not been conducted. However, an initial pilot project has been initiated at ABB Automation Products GmbH with this intention. First experiences, gathered from this, will be discussed in the following.

Understandability and Applicability What has been stressed as positive by all participating developers is the systematics and consistency of the method. In particular the now clear definition of how to map design concepts to ANSI-C constructs has found broad acceptance, as it improves the applicability of the method and helps to create confidence and trust in having models as the predominant engineering artifacts. This is especially of importance to those developers, which are not familiar with model-based engineering.

The concise definition of the employed model structures by means of instance specifications contributes to this as well, it also helps to sharpen the awareness that not the produced UML diagrams but the underlying models are the predominant work products. It has further been appreciated that MeDUSA restricts itself to an essential set of UML diagram types, and within those to a restricted use of the offered UML concepts. This simplifies learnability and ensures a better applicability, especially for those developers not already familiar with the UML. The intense use of taxonomies, which is prescribed by MeDUSA, has as well found broad acceptance, as it supports to break down the problem scope in a systematic way, guided by embedded & real-time related concepts. This has been assessed as a major contribution with respect to the practical applicability of the method.

Learnability and Seamlessness What experience has shown additionally in the context of the *MeDUSA Object Taxonomy* is that inexperienced developers tend to identify at first too many analysis objects when applying the taxonomy, resulting in a very clustered *Analysis UML Model*. Especially entity objects are often modeled too fine grained, often representing only single primitive values and no data capsules. While this seems to be a problem related to a lack of experience at the first glance, it is as well a clear indicator that the paradigm shift between the data-function oriented view, imposed by the producedural implementation languages, developers traditionally face in their everyday work, and the class-based design that is developed by means of MeDUSA. Problems with the demarcation of entity and application-logic objects, which have been observable, seem to be another indicator for problems related to this paradigm shift.

Naturally, this raises the question of how good the seamless transition from detailed design into the procedural implementation, which has been formulated as a central goal for the method (cf. Section 4.3) is indeed achieved. Here, it has to be stated that a class-based design, as facilitated by MeDUSA can indeed be seamlessly transferred into a procedural implemenation, what can be demonstrated by the straight-forwardness and simplicity of the *MeDUSA Code Generation Schema*. However, while a seamless transition from design into source code can be achieved, the backflow seems to be more demanding. That is, learning to break down the problem domain into objects rather than data and functions is indeed - as outlined before - a severe paradigm shift that has to be learned. On the other hand, this increased learning effort offers the advantage of having a systematic, concise and traceable way of proceeding from a set of requirements, which has been gathered in terms of use cases, to a sustainable and adequate design. Possibly, additional guidelines and metrics could be offered to support developers in fulfilling the paradigm shift quicker and easier.

Adequacy and Operability A question that has often been raised is whether the bottom-up development approach, which has been incorporated into MeDUSA is adequate for development in the targeted application domain. Traditionally, developers in the respective domain had adopted some sort of top-down development approach. Having defined a coarse grained architecture on the basis of the given requirements, the individual subsystems had then been developed independently. This procedure had several significant drawbacks. At first it showed that because a concise system architecture had not been defined, as interfaces between vaguely defined subsystems had not been specified explicitly, and as further no behavioral specification for the overall system was formulated, massive integration problems occurred with great certainty. The same holds for performance problems, which were not noticed before integrating the overall system. Further, as the approach did not derive the system architecture from the requirements in a systematic and concise way, traceability of design decisions was always hard to achieve.

The bottom-up approach, as it is proposed by MeDUSA, in turn emphasizes the importance of a systematic procedure, by performing *Requirements Modeling* and *Analysis Modeling* to derive a system architecture in a concise way, following full traceabil-

ity of all design decisions. While this procedure does appeal through its systematics, it has the drawback that distributed development is enabled at a relatively late point in time, not sooner as after having defined the system architecture by completing the *Architectural Design Modeling*. But indeed, the point in time, where the system architecture has been consistently defined, is the single point in time, at which successful distributed development may at first be started, as only based on a concise system architecture definition, a successful integration can later be achieved. In turn, if the system architecture is not developed systematically based on the given requirements, then it will most likely not remain stable, so that a successful integration is again not possible. In the domain of embedded & real-time systems, having stringent non-functional requirements, it seems that within the tradeoff between maximizing the contingent of distributed development and minimizing the risk of later integration problems, the later has to be prioritized.

A question closely related to this is how deeply *Requirements Modeling* and *Analysis Modeling* have to be performed for those parts of the software system, which are definitely covered by existing subsystems. In a top-down procedure this problem does of course not arise, as an existing subsystem has to be integrated with its existing interfaces into the the overall system architecture. The problem of its integration however still holds. Even worse, no resilient judgement about the adequacy of an existing subsystem can be made in the context of a top-down procedure, as there is no basis to assess the adequacy of a given subsystem to fulfill the given requirements. On the other hand, if *Requirements Modeling* and *Analysis Modeling* are consistently performed, the subsystems' interfaces, via which they are integrated to an overall system architecture, are systematically derived. The integration of an existing subsystem can thus be regarded as a constraint to the dividing and consolidation of objects, as it is performed during *Subsystem Identification* and *Subsystem Consolidation* respectively. This way, the adequacy of a given subsystem can be judged and necessary adoptions or customizations can be thoroughly derived. Above raised question can thus be answered as follows: *Requirements Modeling* and *Analysis Modeling* should be performed also for those parts of the overall system, which are intended to be covered by existing subsystems.

While being thus controversially discussed, a bottom-up development approach, as it is incorporated into MeDUSA thus seems to be adequate within the given application domain. While being especially useful for projects, where the problem domain is not yet completely understood, this approach of course may introduce some overhead, an experienced developer, completely aware of the problem domain, would like to avoid. However, having the given rather stringent non-functional constraints, a smooth integration and a complete traceability of all performed design decisions seems to be ultimately of higher importance.

Conclusion The results of the preliminary evaluation may thus be interpreted as follows: due to its systematics and its elaborateness, a good understandability and applicability, as well as a good learnability may be unambigously stated to MeDUSA. And not least because of its detailedness with respect to the included model instance

specifications and code generation schema, a general suitability with respect to seamlessness and operability may as well be attributed to it. Two major hindrances however can be identified. The first is the rather impacting paradigm shift, which is imposed by MeDUSA with respect to traditional software engineering practices within the target application domain. The second is the discrepance between MeDUSA's bottom-up procedure and the traditionally implied top-down approaches. Nevertheless, with respect to the formulated goals (cf. Section 4.3), both discrepancies seem to be somehow unavoidable, as the traditional engineering paradigms and procedures have revealed themselves to be unappropriate on the long term.

Chapter 10

Assessment of ViPER

To assess a software system in terms of its quality, a model as defined by Boehm [BBL76] or by the IEC/ISO 9126 [IEC01] standard can serve as a good guidance for a systematic and structured evaluation.

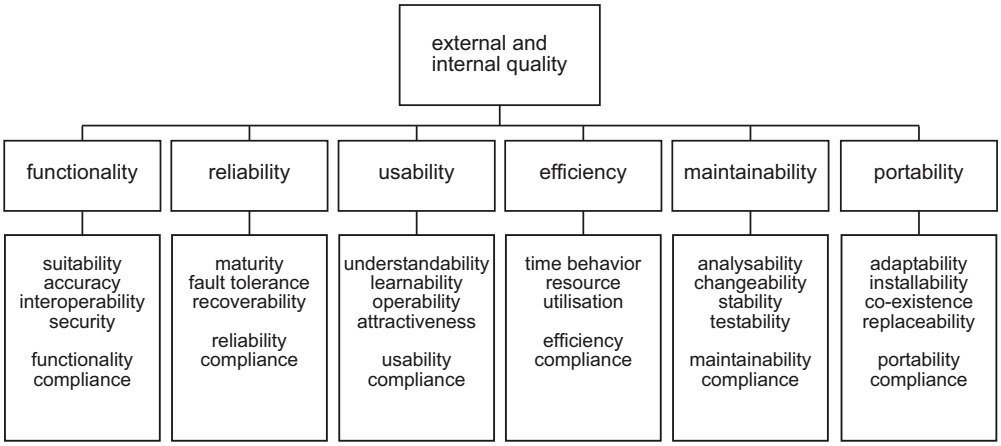


Figure 10.1: Quality Characteristics according to IEC/ISO 9126 (cf. [IEC01])

The IEC/ISO 9126 quality model for instance, which is depicted by Figure 10.1, defines six key quality characteristics, namely *Functionality*, *Reliability*, *Usability*, *Efficiency*, *Maintainability* and *Portability*, together with a set of sub-characteristics for each of them, which allows to break down the evaluation of a software to answering the question of conformity to the respective characteristics respectively sub-characteristics. However, in the context of ViPER this might not lead to satisfying results.

One reason therefore is that ViPER is strongly based on Eclipse [Eclipse] as well as Eclipse-related technology, so that the compliance to several of those quality characteristics is indeed strongly influenced by the underlying platform and the employed frameworks. The usability of the UML graphical editing capabilities is for example

determined to a large extend from what is offered by the underlying GEF framework [GEF], which is used for its realization. The same holds for other characteristics as well. The performance of the code generators is for instance strongly based on what is achieved by the underlying *openArchitectureWare* [GMT] transformation engine. Portability is of course strongly imposed by the underlying Eclispe platform.

Further, as ViPER is a research prototype, which is continuously enhanced to evaluate recent research experiences and to demonstrate the applicability of new technologies and ideas, a coherent set of requirements, against which compliance could be evaluated, has never been explicitly formulated. The tool also has never been intended to reach product quality, so that it is of course hard to assess and evaluate ViPER, using the same quality criteria and benchmarks, as they have been defined for commercially available software, in particular with respect to functionality and reliability. Additionally, most of the defined quality characteristics of the ISO 9126 standard seem to be hard to assess in an objective and quantifyable manner.

Due to this, an itemization of the compliance of ViPER to each defined characteristic, providing a more or less subjective estimation on each, is not provided here. Instead, as most of the relevant quality characteristics, not being directly inferrable from the underlying frameworks, are strongly affected by the software's *inner quality*, i.e. its logical architecture and detailed structure, an in-depth investigation of ViPER's architecture and source code structure is provided, which is regarded to be more enlightening; it also has the nice property that it can be evaluated in a relatively objective way using defined and approved coupling and cohesion metrics (which of course have to be interpreted, so even this bears some inherent subjectivism). As the development process, which is applied to develop a software system, may as well give good guidance to assess some of the quality characteristics of the resulting software, the process and its supporting infrastructure, applied to the development of ViPER, are also shortly characterized afterwards.

10.1 Software Structure & Complexity Evaluation

Having started in July 2004 as a mere experiment to evaluate Eclipse related EMF and GEF technologies in the form of a simple UML state machine diagram editor, which was bundled into a single plug-in and accounted for about 3800 lines of source code, ViPER has grown into a rich and extensive tooling environment in the following years. It up to now bundles 44 plug-ins and subsumes more than 200,000 lines of code, incorporating the contributions of 12 developers (scientific staff as well as students) over the years.

Table 10.1 provides detailed indications on current sizings for the different features, subsumed by the ViPER IDE[1]. All figures were measured using the *SonarJ Architect* tool [Jan07], which was also used to define and check the logical architecture.

[1] The figures presented reflect the state of the ViPER IDE version 1.0.0 RC2, integration build I200808181247.

Size Metric (Number of)	Platform			UML2			NaUTiluS		MetiS	Total
	Core	VMF	MTF	Core	VME	CodeGen	Core	Simulation		
Lines of Code[a]	5878	32906	1354	2243	49640	46980	38690	7866	15263	200820
(generated)	-	(14408)	-	-	-	(45692)	(24900)	(3703)	-	88703
Statements	2248	12924	560	1057	23229	24512	15842	3083	6392	89847
(generated)	-	(6122)	-	-	-	(23918)	(10261)	(1433)	-	41734
Types	126	507	24	57	697	318	632	202	266	4240
Abstract Types	39	169	8	1	51	90	88	32	36	506
Impl. Types[b]	105	369	24	56	654	235	567	177	247	2434
Methods	665	3316	123	207	2829	3741	3864	872	1611	17228
Impl. Methods[c]	614	2760	111	205	2827	2990	567	766	1439	12279

[a] Lines of code that are not a blank or comment line, regardless of the number of statements
[b] Types with Implementation
[c] Methods with Implementation

Table 10.1: ViPER IDE - Size Metrics Evaluation Results

According to it, the ViPER IDE source code base measures 200,820 lines of Java code, comprising 89,847 Java statements (of course not including any third-party contributions). Out of those, 88,703 lines of code respectively 41,734 statements are related to source code, which is generated from EMF-based meta-models, leaving a portion of 112,117 lines of handcraftet code.

It has to be mentioned that those figures of course only count for java source code and do thus not take into account non-java languages, as the *xTend* and *xPand* script languages (cf. [GMT]), which are used within *ViPER UML2 CodeGen* or *ViPER MetiS* to realize the built-in code generators, or even grammar files, which are used within *ViPER UML2 VME* to specify the contained parsers. However, with a resultant average size of 2774 lines of code per plug-in (not counting those plug-ins containing only generated source code), modularity of the ViPER IDE java source code base may be regarded as quite adequate at this point.

While not interpreting those figures further, they may give good guidance on the rather large complexity of the tool, even if only accounting the manually crafted lines of code. This may be further illustrated by taking those 112,117 manually crafted lines of code as input to a COCOMO II [BHM+00] cost and effort estimation. Assuming nominal values for COCOMO's scale and cost drivers, a calculation with the COCOMO II model yields the following results:

Effort: 528.4 Person-Month
Duration: 26.9 Month
Average Staffing: 19.6 Persons

They impressively show that - being a mere research prototype - the ViPER IDE may indeed be regarded as a quite complex piece of software. Comparing the calculated effort and estimations to the actually spent effort, may give another slight indication on the fact that it would be unappropriate to compare its quality characteristics compliance to that of commercially developed software.

Nevertheless, while cutting back on functionality, usability or even efficiency, obtaining a clear and concise software structure has always been facilitated within the ViPER project to achieve maintainability and portability. Therefore, the logical architecture of the ViPER IDE has been thoroughly defined. As depicted by Figure 10.2, this has been done by means of the *SonarJ Architect* tool. While Eclipse's built-in plug-in dependency mechanisms also offers means to define a logical architecture based on plug-in and feature dependencies, such a separate definition is a reasonable means, as it allows to define the logical architecture in a more explicit manner; feature and plug-in dependencies within Eclipse have to be specified for each plug-in and feature separately, so that the overall architecture is indeed only specified indirectly by the combination of all inter-dependencies. Using a tool like the *SonarJ Architect* it is further possible to check conformity of source code to a specified logical software architecture and to assess a given logical architecture by means of metrics.

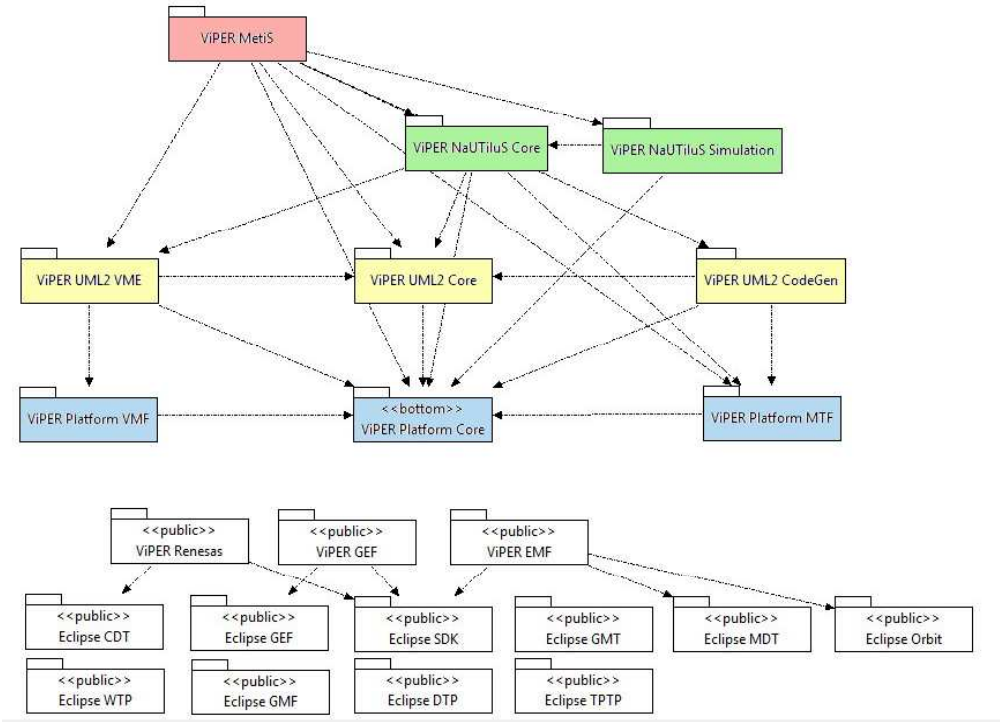


Figure 10.2: ViPER IDE Architecture Definition within SonarJ

Evaluating the ViPER IDE source code by means of the *SonarJ Architect* tool, it can be directly inferred that there are no architecture violations, meaning that the ViPER IDE source code base does indeed comply to the logical architecture specified within Figure 10.2. That is, being stereotyped as *public*, those features bundled by the *ViPER IDE ThirdParty* umbrella feature may be accessed in an arbitrary way. For all non-*public* features, dependencies are explicitly specified, meaning that a dependency arrow indicates that types of the supplier feature may be accessed from the client feature[2].

[2]It has to be mentioned that those legal dependencies between the *ViPER MetiS* feature and the *ViPER NaUTiluS* features are actually not established on the level of source code yet, as there as no MeDUSA

Coupling & Cohesion Assessing the logical architecture - and thus its realizing source code - can be done by evaluation of coupling and cohesion metrics. The results of such an evaluation are depicted within Table 10.1. The measured *relational cohesion* indicates the cohesion between all types within a given assembly, by measuring the average number of internal relationships per type. While a very low relational cohesion might indicate that cohesion within an assembly is too low, a high relational cohesion might point to a too high coupling between the types of the assembly. In [Mar06] a value between 1.5 and 4.0 is regarded to be *good*, while values outside this range are regarded to be potentially problematic.

Coupling & Cohesion	Platform			UML2			NaUTiluS		MetiS
	Core	VMF	MTF	Core	VME	CodeGen	Core	Simulation	
Relational Cohesion[a]	1.57	4.21	2.42	1.25	2.07	7.98	4.25	2.79	2.8
Afferent Coupling[b]	302	364	19	38	2	2	58	0	0
Efferent Coupling[c]	267	565	93	146	527	275	496	407	411
Abstractness[d]	0.31	0.33	0.33	0.02	0.07	0.28	0.14	0.16	0.14
Instability[e]	0.47	0.61	0.83	0.79	1.0	0.99	0.9	1.0	1.0

[a] The ratio of the number of internal type relationships to number of types within this assembly
[b] The number of types outside this assembly that depend on types within this assembly
[c] The number of types inside this assembly that depend on types outside this assembly
[d] The ratio of the number of abstract types to the number of total types within this assembly
[e] The ratio of efferent coupling to total coupling (efferent coupling + afferent coupling)

Table 10.2: ViPER IDE - Coupling & Cohesion Metrics Evaluation Results

Applying those guidelines to the ViPER IDE source code, a too strong relational cohesion within the *ViPER UML2 CodeGen*, *ViPER Platform VMF*, and *ViPER NaUTiluS Core*, as well as a slightly too low relational cohesion within the *ViPER UML2 Core* feature may be noticed. Investigating this in detail unveils that the rather strong coupling within the *ViPER UML2 CodeGen* feature is caused by the high relative cohesion (8.53) of the source code generated automatically for the EMF-based ANSI-C intermediate model that is used for its realization (cf. [FNL08] for details), which makes up 97% of the overall source code or respectively 92% of the comprised types. The same holds for the *ViPER NaUTiluS Core* plug-in, which contains two generated EMF models, out of which the largest one, which makes up 17799 lines of code, has a relatively high relative cohesion of 4.54, as well as for *ViPER Platform VMF*, which contains an EMF model with similar properties (14408 lines of code, relational cohesion of 4.49). The rather low relative cohesion within the *ViPER UML2 VME* can be explained with the fact that this feature somehow acts as sort of a class library, bundling a lot of rather unrelated utility classes. Conclusively all three indications do indeed seem to be *false positives* in a sense that they do not point to actual weaknesses in terms of cohesion.

Abstractness & Instability To detect assemblies which are hard to maintain, *abstractness* and *instability* may be measured. While abstractness should give a guidance on the extensability of an assembly (a completely abstract package is best extensible),

specific task wizard that integrates the functionality of *ViPER NaUTiluS*. However, as it is intended to develop respective MeDUSA support in the future, the definition of the logical architecture does already reflect this.

instability intends to infer the resilience of an assembly to change by investigating its outgoing dependencies (if an assembly does not rely on any other assemblies, it is regarded to be totally stable).

While both metrics themselves can only give limited guidance to assess the maintainability of an assembly, their combination can lead to more reasonable results. Based on the provided definitions of abstractness and instability, one can conclude that completely concrete and stable assemblies (A=0, I=0) are hard to maintain, because they are hard to extend and a lot of other assemblies depend on them, while completely abstract and instable assemblies (A=1, I=1) are potentially useless, because they offer mostly abstract functionality and not many other assemblies rely on them. It is thus desirable to design assemblies that achieve a good *balance* between their abstractness and stability, staying thus out of the *zone of pain* and the *zone of uselessness* (cf. [Mar94]). An assembly is therefore regarded to be adequately balanced if its distance to the *main sequence*, which is a straight line connecting the points (I=0, A=1) and (I=1,A=0), is rather low.

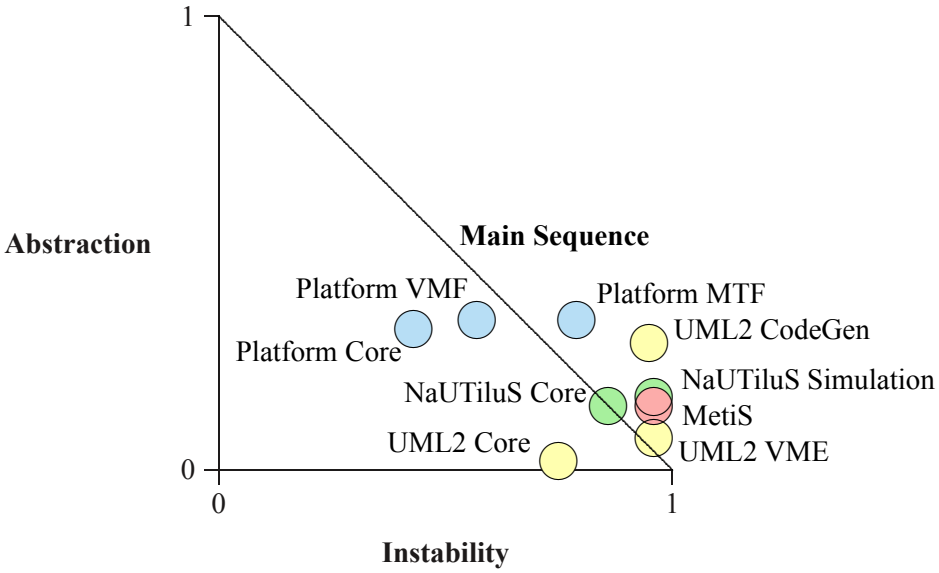


Figure 10.3: Abstraction-Instability Graph for ViPER IDE Features

Figure 10.1 shows the relation between abstraction and instability for all features of the ViPER IDE. It demonstrates that all ViPER IDE features are more or less adequately *balanced*, having only slight distances to the *main sequence*. Naturally, the *ViPER Platform* features have the highest abstraction, as they provide abstract base classes, which are used within the other features, while those features that deliver end-user functionality are inherently more concrete. The features having the highest distance to the main sequence are *ViPER Platform Core* (-0.22), *ViPER UML2 Core* (-0.19), and *ViPER UML2 CodeGen* (0.28). While the rather too low abstraction of the *ViPER UML2 Core* feature can again be explained with the fact that it basically bundles a library of concrete utility classes, the rather too high *abstraction* of the *ViPER UML2 CodeGen* feature can be explained again with the properties of the EMF generated code that it comprises (all abstract types within the feature are actually generated).

The *unbalance* of *ViPER Platform Core* feature interestingly points to a minor weakness within its design. Indeed *ViPER Platform Core*, which is the single *bottom* plug-in within the logical architecture of the ViPER IDE, bundles several unrelated utility classes, which are concrete, and which are used throughout the ViPER IDE. Moving those utility classes into separate *public* features in the style of *ViPER EMF* and *VIPER GEF*, the abstractness of *ViPER Platform Core* could be increased with respect to its instability, thus improving its balance. It thus seems that, due to its role as a *bottom* feature, the *ViPER Platform Core* feature is being misused - at least to some extend - as some sort of collecting tray for globally used utility classes. Nevertheless, a real problem in terms of maintainability does not seem to be related to this, as those classes are relatively unrelated and could be easily moved apart.

10.2 Development Process & Infrastructure Characterization

The ViPER IDE as well as all other related components are developed by means of an agile development process - pretty much inspired by the Eclipse development process [GW05]. In an university setting, as it is faced here, this seems to be an adequate procedure for a number of reasons. At first, most of the functionality, which is to be realized, is actually related to current research experience, so that on the one hand requirements are not immanently clear in advance, while on the other hand direct and fast feedback is required for the sake of evaluation. Both demands for a certain flexibility within the applied development process. Further, the schedules of the involved developers cannot always be rigidly planned, as the minority of developers is actually employed as full-time scientific staff, and large contributions are performed by student workers or respective diploma or master thesis students. Last, as a certain education aspect, which is inherent to a master or diploma thesis work, is also not neglectable, the application of agile techniques like pair programming have proven to be very useful as well.

Development of the ViPER IDE is based on milestone release cycles of four to eight weeks, dependent on the current team formation. This may be strongly fluctuant, due to the relatively large number of involved diploma and master thesis students, who attend the project only for a limited period of time. Milestone planning is done within a weekly conducted project meeting, attended by all participating developers (scientific staff and involved students). An appraisal of the weekly integration build, being headlessly built in advance to the meeting, is also conducted in this context. All project related documentation, including the development plan and coding or style conventions, is maintained online within a *MediaWiki* installation [ViPERc], so it can be accessed and maintained easily. Detailed planning of features, as well as tracking of bugs and change requests is as well performed online, organized by means of a dedicated *Bugzilla* installation [ViPERa]. Mailing lists are also used to coordinate the flow of information within the community.

The development of the ViPER IDE further is strongly based upon the principles of *continuous integration* and *continuous testing*. For this purpose a headless build system has been established in terms of the related *ViPER IDE RelEng* project component (cf. [ViPERc]). It executes nightly snapshot as well as weekly integration builds, based on the respective contents of the ViPER source code repository. The code base is compiled, assembled, and published on a respective ViPER download site [ViPERb], together with detailed build results. The build system further checks conformity of the source code base to predefined style guidelines and detects violations to predefined access restrictions. It further executes automated JUnit tests [GB99] and measures the achieved code coverage. The results of all checks and tests are as well published on the ViPER download site together with the build results, in the form of a quick overview for the overall platform, as depicted by Figure 10.4, as well as a short summary and a detailed report for each plug-in.

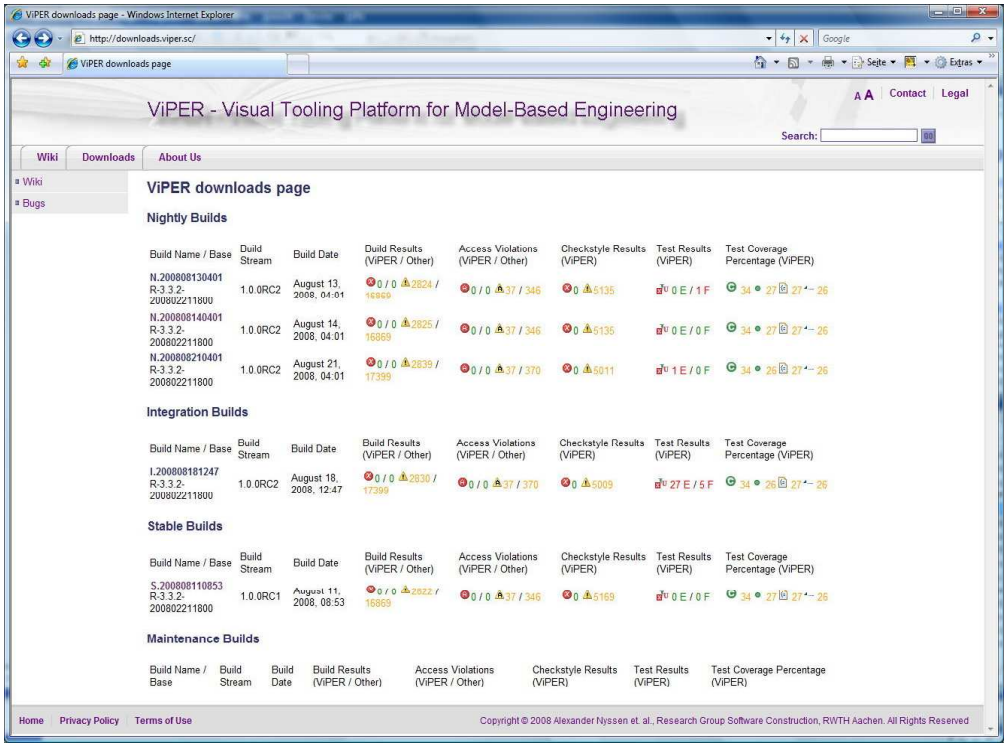


Figure 10.4: ViPER Download Site (Screenshot)

Currently, the *ViPER IDE Tests* project component, which bundles all ViPER IDE related test cases, comprises 318 JUnit test cases, which yield a source code coverage of 34% on the level of classes, 26% on the method level, and 26% on the line level. It is naturally clear that this rather low coverage may not be regarded as sufficient. Investigating it in detail, it can be stated that it is caused to the most extend by a lack of automated test cases for those parts related to the code generation (which affects *ViPER MTF* and *ViPER UML2 CodeGen*), as well as those parts related to graphical editing (which affects *ViPER Platform VMF*, *ViPER UML2 VME*, and *ViPER NaUTiluS*). While in the case of code generation this is explainable because of human resource

limitations (the respective functionaliy has been developed - and tested manually - before continuous automated testing was established within the ViPER project, and automated test cases have not been developed afterwards), in the case of graphical editors, the lack of automated JUnit tests is because of technical difficulties. Effort has been spent to evaluate the applicability of an Eclipse TPTP provided automated GUI recording tool (compare [TPTP]), which could be applied to test respective functionality more adequate [Sim08]. While a general feasibility could be demonstrated, the recorded test cases are not sufficiently robust to test GEF applications (all actions currently have to be recorded in a position-based manner so that small changes to the GUI layout causes the complete test suite to be stale), so its integration into the continuous testing environment has - due to the high effort that is related - been postponed until more stable test cases can be captured (contributions to enhance the GEF-testing capabilities of the auto GUI recording tool have been submitted to the TPTP project but have not been integrated yet).

While thus being far from optimal, the applied development process, and in particular the supporting development infrastructure are regarded to be approved and mature. It is believed that its effect on the quality of the ViPER tool is strongly positive and that realizing a development project of equal size and complexity would otherwise not be possible in an university setting, as it is faced here.

Chapter 11

An Appraisal of Achieved Results

11.1 Goal Attainment

Having evaluated MeDUSA and ViPER individually, it is about time to investigate, to which extend the goals, formulated in Section 4.3, may be regarded as accomplished.

Methodological Integrity The basic goal of the herein presented approach has been formulated within Section 4.3 as that of *Methodological Integrity*, which was understood to denote the delivery of *"an overall software construction methodological approach in terms of a concise method, an appropriate notation, and adequate tool support, being all related by common concepts and principles."*.

Having quite intensely demonstrated and evaluated all parts of the methodology in the preceding chapters, it might be stated that fulfillment of this central goal may be certified to the largest extend. As outlined already before, MeDUSA, in contrast to its predecessors *COMET* or *ROOM*, or even other state-of-the-art approaches, was explicitly designed as a *construction method* to meet the defined sub-goal of *Methodical Completeness*. By explicitly covering all *Implementation* tasks, and by providing a detailed code generation schema for the predominant implementation language within the target domain, namely ANSI-C, the method explicitly covers all software construction steps, from the early requirements modeling up to the resulting implementation.

By offering graphical UML editors as well as textual (use case) editors to support the modeling related tasks of MeDUSA, by providing customizable and flexible code generators, and not last by incorporating dedicated methodical support, the ViPER tool - even if its functionality is yet limited - demonstrates that *Methodological Integration*, the second formulated sub-goal, can as well be achieved, but only if method and tools are integrated via common concepts and principles. The principle of MeDUSA to be use case-driven, is for instance also incorporated into the ViPER IDE, in particular by offering extended support for specifying use case details and for the simulation

of use case models, as offered by ViPER NaUTiluS. The class-based characteristic of MeDUSA, is not last reflected by offering customizable and flexible ANSI-C code generators, as well as a full C/C++ development environment. The close integration that is achieved by means of methodical support, as offered by ViPER MetiS, strongly contributes to this as well, even while still providing limited support for some tasks of the MeDUSA method yet.

Constraint-Adequateness Adequacy with respect to technical and organizational constraints was formulated within Section 4.3 as the second major goal to the herein presented approach, as without explicit *Constraint-Adequateness*, applicability within the target domain is not achievable.

To meet to the very special technical constraints, which are predominantly imposed by those strong resource constraints, being faced in the context of small embedded & real-time systems, MeDUSA was explicitly designed to be a class-based method (compare characterization provided in Section 6.3). That is, by not facilitating object-oriented or even component-based technology, a resource preserving implementation within the ANSI-C target language is possible.

Meeting the technical constraints is further facilitated by explicitly investigating timing and concurrency constraints. That is, the taxonomies and their implementing UML profiles provided by MeDUSA enable the explicit modeling of timing and concurrency constraints, not only during *Analysis Modeling* and *Architectural Design Modeling*, but as well during *Requirements Modeling*. A continuous real-time analysis, as it is incorporated into the MeDUSA method as well contributes to this. That is, by explicitly modeling timing and concurrency constraints, and by analyzing their adherence continuously, an adequate design, appropriately reflecting the technical restriction within the domain, can be developed.

The organizational adequacy of the herein presented approach has to be reflected a bit more critical. That is, while the employment of a standardized modeling language allows to use market available modeling tools, the learning effort and related economical effort, caused by its introduction, may not be underestimated. That is, as experiences with the practical application of MeDUSA within several pilot projects unveiled, accessibility to such a high abstraction language is not straight-forward for most of the domain experts, which are educated in the application of procedural implementation languages or even machine code, and which thus have to perform a paradigm shift when adopting a model-based engineering approach, as it is facilitated herein.

It may as well be discussed - regarding organizational adequacy - how well distributed development is facilitated by the presented approach. As already stated before (cf. Section 9.2), this question is closely related to the discussion on whether a bottom-up development procedure, as facilitated by MeDUSA, or a top-down development approach is applied. As it has been discussed in detail before, the bottom-up procedure, which is regarded to be best adequate, has the drawback of restricting the amount of fully distributed development to the *Detailed Design* and *Implementation* phases.

However, while from the viewpoint of organizational adequacy this might be regarded as a sort of drawback, the necessity of a feasible design and a seamless integration seem to outweigh the reduced parallelization potential.

11.2 Conclusion & Outlook

To sum it up, a general attainability of the goals, formulated within Section 4.3 may be certified. While an ultimate practical evaluation of the MeDUSA method in an industrial size project is still outstanding, and while the functionality of the ViPER tool is yet restricted, also in terms of its currently offered dedicated methodical support, the MeDUSA-ViPER methodology seems to be a quite promising approach. It was explicitly designed to meet the special characteristics, inherent to the software construction of embedded & real-time systems within marginal application areas as the industrial automation. And while there are some slight drawbacks regarding the organizational adequacy of the presented approach, it demonstrates that technical constraint adequacy and methodological integrity and completeness can be achieved.

As such, the MeDUSA-ViPER approach demonstrates that modern, model-based engineering technology and techniques can as well be introduced to those rather marginal application areas. In fact, if model-based engineering is methodologically integrated and regards the application area specific constraints, it is probably the superior development paradigm due to its increased systematics and traceability, as well as its increased potential in terms of reasoning and analyzability. What has however to be clearly stated is that - even if advantages of such a model-based engineering approach are convincing and non deniable - its introduction may at first be rather costly, as the introduction of a new development paradigm is not achievable without spending considerable effort. Stamina and patience may thus be necessary when introducing such an approach. Earnings may on the other hand be outstandingly capacious.

While the MeDUSA-ViPER methodology, presented in this thesis, has already gathered some fundamental ground, further work remains to be done. For MeDUSA, evaluation in an industrial size development project would be the logical next step, after the current evaluation within a rather small size pilot project has been concluded. Regarding ViPER, improvements could in particular be incorporated by extending the dedicated methodical support. Here, research effort within the Research Group is currently spent on the integration of automated real-time analysis capabilities [Rit08], which is regarded to be of outstanding importance in the context of embedded & real-time systems.

Despite the continued development of MeDUSA and ViPER, further research may in particular be identifiably with respect to languages and tools. Especially improvements to further enhance the traceability and analyzability of the employed models is a very promising research field. The simulation and debugging of models may be predominantly named in this context, which could significantly improve the overall quality of the employed models, and would allow to infer earlier and more concise feedback

about the design decisions, being made. It would further help to improve acceptance of higher abstracting modeling languages and would facilitate to move most of the validation effort from implementation into earlier lifecycle phases.

The inception of domain specific languages and frameworks can as well be named in this context. While the OMG has already spent some effort on languages and profiles, targeting the rather broad domain of embedded & real-time systems as a whole (cf. [OMG07b] or [OMG07a]), languages and frameworks dedicated to certain specific application areas can only seldomly be found. While first efforts are observable within the key application areas - for instance AUTOSAR or AADL (cf. Section 3.2.1) - marginal application areas as the industrial automation have up to now been mostly disregarded. Especially with respect to the very special organizational constraints being faced there, the integration of domain specific concepts could however bring a significant improvement in terms of accessibility and learnability.

Regarding methodical aspects, further research will have to be spent on investigating also those software engineering disciplines, apart from software construction. Especially software quality management of embedded & real-time systems is a field that bears a lot of research potential. Testing may be regarded as the most challenging field in this context, as it is also strongly affected by the very special technical constraints that are being faced in the respective domain.

Despite broadening the scope with respect to the covered software engineering disciplines, further research will also have to be spent in terms of delivering profound processes for the overall device development. So called *Systems Engineering*, the integration of software, mechanical, and electronical hardware engineering in an interdisciplinary manner actually is a broad field of activity. While it has already been a question for quite long (first approaches related to systems engineering may already be found in the 1950's and 1960's, as elaborated within Section 3.2.1), real breakthrough approaches may not be named. However, it seems that - not last because of the recent incentive of the OMG related to their systems modeling language (SysML) [OMG07b] - this research area seems to have significantly gained new momentum.

Bibliography

[ABB+02] Colin Atkinson, Joachim Bayer, Christian Bunse, Erik Kamsties, Oliver Laitenberger, Rolan Laqua, Dirk Muthig, Barbara Paech, Jürgen Wüst, and Jörg Zettel. *Component-based Product Line Engineering with UML*. Addison Wesley, 2002.

[ABGP05] C. Atkinson, C. Bunse, H.-G. Gross, and C. Peper, editors. *Component-Based Software Development for Embedded Systems - An Overview of Current Research Trends*. Springer Verlag, 2005.

[AHD04] The American Heritage Dictionary of the English Language, Fourth Edition. Houghton Mifflin Company, 2004. `http://www.bartleby.com/61` (accessed: September 17, 2008).

[AHL+03] Hedley Apperly, Ralph Hofman, Steve Latchem, Barry Maybank, Barry McGibbon, David Piper, and Chris Simons. *Service- and Component-based Development: Using Select Perspective and UML*. Addison Wesley, 2003.

[AKZ96] Maher Awad, Juha Kuusela, and Jürgen Ziegler. *Object-Oriented Technology for Real-Time Systems*. Prentice Hall, 1996.

[Amb02] Scott W. Ambler. *Agile Modeling - Effective Practices for Extreme Programming and the Unified Process*. John Wiley & Sons, 2002.

[ATE07] The ATESST Consortium. *Advancing Traffic Efficiency and Safety through Software Technology - Project Overview*, 2007. `http://www.atesst.org/home/liblocal/docs/ATESST_Overview_2007Q1.pdf` (accessed: September 17, 2008).

[Bar] Michael Barr. Embedded Systems Glossary. `http://www.netrino.com/Embedded-Systems/Glossary` (accessed: September 17, 2008).

[Bau75] Friedrich Ludwig Bauer. *Software Engineering*. Springer Verlag, 1975.

[BBL76] Barry W. Boehm, J. R. Brown, and M. Lipow. Quantitative Evaluation of Software Quality. In *Proceedings of the 2nd International Conference on Software Engineering*, pages 592–605, San Francisco, CA, USA, 1976.

[BBR+05] Andreas Bauer, Manfred Broy, Jan Romberg, Bernhard Schätz, Peter Braun, Ulrich Freund, Nuria Mata, Robert Sandner, and Dirk Ziegenbein. AutoMoDe - Notations, Methods, and Tools for Model-Based Development of Automotive Software. SAE document number 2005-01-1281, 2005. `http://www.validas.de/pub/SAE-05AE-268.pdf` (accessed: September 17, 2008).

[BCK03] Len Bass, Paul Clements, and Rick Kazman. *Software Architecture in Practice (Second Edition)*. Addison Wesley, 2003.

[Bec00] Kent Beck. *Extreme Programming Explained - Embrace Change*. Addison Wesley, 2000.

[BFK+99] J. Bayer, O. Flege, P. Knauber, R. Laqua, D. Muthig, K. Schmid, T. Widen, and J.-M. DeBaud. PuLSE: A Methodology to Develop Software Product Lines. In *Proceedings of the 1999 Symposium on Software Reusability (SSR99)*, pages 122–131, Los Angeles, CA, USA, May 1999.

[BHM+00] B. W. Boehm, E. Horowitz, R. Madachy, D. Reifer, B. K. Clark, B. Steece, A.W. Brown, S. Chulani, and C. Abts. *Software Cost Estimation with COCOMO II*. Prentice Hall, 2000.

[BJR96] Grady Booch, Ivar Jacobson, and James Rumbaugh. Unified Modeling Language for Object-Oriented Development. Rational Software Corporation, Documentation Set, Version 0.9, 1996.

[Boo86] Grady Booch. Object-Oriented Development. *IEEE Transactions on Software Engineering*, 12(2):211–221, February 1986.

[Boo91] Grady Booch. *Object-Oriented Design with Applications*. Addison Wesley, 1991.

[Boo94] Grady Booch. *Object-Oriented Design with Applications (Second Edition)*. Addison Wesley, 1994.

[Bos00] Jan Bosch. *Design & Use of Software Architectures*. Addison Wesley, 2000.

[BR95] Grady Booch and James Rumbaugh. Unified Method for Object-Oriented Development, Version 0.8. Technical report, Rational Software Corporation, 1995.

[BS03] Kurt Bittner and Ian Spence. *Use Case Modeling*. Addison Wesley, 2003.

[BW94] Alan Burns and Andy Wellings. HRT-HOOD: A Structured Design Method for Hard Real-Time Systems. *Real-Time Systems Journal*, 6:73–114, 1994.

[CAB+93] Derek Coleman, Patrick Arnold, Stephanie Bodoff, Chris Dollin, Helena Gilchrist, Fiona Hayes, and Paul Jeremaes. *Object Oriented Development - The Fusion Method*. Prentice Hall, 1993.

[Can01] Murray Cantor. RUP SE: The Rational Unified Process for Systems Engineering. The Rational Edge, November 2001. `http://www.ibm.com/developerworks/rational/library/content/RationalEdge/nov01/RUPSENov01.pdf` (accessed: September 17, 2008).

[CD00] John Cheesman and John Daniels. *UML Components - A Simple Process for Specifying Component-based Software*. Addison Wesley, 2000.

[CDT] Eclipse C/C++ Development Tooling (CDT) project site. `http://www.eclipse.org/cdt` (accessed: September 17, 2008).

[CE00] Krzysztof Czarnecki and Ulrich W. Eisenecker. *Generative Programming - Methods, Tools, and Applications*. Addison Wesley, 2000.

[Che67] Harold Chestnut. *Systems Engineering Methods*. John Wiley & Sons, 1967.

[CLL99] Peter Coad, Eric Lefebrve, and Eric De Luca. *Java Modeling in Color with UML*. Prentice Hall, 1999.

[CN02] Paul Clemens and Linda Northrop. *Software Product Lines - Practices and Patterns*. Addison Wesley, 2002.

[Coc00] Alistair Cockburn. *Writing Effective Use Cases*. Addison Wesley, 2000.

[CR06] Eric Clayberg and Dan Rubel. *Eclipse - Building Commercial-Quality Plug-ins (Second Edition)*. Addison Wesley, December 2006.

[CY91] Peter Coad and Edward Yourdon. *Object-Oriented Analysis*. Prentice Hall, 1991.

[CY92] Peter Coad and Edward Yourdon. *Object-Oriented Design*. Prentice Hall, 1992.

[DDH72] Ole-Johan Dahl, Edsger W. Dijkstra, and C.A.R. Hoare. *Structured Programming*. Academic Press, 1972.

[DeM79] Tom DeMarco. *Structured Analysis and System Specification*. Prentice Hall, 1979.

[DFK+04] Jim D'Anjou, Scott Fairbrother, Dan Kehn, John Kellerman, and Pat McCarthy. *The Java Developer's Guide to Eclipse (Second Edition)*. Addison Wesley, 2004.

[Dis04] Pierre Dissaux. Using the AADL for mission critical software development. In *Proceedings of 2nd European Congress on Embedded Real Time Software (ERTS)*, Toulouse, France, January 2004. `http://www.ellidiss.com/erts04.pdf` (accessed: September 17, 2008).

[Do08] Bui Trung Huy Do. Development of a Sequence Diagram Editor for the ViPER Platform, 2008. Diploma Thesis (German), RWTH Aachen University, http://www.swc.rwth-aachen.de/lufgi/teaching/theses/completed/Huy_Do_Thesis_Report.pdf (accessed: September 17, 2008).

[Dou99a] Bruce Powel Douglass. *Doing Hard Time - Developing Real-T-ime Systems with UML, Objects, Frameworks, and Patterns*. Addison Wesley, 1999.

[Dou99b] Bruce Powel Douglass. *Real Time UML - Developing Efficient Objects for Embedded Systems (Second Edition)*. Addison Wesley, 1999.

[Dou07] Bruce Powel Douglass. *Real-Time UML Workshop for Embedded Systems*. Elsevier Newnes, 2007.

[DW98] Desmond D'Souza and Alan Cameron Wills. *Objects, Components, and Frameworks with UML - The Catalysis Approach*. Addison Wesley, 1998.

[Eclipse] Eclipse project site. http://www.eclipse.org (accessed: September 17, 2008).

[EKW92] David W. Embley, Barry D. Kurtz, and Scott N. Woodfield. *Object Oriented Systems Analysis - A Model-Driven Approach*. Yourdon Press, Prentice Hall, 1992.

[EMF] Eclipse Modeling Framework (EMF) project site. http://www.eclipse.org/emf (accessed: September 17, 2008).

[EPF] Eclipse Process Framework (EPF) project site. http://www.eclipse.org/epf (accessed: September 17, 2008).

[ESL05] ESL Now! Online Survey, July 2005. http://www.esl-now.com/pdfs/survey_results.pdf (accessed: September 17, 2008).

[Fav05] Jean-Marie Favre. Foundations of Meta-Pyramids: Languages vs. Metamodels – Episode II: Story of Thotus the Baboon. In Jean Bezivin and Reiko Heckel, editors, *Language Engineering for Model-Driven Software Development*, number 04101 in Dagstuhl Seminar Proceedings. Internationales Begegnungs- und Forschungszentrum fuer Informatik (IBFI), Schloss Dagstuhl, Germany, 2005.

[Fav06] Jean-Marie Favre. Megamodelling and Etymology. In James R. Cordy, Ralf Lämmel, and Andreas Winter, editors, *Transformation Techniques in Software Engineering*, number 05161 in Dagstuhl Seminar Proceedings. Internationales Begegnungs- und Forschungszentrum fuer Informatik (IBFI), Schloss Dagstuhl, Germany, 2006.

[FBH+06] Helmut Fennel, Stefan Bunzel, Harald Heinecke, et al. Achievements and Exploitation of the AUTOSAR Development Partnership. In *Proceedings of Convergence 2006*, Detroit, MI, USA, October 2006. http://www.autosar.org/download/AUTOSAR_Paper_Convergence_2006.pdf (accessed: September 17, 2008).

[Fei07] Peter H. Feiler. AADL in Industrial Pilot Use and Research. Presentation at the 1st International Workshop on Aerospace Software Engineering (AeroSE 07), May 2007. http://crisys.cs.umn.edu/icse-workshop/Presentations/(2)AADL-Users-Feiler-52007.pdf (accessed: September 17, 2008).

[FG+04] Patrick Farail, Pierre Gaufillet, et al. The COTRE Project: How to Model and Verify Real Time Architecture? In *Proceedings of 2nd European Congress on Embedded Real Time Software (ERTS)*, pages 55–64, Toulouse, France, January 2004. http://www.laas.fr/COTRE/work/output/cotre-erts.pdf (accessed: September 17, 2008).

[FNL08] Mathias Funk, Alexander Nyßen, and Horst Lichter. FROM UML TO ANSI-C - An Eclipse-Based Code Generation Framework. In *Proceedings of 3rd International Conference on Software and Data Technologies (ICSOFT)*, pages 12–19, Porto, Portugal, August 2008.

[Fun06] Mathias Funk. Generating Efficient C-Code from UML2 Structure Diagrams, 2006. Diploma Thesis (German), RWTH Aachen University, http://www.swc.rwth-aachen.de/lufgi/teaching/theses/completed/Mathias_Funk_Thesis_Report.pdf (accessed: September 17, 2008).

[Gan06] Jack Ganssle. The Embedded Muse 132, August 2006. http://www.ganssle.com/tem/tem132.pdf (accessed: September 17, 2008).

[Gav02] Anastasius Gavras. MODA-TEL - An IST project on Model Driven Architectures for Telecommunications System Development and Operation. Presentation to the audience of the joint Telecom and MARS task forces during the OMG TC meeting in Helsinki, September/October 2002. http://www.omg.org/docs/telecom/02-10-01.pdf (accessed: September 17, 2008).

[GB99] Erich Gamma and Kent Beck. JUnit - A Cook's Tour, May 1999. http://junit.sourceforge.net/doc/cookstour/cookstour.htm (accessed: September 17, 2008).

[GB03] Erich Gamma and Kent Beck. *Contributing to Eclipse - Principles, Patterns, and Plug-Ins*. Addison Wesley, October 2003.

[GEF] Eclipse Graphical Editing Framework (GEF) project site. http://www.eclipse.org/gef.

[Gei02] Martin Geier. Codegenerierung aus UML nach ANSI-C für kleine Embedded Systeme. Course material, Lecture on UML für verlässliche mobile und eingebettete Systeme, University of Erlangen-Nuremberg, Germany, December 2002. http://www3.informatik.uni-erlangen.de/Lehre/UMLEmbSys/WS2002/folien/12-oo2c.pdf (accessed: September 17, 2008).

[Gen05] Gentleware AG. *Poseidon for UML Embedded Edition - ANSI C Tutorial*, March 2005. http://www.gentleware.com/fileadmin/media/pdfs/tutorials/EmbeddedTutorial.pdf (accessed: September 17, 2008).

[GHH+04] H. Grothey, C. Habersetzer, M. Hiatt, W. Hogrefe, M. Kirchner, G. Lütkepohl, W. Marchewka, U. Mecke, M. Ohm, F. Otto, K.-H. Rackebrandt, M. Schönsee, D. Sievert, A. Thöne, and H.-J. Wegener. *Praxis der industriellen Durchflussmessung*. ABB Automation Products GmbH, Göttingen, 2004.

[GHSY97] Ian Graham, Brian Henderson-Sellers, and H. Younessi. *The OPEN Process Specification*. Addison Wesley, 1997.

[GM57] Harry H. Goode and Robert E. Machol. *System Engineering: An Introduction to the Design of Large-scale Systems*. McGraw-Hill, 1957.

[GMF] Eclipse Graphical Modeling Fframework (GMF) project site. http://www.eclipse.org/gmf (accessed: September 17, 2008).

[GMT] Eclipse Generative Modeling Technologies (GMT) project site. http://www.eclipse.org/gmt (accessed: September 17, 2008).

[Gom84] Hassan Gomaa. A Software Design Method for Real Time Systems. *Communications ACM*, 29(9):938–949, September 1984.

[Gom93] Hassan Gomaa. *Software Design Methods for Concurrent and Real-Time Systems*. Addison Wesley, 1993.

[Gom00] Hassan Gomaa. *Designing Concurrent, Distributed, and Real-Time Applications with UML*. Addison Wesley, 2000.

[GPB04] James Grenning, Johan Peeters, and Carsten Behring. Agile Development for Embedded Software. In *Proceedings of 4th Conference on Extreme Programming and Agile Methods (XP/Agile Universe)*, pages 194–195, Calgary, Canada, 2004.

[Gra91] Ian Graham. *Object-oriented Methods*. Addison Wesley, 1991.

[Gra95] Ian Graham. *Migrating to Object Technology*. Addison Wesley, 1995.

[Gro03] DoD Architecture Framework Working Group. DoD Architecture Framework Version 1.0 Deskbook, August 2003. https://acc.dau.mil/GetAttachment.aspx?id=31667&pname=file&aid=28906&lang=en-US (accessed: September 17, 2008).

[GS79] Chris Gane and Trish Sarson. *Structured System Analysis: Tools and Techniques*. Prentice Hall, 1979.

[GSCK04] Jack Greenfield, Keith Short, Steve Cook, and Stuart Kent. *Software Factories - Assembling Applications With Patterns, Models, Frameworks, and Tools*. John Wiley & Sons, 2004.

[GW05] Erich Gamma and John Wiegand. The Eclipse Way - Processes that Adapt. Presentation at EclipseCon 2005, March 2005. `http://www.eclipsecon.org/2005/presentations/econ2005-eclipse-way.pdf` (accessed: September 17, 2008).

[Hal62] A. D. Hall. *A Methodology for Systems Engineering*. Princeton, NJ: Van Nostrand, 1962.

[Har88] David Harel. On Visual Formalisms. *Communication of the ACM*, 31(5):514–530, May 1988.

[Hau05] Peter Haumer. *IBM Rational Method Composer - Part 1: Key Concepts*. The Rational Edge, December 2005. `www.ibm.com/developerworks/rational/library/dec05/haumer/index.html` (accessed: September 17, 2008).

[Hau06] Peter Haumer. *IBM Rational Method Composer - Part 2: Authoring method content and processes*. The Rational Edge, January 2006. `www.ibm.com/developerworks/rational/library/jan06/haumer/index.html` (accessed: September 17, 2008).

[Her07] Marcel Hermanns. Extending the ViPER Environment with Method-Based Support for MeDUSA, 2007. Diploma Thesis (German), RWTH Aachen University, `http://www.swc.rwth-aachen.de/lufgi/teaching/theses/completed/Marcel_Hermanns_Thesis_Report.pdf` (accessed: September 17, 2008).

[HG96] David Harel and Eran Gery. Executable Object Modeling with Statecharts. In *Proceedings 18th Conference on Software Engineering*, Berlin, March 1996.

[HLNW09] Veit Hoffmann, Horst Lichter, Alexander Nyßen, and Andreas Walter. Towards the Integration of UML- and Textual Use Case Modeling. In *Journal of Object Technology*, Mai-June 2009. to be published.

[Hof06] Hans-Peter Hoffmann. SysML-Based Systems Engineering Using a Model-Driven Development Approach. In *Proceedings of INCOSE 2006 International Symposium*, Orlanda, Florida, July 2006.

[HP88] Derek Hatley and Imtiaz Pirbhai. *Strategies for Real Time System Specification*. Dorset House, New York, 1988.

[HR02] Peter Hruschka and Chris Rupp. *Agile Softwareentwicklung für Embedded Real-Time Systems mit der UML*. Hanser Verlag, 2002.

[HS01] Brian Henderson-Sellers. An OPEN Process for Component-Based Development. In George T. Heineman and William T. Councill, editors, *Component-Based Software Engineering - Putting the Pieces Together*. Addison Wesley, 2001.

[HSE94] Brian Henderson-Sellers and Julian Edwards. *Book Two of Object-Oriented Knowledge: The Working Object*. Prentice Hall, 1994.

[Hum89] W. S. Humphrey. The Software Engineering Process: Definition and Scope. *ACM SIGSOFT Software Engineering Notes*, 14(4), June 1989.

[IEC98] International Electrotechnical Commission (IEC). *ISO/IEC 61508: Functional safety of electrical/electronic/programmable electronic safety-related systems, version 1.0*, 1998.

[IEC01] International Electrotechnical Commission (IEC). *ISO/IEC 9126: Software engineering - Product quality*, 2001.

[IEEE91] IEEE Std 610.12-1990: Standard Glossary of Software Engineering Terminology, 1991.

[IEEE04] Guide to the Software Engineering Body of Knowledge (2004 Version), 2004. `http://www.swebok.org/ironman/pdf/SWEBOK_Guide_2004.pdf` (accessed: September 17, 2008).

[Ikr08] Muhammad Tauseef Ikram. UML2 Interaction Modeling Capabilities for the ViPER Platform, 2008. Master Thesis, RWTH Aachen University, `http://www.swc.rwth-aachen.de/lufgi/teaching/theses/completed/Tauseef_Ikram_Thesis_Report.pdf` (accessed: September 17, 2008).

[ITU99] International Telecommunication Union. *ITU-T Recommendation Z.100 (11/99): Specification and Description Language (SDL)*, November 1999.

[ITU02] International Telecommunication Union. *ITU-T Recommendation X.680 (07/2002): Abstract Syntax Notation One (ASN.1), Specification of Basic Notation*, July 2002.

[ITU04] International Telecommunication Union. *ITU-T Recommendation Z.120 (04/2004): Message Sequence Chart (MSC)*, April 2004.

[J+07] M. Jersak et al. Timing model and methodology for AUTOSAR. *Elektronik Automotive*, October 2007. `http://www.elektroniknet.de/home/automotive/autosar/english/timing-model-and-methodology-for-autosar` (accessed: September 17, 2008).

[Jac75] Michael A. Jackson. *Principles of Program Design*. Academic Press, London, 1975.

[Jac83] Michael A. Jackson. *System Development*. Prentice Hall, 1983.

[Jac87] Ivar Jacobson. Object-oriented development in an industrial environment. In *OOPSLA '87: Conference proceedings on Object-oriented programming systems, languages and applications*, pages 183–191, New York, NY, USA, 1987. ACM Press.

[Jac96] Ivar Jacobson. Time for a cease-fire in the methods war. *Journal of Object-Oriented Programming, Guest Editorial*, 6(4), July/August 1996.

[Jac04] Ivar Jacobson. Use cases - Yesterday, today, and tomorrow. *Software and System Modeling*, 3(3):210–220, 2004.

[Jac08] Ivar Jacobson International. *Jaczone Waypointer 6.0 User Guide*, February 2008. bundled with the Jaczone Waypointer 6.0 evaluation version, obtainable from `http://www.ivarjacobson.com/products/waypointer.cfm` (accessed: September 17, 2008).

[Jan07] Matt Jane. Monitoring quality - Dealing with structural erosion in software development. *eStrategies Europe*, Mai 2007. `http://www.hello2morrow.de/download/estrategies.pdf` (accessed: September 17, 2008).

[JBR99] Ivar Jacobson, Grady Booch, and James Rumbaugh. *The Unified Software Development Process*. Addison Wesley, 1999.

[JCJv92] Ivar Jacobson, Magnus Christerson, Patrik Jonsson, and Gunnar Övergaard. *Object-Oriented Software Engineering - A Use Case Driven Approach*. Addison Wesley, ACM Press, 1992.

[KD06] Timo Käkölä and Juan Carlos Dueñas, editors. *Software Product Lines: Research Issues in Engineering and Management*. Springer Verlag, 2006.

[Kev07] Özgür Kevinç. Enhancing the ViPER Code-Generator by Concurrency and Timing, 2007. Diploma Thesis (German), RWTH Aachen University, `http://www.swc.rwth-aachen.de/lufgi/teaching/theses/completed/Oezguer_Kevinc_Thesis_Report.pdf` (accessed: September 17, 2008).

[Kop97] Hermann Kopetz. *Real-Time Systems - Design Principles for Distributed Embedded Applications*. Kluwer Academic Publishers, 1997.

[Leh08] Mark Lehmacher. Eine Simulationsumgebung für strukturierte natürlichsprachliche Anwendungsfallbeschreibungen, 2008. Diploma Thesis (German), RWTH Aachen University, `http://www.swc.rwth-aachen.de/lufgi/teaching/theses/completed/Mark_Lehmacher_Report.pdf` (accessed: September 17, 2008).

[LFM00] Howard Lykins, Sanford Friedenthal, and Abraham Meilich. Adapting UML for an Object-Oriented Systems Engineering Method (OOSEM).

In *Proceedings of INCOSE 2000 International Symposium*, Minneapolis, MN, USA, July 2000. http://www.omg.org/docs/syseng/01-09-05.pdf (accessed: September 17, 2008).

[LL07] Jochen Ludewig and Horst Lichter. *Software Engineering - Grundlagen, Menschen, Prozesse, Techniken*. dpunkt.verlag, 2007.

[Lun02] Jan Lunze. What is a Hybrid System ? In Sebastian Engell, Goran Frehse, and Eckehard Schnieder, editors, *Modelling, Analysis, and Design of Hybrid Systems*, LNCIS 279, pages 3–14. Springer Verlag, 2002.

[Mar94] Robert Martin. OO Design Quality Metrics - An Analysis of Dependencies, October 1994. http://www.objectmentor.com/resources/articles/oodmetrc.pdf (accessed: September 17, 2008).

[Mar03] Peter Marwedel. *Embedded System Design*. Kluwer Academic Press, 2003.

[Mar06] Robert C. Martin. *Agile Software Development: Principles, Patterns, and Practices in C#.* Prentice Hall, 2006.

[MB02] Stephen J. Mellor and Marc J. Balcer. *Executable UML - A Foundation for Model-Driven Archicture*. Addison Wesley, 2002.

[MDT] Eclipse Model Development Tools (MDT) project site. http://www.eclipse.org/mdt (accessed: September 17, 2008).

[MeDUSA] Method for UML2-based Construction of Embedded & Real-Time Software (MeDUSA) - project site. http://www.medusa.sc (accessed: September 17, 2008).

[MG92] Erik Mettala and Marc H. Graham. The Domain Specific Software Architecture Program. Technical report, Software Engineering Institute, June 1992. http://www.sei.cmu.edu/pub/documents/92.reports/pdf/sr09.92.pdf (accessed: September 17, 2008).

[Mik07] Tommi Mikkonen. *Programming Mobile Devices: An Introduction for Practitioners*. John Wiley & Sons, February 2007.

[MIS94] The Motor Industry Software Reliability Association (MISRA). *ISO/TR 15497: Development Guidelines for Vehicle Based Software*, November 1994.

[ML05] Jeff McAffer and Jean-Michel Lemieux. *Eclipse Rich Client Platform : Designing, Coding, and Packaging Java Applications*. Addison Wesley, Oktober 2005.

[MO92] James Martin and James J. Odell. *Object Oriented Analysis & Design*. Prentice Hall, 1992.

[MSZ01] Peter Müller, Christian Stich, and Christian Zeidler. Components @ Work: Component Technology for Embedded Systems. In *Proceedings of 27th International Workshop on Component-Based Software Engineering, EUROMICRO*, 2001.

[NL07a] Alexander Nyßen and Horst Lichter. MeDUSA - Method for UML2-based Design of Embedded Software Applications. Technical Report AIB-2007-07, RWTH Aachen University, May 2007. `http://aib.informatik.rwth-aachen.de/2007/2007-07.pdf` (accessed: September 17, 2008).

[NL07b] Alexander Nyßen and Horst Lichter. Use Case Modeling for Embedded Software Systems - Deficiencies & Workarounds. In M. Gehrke, H. Giese, and J. Stroop, editors, *Preproceedings of the 4th Workshop on Object-oriented Modeling of Embedded Real-Time Systems (OMER4)*, 30.-31. October 2007, HNF Musuems Forum, Paderborn, Germany, 2007.

[NL08] Alexander Nyßen and Horst Lichter. The MeDUSA Reference Manual, Second Edition. Technical Report AIB-2008-07, RWTH Aachen University, July 2008. `http://aib.informatik.rwth-aachen.de/2008/2008-07.pdf` (accessed: September 17, 2008).

[NLS+05] Alexander Nyßen, Horst Lichter, Jan Suchotzki, Peter Müller, and Andreas Stelter. UML2-basierte Architekturmodellierung kleiner eingebetteter Systeme - Erfahrungen einer Feldstudie. In Klein, Rumpe, and Schätz, editors, *Tagungsband des Dagstuhl-Workshops Modellbasierte Entwicklung eingebetteter Systeme (MBEES), Technischer Bericht, TU Braunschweig*, Dagstuhl, Germany, 2005. TUBS-SSE-2005-01.

[NMSL04] Alexander Nyßen, Peter Müller, Jan Suchotzki, and Horst Lichter. Erfahrungen bei der systematischen Entwicklung kleiner eingebetteter Systeme mit der COMET-Methode. In *Proceedings of Modellierung 2004, Lecture Notes in Informatics (LNI)*, volume P-45, pages 229–234, Marburg, Germany, March 2004.

[Oes01] Bernd Oestereich. *Developing Software with UML - Object-Oriented Analysis and Design in Practice (Second Edition)*. Addison Wesley, 2001.

[OMG95] The Common Object Request Broker: Architecture and Specification - Revision 2.0. OMG Formal Document 97-02-25, July 1995. `http://www.omg.org/docs/formal/97-02-25.pdf` (accessed: September 17, 2008).

[OMG97] UML Semantics v. 1.1. OMG document ad/97-08-04, 1997. `http://www.omg.org/docs/ad/97-08-04.pdf` (accessed: September 17, 2008).

[OMG01] Model Driven Architecture (MDA), July 2001. `http://www.omg.org/docs/ormsc/01-07-01.pdf`(accessed: September 17, 2008).

[OMG02] UML Profile for Schedulability, Performance, and Time Specification. OMG Final Adopted Specification ptc/02-03-02, March 2002. http://www.omg.org/docs/ptc/02-03-02.pdf (accessed: September 17, 2008).

[OMG03] MDA Guide version 1.0.1, June 2003. http://www.omg.org/docs/omg/03-06-01.pdf (accessed: September 17, 2008).

[OMG05a] Software Process Engineering Metamodel Specification, Version 1.1. OMG Formal Document formal/05-01-06, January 2005. http://www.omg.org/docs/formal/05-01-06.pdf (accessed: September 17, 2008).

[OMG05b] UML Profile for Schedulability, Performance, and Time v1.0. OMG Formal Document 05-01-02, September 2005. http://www.omg.org/docs/formal/05-01-02.pdf (accessed: September 17, 2008).

[OMG05c] UML Superstructure Specification, v2.0. OMG Formal Document 05-07-04, July 2005. http://www.omg.org/docs/formal/05-07-04.pdf (accessed: September 17, 2008).

[OMG06a] Common Object Request Broker Architecture - For Embedded. OMG Document 06-05-01, May 2006. http://www.omg.org/docs/ptc/06-05-01.pdf (accessed: September 17, 2008).

[OMG06b] Meta Object Facility (MOF) Core Specification. OMG Formal Document 06-01-01, January 2006. http://www.omg.org/docs/formal/06-01-01.pdf (accessed: September 17, 2008).

[OMG07a] A UML Profile for MARTE, Beta 1. OMG Document 07-08-04, August 2007. http://www.omg.org/docs/ptc/07-08-04.pdf (accessed: September 17, 2008).

[OMG07b] OMG Systems Modeling Language (OMG SysML) Specification. OMG Proposed Available Specification 07-02-04, February 2007. http://www.omg.org/docs/ptc/07-02-04.pdf (accessed: September 17, 2008).

[OMG07c] UML Infrastructure Specification, Version 2.1.2. OMG Formal Document 07-11-04, November 2007. http://www.omg.org/docs/formal/07-11-04.pdf (accessed: September 17, 2008).

[OMG07d] UML Superstructure Specification, Version 2.1.2. OMG Formal Document 07-11-02, November 2007. http://www.omg.org/docs/formal/07-11-02.pdf (accessed: September 17, 2008).

[OMG08] Software & Systems Process Engineering Meta-Model Specification, Version 2.0. OMG Document formal/08-04-01, April 2008. http://www.omg.org/docs/formal/08-04-01.pdf (accessed: September 17, 2008).

[Orr78] Kenneth T. Orr. *Structured Systems Development*. Prentice Hall, 1978.

[Par72] David Parnas. On the Criteria for Decomposing a System into Modules. *Communications of the ACM*, 15(12), December 1972.

[Par76] David Parnas. On the Design and Development of Program Families. *IEEE Transactions on Software Engineering*, 2(1), March 1976.

[RBP+91] James Rumbaugh, Michael J. Blaha, William Premerlani, Frederick Eddy, and William Lorenson. *Object-Oriented Modeling and Design*. Prentice Hall, 1991.

[RG92] Kenneth S. Rubin and Adele Goldberg. Object Behaviour Analysis. *Communications of the ACM*, 35(9):48–62, September 1992.

[Rit08] Philip Ritzkopf. Extending the ViPER IDE with methodical support for Use Case-based real-time analysis, 2008. Diploma Thesis, RWTH Aachen University, to be published.

[Rob92] Peter J. Robinson. *HOOD: Hierarchical Object Oriented Design*. Prentice Hall, 1992.

[Rom06] David Roman. By the Numbers - Software: What Gets Embedded. *Electronic Engineering Times*, April 2006. `http://i.cmpnet.com/eet/news/06/04/1418pg32_lay.pdf` (accessed: September 17, 2008).

[RS99] Doug Rosenberg and Kendall Scott. *Use Case Driven Object Modeling with UML - A Practical Approach*. Addison Wesley, 1999.

[RWL96] Trygve Reenskaug, Per Wold, and Odd Arild Lehne. *Working with Objects: The OOram Software Engineering Method*. Manning Publications, 1996.

[Sch04] Bernhard Schätz. Model-Based Development of Embedded Software Beyond UML. Position Paper at Workshop on Model-Based Development of Computer Based Systems: Appropriateness, Consistency and Integration of Models, 11th IEEE Symposium and Workshops on Engineering of Computer-Based Systems (ECBS'04) Brno, Czech Republic, May 2004.

[Sel03a] Bran Selic. Modeling Real-Time System Architectures. Presentation at the EmSys Summer School, Salzburg, Austria, June/July 2003. `http://www.softwareresearch.net/site/other/EmSys03/docs/09.Selic.pdf` (accessed: September 17, 2008).

[Sel03b] Bran Selic. *Models, Software Models and UML*. Kluwer Academic Publishers, 2003.

[Sel06] Bran Selic. Model-Driven Development: Its Essence and Opportunities. In *Proceddings of the Ninth IEEE International Symposium on Object and Component-Oriented Real-Time Distributed Computing (ISORC)*, pages 313–319, Gyeongju, Korea, 2006.

[SGW94] Bran Selic, Garth Gullekson, and Paul T. Ward. *Real-Time Object-Oriented Modeling*. John Wiley & Sons, 1994.

[Sim86] Hugo Simpson. The MASCOT Method. *IEE/BCS Software Engineering Journal*, 1(3):103–120, 1986.

[Sim08] Supaporn Simcharoen. Development of a TPTP-based GUI-Test Suite for the ViPER Integrated Development Environment, 2008. Master Thesis, RWTH Aachen University.

[SM88] Sally Shlaer and Stephen J. Mellor. *Object Oriented Systems Analysis*. Prentice Hall, 1988.

[SM92] Sally Shlaer and Stephen J. Mellor. *Object Lifecycles - Modeling the World in States*. Prentice Hall, 1992.

[SM97] Sally Shlaer and Stephen J. Mellor. Recursive Design of an Application-Independent Architecture. *IEEE Software*, 14(1), January 1997.

[SR98] Bran Selic and James Rumbaugh. Using UML for Modeling Complex Real-Time Systems, 1998. `http://www.ibm.com/developerworks/rational/library/content/03July/1000/1155/1155_umlmodeling.pdf` (accessed: September 17, 2008).

[Sta97] Jennifer Ann Margaret Stapleton. *DSDM: The Method in Practice*. Addison Wesley, 1997.

[Sub07] SAE AADL Subcomittee. AADL Annex - Behavior Language Compliance and Application Program Interface, March 2007. `http://la.sei.cmu.edu/aadl/documents/Behaviour_Annex1.6.pdf` (accessed: September 17, 2008).

[TEF+03] T. Thurner, J. Eisenmann, U. Freund, R. Geiger, M Haneberg, U Virnich, and S. Voget. Das EAST-EEA Projekt: Eine middlewarebasierte Softwarearchitektur für vernetzte Kfz-Steuergeräte. In *Proceedings of VDI-Kongress Elektronik im Kraftfahrzeug*, volume 1789 of *VDI-Berichte*, Baden-Baden, 2003.

[TGRT06] Yann Tanguy, Sébastien Gérard, Ansgar Radermacher, and François Terrier. Model Driven Engineering for Real Time Embedded Systems. In *Proceedings of the 3rd European Congress on Embedded Real Time Software (ERTS)*, Toulouse, France, January 2006.

[TK04] Maarit Tihinen and Pasi Kuvaja. Embedded Software Development - State of the Practice. Presentation at the MOOSE Seminar, Oulu, Finland, October 2004. `http://virtual.vtt.fi/moose/docs/oulu/embedded_sw_development_tihinen_kuvaja.pdf` (accessed: September 17, 2008).

[TPTP] Eclipse Test & Performance Tools Platform (TPTP) project site. `http://www.eclipse.org/tptp` (accessed: September 17, 2008).

[vdLSR07] Frank van der Linden, Klaus Schmid, and Eelco Rommes. *Software Product Lines in Action - The Best Industrial Practice in Product Line Engineering*. Springer Verlag, 2007.

[ViPERa] Visual Tooling Platform for Model-Based Engineering - bug-tracking site. `http://bugs.viper.sc` (accessed: September 17, 2008).

[ViPERb] Visual Tooling Platform for Model-Based Engineering - download site. `http://downloads.viper.sc` (accessed: September 17, 2008).

[ViPERc] Visual Tooling Platform for Model-Based Engineering - project site. `http://www.viper.sc` (accessed: September 17, 2008).

[vS02] Rini van Solingen. Integrating Software Engineering Technologies for Embedded Systems Development. In M. Oivo and S. Komi-Sirviö, editors, *Proceedings of PROFES 2002, LNCS 2559*, pages 466–474, Rovaniemi, Finland, 2002. Springer Verlag. `http://virtual.vtt.fi/virtual/proj1/projects/moose/docs/solingen_in_template_springer.pdf` (accessed: September 17, 2008).

[vS04] Rini van Solingen. State of the practice in European embedded software engineering. Keynote presentation at MOOSE Seminar, Helsinki, Finland, June 2004. `http://virtual.vtt.fi/moose/docs/seminar2004/state\%20of\%20the\%20practice\%20_\%20rini\%20van\%20solingen.pdf` (accessed: September 17, 2008).

[vvKM00] Rob van Ommering, Frank van der Linden, Jeff Kramer, and Jeff Magee. The Koala Component Model for Consumer Electronics Software. *IEEE Computer*, 33(3):78–85, March 2000.

[Wal07] Andreas Walter. A Use Case Modeling Tool for the ViPER-Platform, 2007. Diploma Thesis (German), RWTH Aachen University, `http://www.swc.rwth-aachen.de/lufgi/teaching/theses/completed/Andreas_Walter_Thesis_Report.pdf` (accessed: September 17, 2008).

[WBWW90] Rebecca Wirfs-Brock, Brian Wilkerson, and Lauren Wiener. *Designing Object-Oriented Software*. Prentice Hall, 1990.

[Weg87] Peter Wegner. Dimensions of object-based language design. In *OOPSLA '87: Conference proceedings on Object-oriented programming systems, languages and applications*, pages 168–182, New York, NY, USA, 1987. ACM Press.

[Wei06] Tim Weilkiens. *Systems Engineering mit SysML/UML*. dpunkt.verlag, 2006.

[Wir71] Niklaus Wirth. Program Development by Stepwise Refinement. *Communications of the ACM*, 14(4):221–227, 1971.

[WL99] David M. Weiss and Chi Tau Robert Lai. *Software Product Line Engineering: A Family Based Software Engineering Process*. Addison Wesley, 1999.

[WL06] Daniel Waddington and Patrick Lardieri. Model-Centric Software Development. In *IEEE Computer*, number 39 in 2, pages 28–30, February 2006.

[WM85] Paul T. Ward and Stephen J. Mellor. *Structured Development for Real-Time Systems*, volume 1, 2, and 3. Yourdon Press, New York, 1985.

[WNHL08] Andreas Walter, Alexander Nyßen, Veit Hoffmann, and Horst Lichter. Werkzeugunterstützung für textbasierte Use Case Modellierung. In K. Herrmann and B. Bruegge, editors, *Proceedings of Software Engineering 2008, GI Lecture Notes in Informatics (LNI)*, volume 121, 2008.

[Won93] William Wong. *Plug & Play Programming - An Object-Oriented Construction Kit*. M & T Books, New York, NY, 1993.

[YC79] Edward Yourdon and Larry L. Constantine. *Structured Design: Fundamentals of a Discipline of Computer Program and Systems Design*. Prentice Hall, 1979.

Aachener Informatik-Berichte

This list contains all technical reports published during the past five years. A complete list of reports dating back to 1987 is available from `http://aib.informatik.rwth-aachen.de/`**. To obtain copies consult the above URL or send your request to: Informatik-Bibliothek, RWTH Aachen, Ahornstr. 55, 52056 Aachen, Email:** `biblio@informatik.rwth-aachen.de`

2003-01 *	Jahresbericht 2002
2003-02	Jürgen Giesl, René Thiemann: Size-Change Termination for Term Rewriting
2003-03	Jürgen Giesl, Deepak Kapur: Deciding Inductive Validity of Equations
2003-04	Jürgen Giesl, René Thiemann, Peter Schneider-Kamp, Stephan Falke: Improving Dependency Pairs
2003-05	Christof Löding, Philipp Rohde: Solving the Sabotage Game is PSPACE-hard
2003-06	Franz Josef Och: Statistical Machine Translation: From Single-Word Models to Alignment Templates
2003-07	Horst Lichter, Thomas von der Maßen, Alexander Nyßen, Thomas Weiler: Vergleich von Ansätzen zur Feature Modellierung bei der Softwareproduktlinienentwicklung
2003-08	Jürgen Giesl, René Thiemann, Peter Schneider-Kamp, Stephan Falke: Mechanizing Dependency Pairs
2004-01 *	Fachgruppe Informatik: Jahresbericht 2003
2004-02	Benedikt Bollig, Martin Leucker: Message-Passing Automata are expressively equivalent to EMSO logic
2004-03	Delia Kesner, Femke van Raamsdonk, Joe Wells (eds.): HOR 2004 – 2nd International Workshop on Higher-Order Rewriting
2004-04	Slim Abdennadher, Christophe Ringeissen (eds.): RULE 04 – Fifth International Workshop on Rule-Based Programming
2004-05	Herbert Kuchen (ed.): WFLP 04 – 13th International Workshop on Functional and (Constraint) Logic Programming
2004-06	Sergio Antoy, Yoshihito Toyama (eds.): WRS 04 – 4th International Workshop on Reduction Strategies in Rewriting and Programming
2004-07	Michael Codish, Aart Middeldorp (eds.): WST 04 – 7th International Workshop on Termination
2004-08	Klaus Indermark, Thomas Noll: Algebraic Correctness Proofs for Compiling Recursive Function Definitions with Strictness Information
2004-09	Joachim Kneis, Daniel Mölle, Stefan Richter, Peter Rossmanith: Parameterized Power Domination Complexity
2004-10	Zinaida Benenson, Felix C. Gärtner, Dogan Kesdogan: Secure Multi-Party Computation with Security Modules
2005-01 *	Fachgruppe Informatik: Jahresbericht 2004
2005-02	Maximillian Dornseif, Felix C. Gärtner, Thorsten Holz, Martin Mink: An Offensive Approach to Teaching Information Security: "Aachen Summer School Applied IT Security"

2005-03 Jürgen Giesl, René Thiemann, Peter Schneider-Kamp: Proving and Disproving Termination of Higher-Order Functions

2005-04 Daniel Mölle, Stefan Richter, Peter Rossmanith: A Faster Algorithm for the Steiner Tree Problem

2005-05 Fabien Pouget, Thorsten Holz: A Pointillist Approach for Comparing Honeypots

2005-06 Simon Fischer, Berthold Vöcking: Adaptive Routing with Stale Information

2005-07 Felix C. Freiling, Thorsten Holz, Georg Wicherski: Botnet Tracking: Exploring a Root-Cause Methodology to Prevent Distributed Denial-of-Service Attacks

2005-08 Joachim Kneis, Peter Rossmanith: A New Satisfiability Algorithm With Applications To Max-Cut

2005-09 Klaus Kursawe, Felix C. Freiling: Byzantine Fault Tolerance on General Hybrid Adversary Structures

2005-10 Benedikt Bollig: Automata and Logics for Message Sequence Charts

2005-11 Simon Fischer, Berthold Vöcking: A Counterexample to the Fully Mixed Nash Equilibrium Conjecture

2005-12 Neeraj Mittal, Felix Freiling, S. Venkatesan, Lucia Draque Penso: Efficient Reductions for Wait-Free Termination Detection in Faulty Distributed Systems

2005-13 Carole Delporte-Gallet, Hugues Fauconnier, Felix C. Freiling: Revisiting Failure Detection and Consensus in Omission Failure Environments

2005-14 Felix C. Freiling, Sukumar Ghosh: Code Stabilization

2005-15 Uwe Naumann: The Complexity of Derivative Computation

2005-16 Uwe Naumann: Syntax-Directed Derivative Code (Part I: Tangent-Linear Code)

2005-17 Uwe Naumann: Syntax-directed Derivative Code (Part II: Intraprocedural Adjoint Code)

2005-18 Thomas von der Maßen, Klaus Müller, John MacGregor, Eva Geisberger, Jörg Dörr, Frank Houdek, Harbhajan Singh, Holger Wußmann, Hans-Veit Bacher, Barbara Paech: Einsatz von Features im Software-Entwicklungsprozess - Abschlußbericht des GI-Arbeitskreises “Features”

2005-19 Uwe Naumann, Andre Vehreschild: Tangent-Linear Code by Augmented LL-Parsers

2005-20 Felix C. Freiling, Martin Mink: Bericht über den Workshop zur Ausbildung im Bereich IT-Sicherheit Hochschulausbildung, berufliche Weiterbildung, Zertifizierung von Ausbildungsangeboten am 11. und 12. August 2005 in Köln organisiert von RWTH Aachen in Kooperation mit BITKOM, BSI, DLR und Gesellschaft fuer Informatik (GI) e.V.

2005-21 Thomas Noll, Stefan Rieger: Optimization of Straight-Line Code Revisited

2005-22 Felix Freiling, Maurice Herlihy, Lucia Draque Penso: Optimal Randomized Fair Exchange with Secret Shared Coins

2005-23 Heiner Ackermann, Alantha Newman, Heiko Röglin, Berthold Vöcking: Decision Making Based on Approximate and Smoothed Pareto Curves

2005-24 Alexander Becher, Zinaida Benenson, Maximillian Dornseif: Tampering with Motes: Real-World Physical Attacks on Wireless Sensor Networks

2006-01 * Fachgruppe Informatik: Jahresbericht 2005

2006-02 Michael Weber: Parallel Algorithms for Verification of Large Systems

2006-03 Michael Maier, Uwe Naumann: Intraprocedural Adjoint Code Generated by the Differentiation-Enabled NAGWare Fortran Compiler

2006-04 Ebadollah Varnik, Uwe Naumann, Andrew Lyons: Toward Low Static Memory Jacobian Accumulation

2006-05 Uwe Naumann, Jean Utke, Patrick Heimbach, Chris Hill, Derya Ozyurt, Carl Wunsch, Mike Fagan, Nathan Tallent, Michelle Strout: Adjoint Code by Source Transformation with OpenAD/F

2006-06 Joachim Kneis, Daniel Mölle, Stefan Richter, Peter Rossmanith: Divide-and-Color

2006-07 Thomas Colcombet, Christof Löding: Transforming structures by set interpretations

2006-08 Uwe Naumann, Yuxiao Hu: Optimal Vertex Elimination in Single-Expression-Use Graphs

2006-09 Tingting Han, Joost-Pieter Katoen: Counterexamples in Probabilistic Model Checking

2006-10 Mesut Günes, Alexander Zimmermann, Martin Wenig, Jan Ritzerfeld, Ulrich Meis: From Simulations to Testbeds - Architecture of the Hybrid MCG-Mesh Testbed

2006-11 Bastian Schlich, Michael Rohrbach, Michael Weber, Stefan Kowalewski: Model Checking Software for Microcontrollers

2006-12 Benedikt Bollig, Joost-Pieter Katoen, Carsten Kern, Martin Leucker: Replaying Play in and Play out: Synthesis of Design Models from Scenarios by Learning

2006-13 Wong Karianto, Christof Löding: Unranked Tree Automata with Sibling Equalities and Disequalities

2006-14 Danilo Beuche, Andreas Birk, Heinrich Dreier, Andreas Fleischmann, Heidi Galle, Gerald Heller, Dirk Janzen, Isabel John, Ramin Tavakoli Kolagari, Thomas von der Maßen, Andreas Wolfram: Report of the GI Work Group “Requirements Management Tools for Product Line Engineering”

2006-15 Sebastian Ullrich, Jakob T. Valvoda, Torsten Kuhlen: Utilizing optical sensors from mice for new input devices

2006-16 Rafael Ballagas, Jan Borchers: Selexels: a Conceptual Framework for Pointing Devices with Low Expressiveness

2006-17 Eric Lee, Henning Kiel, Jan Borchers: Scrolling Through Time: Improving Interfaces for Searching and Navigating Continuous Audio Timelines

2007-01 * Fachgruppe Informatik: Jahresbericht 2006

2007-02 Carsten Fuhs, Jürgen Giesl, Aart Middeldorp, Peter Schneider-Kamp, René Thiemann, and Harald Zankl: SAT Solving for Termination Analysis with Polynomial Interpretations

2007-03 Jürgen Giesl, René Thiemann, Stephan Swiderski, and Peter Schneider-Kamp: Proving Termination by Bounded Increase

2007-04 Jan Buchholz, Eric Lee, Jonathan Klein, and Jan Borchers: coJIVE: A System to Support Collaborative Jazz Improvisation

2007-05 Uwe Naumann: On Optimal DAG Reversal

2007-06 Joost-Pieter Katoen, Thomas Noll, and Stefan Rieger: Verifying Concurrent List-Manipulating Programs by LTL Model Checking

2007-07 Alexander Nyßen, Horst Lichter: MeDUSA - MethoD for UML2-based Design of Embedded Software Applications

2007-08 Falk Salewski and Stefan Kowalewski: Achieving Highly Reliable Embedded Software: An empirical evaluation of different approaches

2007-09 Tina Kraußer, Heiko Mantel, and Henning Sudbrock: A Probabilistic Justification of the Combining Calculus under the Uniform Scheduler Assumption

2007-10 Martin Neuhäußer, Joost-Pieter Katoen: Bisimulation and Logical Preservation for Continuous-Time Markov Decision Processes

2007-11 Klaus Wehrle (editor): 6. Fachgespräch Sensornetzwerke

2007-12 Uwe Naumann: An L-Attributed Grammar for Adjoint Code

2007-13 Uwe Naumann, Michael Maier, Jan Riehme, and Bruce Christianson: Second-Order Adjoints by Source Code Manipulation of Numerical Programs

2007-14 Jean Utke, Uwe Naumann, Mike Fagan, Nathan Tallent, Michelle Strout, Patrick Heimbach, Chris Hill, and Carl Wunsch: OpenAD/F: A Modular, Open-Source Tool for Automatic Differentiation of Fortran Codes

2007-15 Volker Stolz: Temporal assertions for sequential and concurrent programs

2007-16 Sadeq Ali Makram, Mesut Güneç, Martin Wenig, Alexander Zimmermann: Adaptive Channel Assignment to Support QoS and Load Balancing for Wireless Mesh Networks

2007-17 René Thiemann: The DP Framework for Proving Termination of Term Rewriting

2007-18 Uwe Naumann: Call Tree Reversal is NP-Complete

2007-19 Jan Riehme, Andrea Walther, Jörg Stiller, Uwe Naumann: Adjoints for Time-Dependent Optimal Control

2007-20 Joost-Pieter Katoen, Daniel Klink, Martin Leucker, and Verena Wolf: Three-Valued Abstraction for Probabilistic Systems

2007-21 Tingting Han, Joost-Pieter Katoen, and Alexandru Mereacre: Compositional Modeling and Minimization of Time-Inhomogeneous Markov Chains

2007-22 Heiner Ackermann, Paul W. Goldberg, Vahab S. Mirrokni, Heiko Röglin, and Berthold Vöcking: Uncoordinated Two-Sided Markets

2008-01 * Fachgruppe Informatik: Jahresbericht 2007

2008-02 Henrik Bohnenkamp, Marielle Stoelinga: Quantitative Testing

2008-03 Carsten Fuhs, Jürgen Giesl, Aart Middeldorp, Peter Schneider-Kamp, René :Thiemann, Harald Zankl: Maximal Termination

2008-04 Uwe Naumann, Jan Riehme: Sensitivity Analysis in Sisyphe with the AD-Enabled NAGWare Fortran Compiler

2008-05 Frank G. Radmacher: An Automata Theoretic Approach to the Theory of Rational Tree Relations

2008-06 Uwe Naumann, Laurent Hascoet, Chris Hill, Paul Hovland, Jan Riehme, Jean Utke: A Framework for Proving Correctness of Adjoint Message Passing Programs

2008-07 Alexander Nyßen, Horst Lichter:: The MeDUSA Reference Manual, Second Edition

2008-08 George B. Mertzios, Stavros D. Nikolopoulos: The λ-cluster Problem on Parameterized Interval Graphs

2008-09 George B. Mertzios, Walter Unger: An optimal algorithm for the k-fixed-endpoint path cover on proper interval graphs

2008-10 George B. Mertzios, Walter Unger: Preemptive Scheduling of Equal-Length Jobs in Polynomial Time

2008-11 George B. Mertzios: Fast Convergence of Routing Games with Splittable Flows

2008-12 Joost-Pieter Katoen, Daniel Klink, Martin Leucker, Verena Wolf: Abstraction for stochastic systems by Erlang's method of stages

2008-13 Beatriz Alarcón, Fabian Emmes, Carsten Fuhs, Jürgen Giesl, Raúl Gutiérrez, Salvador Lucas, Peter Schneider-Kamp, René Thiemann: Improving Context-Sensitive Dependency Pairs

2008-14 Bastian Schlich: Model Checking of Software for Microcontrollers

2008-15 Joachim Kneis, Alexander Langer, Peter Rossmanith: A New Algorithm for Finding Trees with Many Leaves

2008-16 Hendrik vom Lehn, Elias Weingärtner and Klaus Wehrle: Comparing recent network simulators: A performance evaluation study

2008-17 Peter Schneider-Kamp: Static Termination Analysis for Prolog using Term Rewriting and SAT Solving

2008-18 Falk Salewski: Empirical Evaluations of Safety-Critical Embedded Systems

* These reports are only available as a printed version.

Please contact `biblio@informatik.rwth-aachen.de` to obtain copies.

Curriculum Vitae

Alexander Nyßen
Born March 7th, 1978, in Heinsberg (Rhld.), Germany
Unmarried

Secondary Education & Military Service

08/1988 - 06/1997	Kreisgymnasium Heinsberg
11/1997 - 08/1998	Basic Military Service, Gerolstein

Studies & Student Employment

10/1998 - 12/2002	Studies in Computer Science RWTH Aachen University
05/2000 - 01/2003	Student Employee Datus AG, Aachen

Doctoral Studies & Employment

02/2003 - 01/2009	Doctoral Studies in Computer Science RWTH Aachen University
02/2003 - 12/2008	Scientific Staff Research Group Software Construction RWTH Aachen University
since 02/2009	Software Engineer and IT Consultant Itemis AG, Lünen