TECHNISCHE UNIVERSITÄT MÜNCHEN
Fakultät für Informatik
Lehrstuhl für Echtzeitsysteme und Robotik

Managing Non-Functional Communication Aspects in the Entire Life-Cycle of a Component-Based Robotic Software System

Alex Lotz

Vollständiger Abdruck der von der Fakultät für Informatik der Technischen Universität München zur Erlangung des akademischen Grades eines

Doktors der Naturwissenschaften (Dr. rer. nat.)

genehmigten Dissertation.

Vorsitzender:

Prof. Dr. **Uwe Baumgarten**

Prüfer der Dissertation:

1. Prof. Dr.-Ing. habil. **Alois Knoll**
2. Prof. Dr. **Christian Schlegel**
3. Prof. Dr.-Ing. **Rüdiger Dillmann**

Die Dissertation wurde am 10.08.2017 bei der Technischen Universität München eingereicht und durch die Fakultät für Informatik am 21.12.2017 angenommen.

This thesis has been conducted in a
cooperative doctoral program („kooperative Promotion“)

with the **Technische Universität München**,
Department of **Informatics**,
Chair of **Robotics and Embedded Systems**

and the **Ulm University of Applied Sciences**,
Service Robotics Research Center.

Hochschule Ulm
University of
Applied Sciences

servicerobotik
Service Robotics
autonomous mobile service robots

Abstract

The development of software systems for autonomous, mobile service robots is becoming increasingly complex. One reason for this is the massive increase in the number of available algorithms and basic functionalities that are continuously introduced and extended by the robotic community. However, the growing number of algorithms presents more challenges when integrating them into consistent systems in such a way that these algorithms can be coordinately executed in combination, without being conflicting within their individual execution contexts and action spaces.

Moreover, there has been a recent surge in interest in robotic technologies in domains such as the factory of the future, agriculture, and healthcare. However, these domains call for a much higher maturity level in robotic technologies than what is currently provided by the numerous lab prototypes. In particular, non-functional, application-specific system aspects, such as end-to-end guarantees for sensor-to-actuator control loops, are important system properties that must be managed from the very beginning and throughout the entire lifecycle of a robotic software system.

Current approaches in robotics mainly focus on coping with software complexity. These approaches typically introduce a type of robotic component model. Some prominent examples are the *BRICS Component Model (BCM)* [Bru+13], the *Robot Modeling Language (RobotML)* [Dho+12], and the *Robot-Technology Component (RTC)* [And+05]. While these approaches help to deal with parts of the overall software complexity, they neglect the necessity of managing non-functional system aspects.

Other approaches, derived from fields other than robotics, such as *AADL* [AAD04] from avionics, or *MARTE* [MAR11] from embedded-systems, already deal with non-functional aspects. However, these approaches are not yet accessible to the domain of robotics due to a general lack of support for component-based development, which is important in robotics. This dissertation therefore develops a flexible methodology that combines approaches and tools from other domains with established component-based modeling tools used within the robotic domain. The overall goal is to reuse available solutions in the best way possible and to tailor the approaches to the specific demands of robotics (such as the natural need for mixed criticality) wherever needed.

On the whole, this dissertation develops an integrated modeling approach that simultaneously addresses two common robotic software engineering needs. The first need is related to complexity management that is addressed by a *Component-Based Software Development (CBSD)* methodology. The second need is related to the management of non-functional properties as an integral part of the CBSD methodology. This overall methodology is implemented in an Eclipse-based modeling toolchain. Additionally, a real-world example (introduced and discussed in this dissertation) is used for the evaluation of the modeling toolchain. Consequently, this dissertation shows that external analysis tools (i.e., those from other domains) can be made accessible and usable to the domain of robotics, and that the combination of modeling and analysis tools enables the management of non-functional aspects in robotic applications and scenarios.

Zusammenfassung

Die Entwicklung von Software-Systemen für autonome, mobile Service-Roboter wird zunehmend komplexer. Ein Grund hierfür ist die steigende Anzahl von verfügbaren Algorithmen und Funktionalitäten, die fortwährend von der Robotik-Community ausgebaut werden. Jedoch wird es – mit der immer weiter ansteigenden Anzahl von Algorithmen – zunehmend schwieriger, diese in konsistente Gesamtsysteme zu integrieren, sodass diese Algorithmen zusammen und koordiniert ausgeführt werden, ohne dabei ihre jeweiligen Entscheidungsräume zu verletzen.

Des Weiteren steigt das Gesamtinteresse für Robotiktechnologien in Domänen, wie z. B. Industrie 4.0, Landwirtschaft und Medizin. Das Problem ist, dass diese Domänen einen viel höheren Reifegrad für Robotiktechnologien benötigen als das, was heutzutage in den vielen Laborprototypen gezeigt wird. Insbesondere nicht-funktionale, applikationsspezifische Systemeigenschaften, wie z. B. Ende-zu-Ende-Garantien für Sensor-Aktuator-Koppelungen, sind bedeutsam und müssen durchgängig im gesamten Entwicklungsprozess berücksichtigt und verwaltet werden.

Aktuelle Ansätze in der Robotikdomäne konzentrieren sich hauptsächlich auf Komplexitätsbeherrschung. Diese Ansätze nutzen typischerweise eine Art Komponentenmodell für Robotik. Prominente Beispiele sind das *BRICS Component Model (BCM)* [Bru+13], die *Robot Modeling Language (RobotML)* [Dho+12] und die *Robot-Technology Component (RTC)* [And+05]. Obwohl diese Ansätze bereits die Komplexitätsbeherrschung verbessern, so ignorieren sie weiterhin das Problem der nicht-funktionalen Systemeigenschaften.

Andere Ansätze außerhalb der Robotikdomäne, wie z. B. *AADL* [AAD04] aus der Luftfahrt oder *MARTE* [MAR11] für eingebettete Systeme, erlauben bereits das Modellieren und Verwalten nicht-funktionaler Eigenschaften. Jedoch sind diese Ansätze noch nicht für die Robotikdomäne zugänglich, da diese die Komponentenbauweise, so wie sie in der Robotik benötigt wird, nicht unterstützen. Deswegen wird in dieser Dissertation eine flexible Methodik entwickelt, die es erlaubt, die Ansätze und Werkzeuge aus den anderen Domänen in der Robotikdomäne zugänglich zu machen. Ein Gesamtziel ist es, vorhandene Ansätze und Werkzeuge so weit wie möglich wiederzuverwenden und diese – wo immer nötig – an die Besonderheiten der Robotik (wie z. B. die natürliche Notwendigkeit für unterschiedlich kritische Laufzeitanforderungen) anzupassen.

Insgesamt wird in dieser Dissertation ein integrierter Modellierungsansatz entwickelt, der zwei gängige Softwareentwicklungsprobleme der Robotik in Kombination adressiert. Das erste Problem betrifft die Komplexitätsbeherrschung, was durch einen komponentenbasierten Ansatz gelöst wird. Das zweite Problem bezieht sich auf das Explizieren und Verwalten nicht-funktionaler Eigenschaften als einen integralen Bestandteil von dem obigen komponentenbasierten Ansatz. Die entwickelte Methodik wird in einer Eclipse-basierten Modellierungstoolchain implementiert. Zusätzlich wird ein Realwelt-Beispiel vorgestellt, das unter anderem für die Evaluierung der vorgestellten Modellierungswerkzeuge genutzt wird. Das Ergebnis dieser Dissertation zeigt, dass externe Analysewerkzeuge aus anderen Domänen für Robotik zugänglich und nutzbar gemacht werden können und dass die Kombination aus Modellierungswerkzeugen und angebundenen Analysewerkzeugen das Verwalten und Analysieren von nicht-funktionalen Eigenschaften in den Robotik Anwendungen und Szenarien ermöglichen.

It is my conviction that robotic technologies – if applied carefully and with great responsibility – have the potential to alleviate many nowadays problems of human society on both the local and the global scale, let it be robots autonomously exploring the space or robots serving and helping people by doing laborious or dangerous works. Moreover, robotics science started a new technological revolution that opens up tremendous opportunities for industries and for private sectors but also comes with huge societal challenges that need to be addressed not only in science but in particular also in the broad public in order to reduce the risk of harming humanity and to shape a better future.

This work is dedicated to my family and to my best friends around the world.

Acknowledgements

I would like to thank my family (my mother and my brother) for supporting me in my undertaking of writing this dissertation. In particular at times full of frustration that are unavoidable in such an endeavor, I always received great support and encouragement to keep going.

Next, I would like to thank Prof. Dr.-Ing. habil. Alois Knoll for giving me the chance to do this PhD and for providing me helpful feedback for my initial ideas.

Furthermore, I am very grateful to all the members of the Bosch collaboration. In particular, Dr. Arne Hamann greatly supported me in coming up with sound structures and industry-oriented argumentation. While some discussions in face-to-face meetings have been intense at times, most notably together with Dr.-Ing. Ingo Lütkebohle and Dr. Christian Heinzemann, it were precisely these discussions that helped me the most in refining and improving the overall ideas in this dissertation. Noteworthy, these discussions also resulted in two joint, peer-reviewed publications that form the basis for this dissertation. I would like to thank Vincent Kesel for the first prototypical implementations of the mapping towards SymTA/S.

Last, but most certainly not least, I would like to thank Matthias Lutz, Dennis Stampfer and Prof. Dr. Christian Schlegel. Dennis and Matthias are my valued and respected colleagues who have been my scientific companions since the early graduate courses. They both helped me to become the researcher that I am today. At times they challenged my assumptions and conclusions, and at other times they backed me up in difficult meetings. I am most grateful for their sheer infinite patience in answering uncountable questions and in spending uncountable hours of most fruitful discussions, regardless of the high pressure of their daily project work. While my colleagues helped me to become a researcher, Prof. Dr. Christian Schlegel basically shaped me (both personally and professionally) and led out the foundation for my scientific career. He has been (and still is) my mentor, my supervisor and my guide in my scientific life. He is somebody you can entirely rely on in life, both personally and professionally. Not only has he shaped the overall vision of our research team over the years, wrote uncountable research proposals, fought ideological wars in highly political discussions with other research groups and politicians, but he nonetheless still was able to find times for discussions with me and my colleagues. These discussions helped me at times to sort out and to clean up my ideas, and at other times to bring me back on track by setting the priorities right and by patiently explaining the overall vision. His clever balance between guidance and freedom provided me a lot of room for personal development while at the same time always giving a general aim. Overall, my entire research career has been highly influenced by these three persons, to whom I cannot be thankful enough.

Contents

Abstract v

Contents xi

Glossary xv

I. Introduction and Fundamentals 1

1. Introduction 3
   - 1.1. Motivation . . . 4
   - 1.2. Research Question and Problem Statements . . . 6
   - 1.3. Dissertation Objectives . . . 7
   - 1.4. Core Contributions . . . 8
   - 1.5. Dissertation Outline . . . 10
   - Core Publications . . . 12

2. Related Works and SmartSoft Fundamentals 15
   - 2.1. Related Works . . . 15
     - 2.1.1. Robot programming and robotic frameworks . . . 15
     - 2.1.2. Model-driven engineering (MDE) and domain-specific languages (DSLs) in robotics . . . 17
     - 2.1.3. MDSE in other domains close to robotics and MDSE community involvements . . . 19
     - 2.1.4. Models of computation (MoC), non-functional system properties, and quality of service . . . 21
     - 2.1.5. List of own publications . . . 23
     - 2.1.6. Summary . . . 24
   - 2.2. The SmartSoft Fundamentals . . . 25
     - 2.2.1. The SmartSoft communication patterns . . . 26
     - 2.2.2. Toward QoS and variability management in a robotic development process 27
     - 2.2.3. Platform-independent component model with different middleware mappings: from shared memory to DDS . . . 27
     - 2.2.4. The SmartMDSD Toolchain . . . 29

II. The Method 31

3. Toward Structures Supporting Domain-Specific Robotic S/W Development 33
- 3.1. Common Challenges in Robotic Software Development . . . 34
  - 3.1.1. Problem complexity in robotic application domains . . . 36
  - 3.1.2. Design complexity in robotic software development . . . 37
  - 3.1.3. Bounded rationality and partitioned expertise . . . 38
  - 3.1.4. Non-functional, application-specific needs in robotic software development 38
- 3.2. Use Case-driven State-of-the-Art Analysis . . . 39
  - 3.2.1. The navigation use case . . . 39
  - 3.2.2. Best practices in robotic software and system engineering . . . 42
  - 3.2.3. Vertical (layered) and horizontal (workflow) decomposition . . . 43
  - 3.2.4. Relationship between components' internal processing and inter-component communication . . . 47
  - 3.2.5. Trading-off QoS requirements . . . 50
- 3.3. The Overall MDSE-Based Approach and Method Overview . . . 51
  - 3.3.1. The central role of MDSE in this dissertation . . . 52
  - 3.3.2. The main MDSE tools used in this dissertation . . . 53
  - 3.3.3. Summary and overview of the meta-model packages . . . 55

4. Extended SmartSoft Component Model 57
- 4.1. The Component Developer View . . . 58
  - 4.1.1. Toward middleware-independent service definitions . . . 59
  - 4.1.2. Component's internal models of computation . . . 60
  - 4.1.3. Component's orchestration interface and lifecycle state automaton . . . 62
- 4.2. Extended SmartSoft Component Meta-Model and Flexible Task Implementation . 64
  - 4.2.1. Simplified component model with configurable services . . . 65
  - 4.2.2. Component-performance extension . . . 69
  - 4.2.3. Flexible TaskTrigger implementation and decoupled component's internal service interface . . . 72
- 4.3. Related Works and Conclusion . . . 74
  - 4.3.1. Component models . . . 74
  - 4.3.2. Robotic frameworks and communication middlewares . . . 75
  - 4.3.3. Conclusion . . . 76

5. Models in the System Integration Phase 77
- 5.1. The System-Integrator View . . . 78
  - 5.1.1. Variability management and stepwise system refinement . . . 79
  - 5.1.2. Designing reactivity and responsiveness of a robotic software system . . 80
  - 5.1.3. Cause–effect (task) chains . . . 81

5.2. The System Integration Meta-Models . . . 82
5.2.1. Simplified system-configuration meta-model . . . 83
5.2.2. Deployment meta-model . . . 89
5.2.3. Further possible meta-models . . . 92
5.3. The Performance Specification View . . . 93
5.3.1. The definition of task nets . . . 96
5.3.2. The definition of cause–effect task chains . . . 103
5.3.3. Further performance-analysis related elements . . . 106
5.4. Related Works and Conclusion . . . 106
5.4.1. System integration approaches within the domain of service robotics . . . 107
5.4.2. System integration approaches outside and beyond service robotics . . . 107
5.4.3. Conclusion . . . 108

6. SymTA/S Performance Analysis and Other Runtime Aspects 109
6.1. Integrating SymTA/S into the Overall Robotic Development Workflow . . . 110
6.1.1. SymtaBase meta-model and the model-to-model transformation . . . 111
6.1.2. SymtaConfig meta-model . . . 121
6.1.3. SymTA/S Python script generation and SymTA/S project configuration . 124
6.2. Logging End-to-End Delays in a Real System . . . 126
6.2.1. Designing the logging infrastructure . . . 126
6.2.2. Logging core instrumentation . . . 127
6.3. Monitoring System Properties at Runtime . . . 131
6.3.1. Runtime variability exploitation for maintaining a service quality . . . 131
6.3.2. Best-effort and hard real-time execution . . . 132
6.4. Related Works and Conclusion . . . 133
6.4.1. Performance analysis approaches . . . 133
6.4.2. Monitoring and runtime adaptation . . . 134
6.4.3. Conclusion . . . 135

III. Results, Future Works and Conclusions 137

7. Demonstration Examples and Performance Analysis Results 139
7.1. Navigation Component- and System-Models . . . 139
7.1.1. Navigation component-model-diagrams . . . 140
7.1.2. Navigation model-diagrams in the system-integration phase . . . 147
7.2. SymTA/S Performance Analysis and its Results . . . 154
7.2.1. Case-study context . . . 155
7.2.2. Setting the Hypothesis . . . 157
7.2.3. Preparing the case-study . . . 158
7.2.4. Validating the Hypothesis . . . 159

7.2.5. Analysing the results 165
7.3. Summary and Discussion of the Scenario Results 166

8. Borderline Topics and Future Works **169**
8.1. Recent Modeling Refinements 169
8.2. Conceptual Mapping Toward AADL 171
8.3. Toward a Combination of the Request-Response - and the Data-Flow Communication Semantics 175

9. Summary and Conclusions **177**
9.1. Summary of the Core Contributions 177
9.2. Discussion of the Achieved Results 179
9.3. Conclusions 181

A. Appendix **183**
A.1. Communication Objects Xtext Grammar 183
A.2. Component Xtext Grammar 186
A.3. System Xtext Grammar 188
A.4. Deployment Xtext Grammar 189
A.5. Performance Xtext Grammar and Two Completion Proposal Providers 191
A.6. SymtaBase Xtext Grammar 195
A.7. SymtaConfig Xtext Grammar 197
A.8. Matlab Histogram Plots for the Execution- and End-to-End Times 198

Bibliography **203**

Glossary

SmartSoft is an umbrella term for a component-based robotic framework (see [SW99]) and a robotic component-model based on the SMARTSOFT communication patterns. 16, 18, 25–28, 52, 54, 57, 60, 64, 72, 75, 76, 90, 127, 128, 140, 177, *see* Communication Pattern & framework

AADL Architecture Analysis & Design Language: `http://www.aadl.info`. 19, 25, 75, 107, 108, 110, 134, 167, 171–173, 175, 179, 181

ACE (Adaptive Communication Environment) is a message-oriented middleware solution. ACE implements a rich set of design patters for concurrent communication software and is freely available as open-source under: `http://www.cs.wustl.edu/~schmidt/ACE.html`. 28, 54, 72, 90, 127, 128, 140, *see* middleware

API Application Programming Interface. 59, 67, 72, 111, 117, 122, 124, 125, 169

ASCII American Standard Code for Information Interchange. 111

AST Abstract Syntax Tree (AST) is the in-memory representation of a model based on a predefined Ecore meta-model. 54, 140, *see* Ecore

BCM The BRICS Component Model (BCM) [Bru+13] mainly driven by KU Leuven. 16, 18, 25, 43, 52, 75, 107

BCRT Best-Case Response-Time. 158, 162

CBSE Component-Based Software Engineering. 18, 24, 25, 42, 43, 177, 179

CCM OMG's CORBA Component Model. 74, 75

CDL Curvature Distance Lookup (CDL) [Sch98] is a fast local obstacle avoidance algorithm for mobile robots that considers kinematic and dynamic constraints. 41

Communication Object defines data structure for communication between components (see [Lut+14]) in a service. 67, 73, 179, *see* service

Communication Pattern refers to a fix set of software patterns defining recurring communication solutions (see [SW99]) for robotic software components. 26–28, 60, 179

CORBA OMG's Common Object Request Broker Architecture. 28, 76, 90

CPA Compositional Performance Analysis. 10, 94, 98, 109, 135, 139, 153–155, 158, 159, 166, 167, 178, 179

CPU Central Processing Unit. 46, 106, 111, 119, 121, 122

DDS OMG's Data Distribution Service. 28, 67, 76

DRE Distributed, Real-time and Embedded. 5, 75, 110

DSL Domain-Specific Language. 9, 17, 19, 23, 25, 27, 52, 53

DSPL Dynamic Software Product Line. 17, 18

Ecore is the main meta-model definition language of the EMF [Ste+11, Ch. 2]. 65, 66, 70, *see* EMF

EMF Eclipse Modeling Framework provides the core meta-model facilities in Eclipse such as Ecore [Ste+11]. 53, *see* Ecore

EMOF Essential Meta-Object Facility. 53

EMT A collection of Eclipse modeling tools around EMF. 53, *see* EMF

FIFO First-In First-Out. 71, 128

Framework abstracts away platform-specific details such as independence of a particular operating system and communication middleware by providing a unified and platform independent API. 15, 16, 64, 72, 75, 76, 126–128, 140, 178, *see also* middleware & API

GPL General-Purpose Language. 53

GPML General-Purpose (Modeling) Language. 53, 107, 171

GUI Graphical User Interface. 131, 134

HRI Human-Robot Interaction. 34

IDE Integrated Development Environment. 9, 180

IDL Interface Definition Language. 67

KPN Kahn Process Network. 21, 98

LET Logical Execution Time. 21

MAR Multi-Annual Roadmap. 3, 34

MARTE Modeling And Analysis Of Real-Time Embedded Systems. 19, 21, 75, 107, 108

MDA Model-Driven Architecture: `http://www.omg.org/mda/`. 19

MDSE Model-Driven Software Engineering. 8, 24–26, 34, 43, 52, 53, 72, 177, 178, 180

Middleware abstracts away communication-specific platform details and implements particular (often standardized) communication technologies (such as e.g. OMG's CORBA, or OMG's DDS, etc.). 15, 59, 60, 67, 75, 76, *see* OMG, CORBA & DDS

MoC Model of Computation (MoC) defines a scheme how data is propagated in a component based system. 21, *see also* SDF

MOF Meta-Object Facility. 19, 53

OCL Object Constraint Language. 53, 85, 87

OMG Object Management Group: `http://www.omg.org/`. 16, 19, 21, 76, 107, 108

OS Operating System. 106, 122, 124

QoS (Quality of Service) defines the ability of a system to meet application-specific customer needs and expectations while remaining economically competitive. 22, 27, 39, 181

RBS Reservation-Based Scheduling. 22, 134

RobotML The Robot Modeling Language (RobotML) [Dho+12] originating from the French Proteus project. 16, 18, 25, 52, 75

ROS Robot Operating System (ROS) [Qui+09] is a popular communication middleware and algorithm repository. 15, 16, 39, 75, 76, 107, *see also* middleware

RTC Robot Technology Component (RTC) [And+05] is an OMG standard for a component model, driven by Japan Robot Association. 16, 18, 25, 68, 75, *see* OMG

SDF Synchronous Data Flow (SDF) [LM87] is a widely known model of computation that specifies a synchronous communication scheme. 21, 61, 97, 98, *see* MoC

Service A service defines a communication contract between two interacting components, where one component provides (i.e. implements) the service and the other component requires (i.e. uses) it. A service is specified by (1) a name, (2) a communication object and (3) a communication pattern (see [Lut+14]). 27, 45, 59–64, 67, 179, *see* Communication Object & Communication Pattern

SOA Service-Oriented Architecture. 18, 24, 26, 43, 179

SPL Software Product Line. 17, 18

SymTA/S The SymTA/S approach [Hen+05] allows a symbolic timing analysis (including a scheduling and a response-time analyses) optimized for embedded and automotive systems. xx, 10, 12, 22, 54, 55, 110, 111, 116, 117, 119–122, 124–127, 129, 132, 134, 135, 139, 154, 155, 157–159, 161, 164, 166, 175, 179, 181

SymTA/S & Trace Analyser is an Eclipse based workbench developed by Symtavision that implements the SymTA/S approach. 10, 109–111, 124, 125, 132, 134, 135, 155, 158, 159, 166, 167, 175, 178, 180, *see* SymTA/S

SysML System Modeling Language. 19, 75

TDL Timing Definition Language. 21

TRL Technology Readiness Level. 34

UCM Unified Component Model. 19, 20, 75

UML Unified Modeling Language: `http://www.uml.org/`. 19, 53, 72–74, 89

WCET Worst-Case Execution Time. 22, 99, 133, 165

WCRT Worst-Case Response-Time. 158, 162, 165

XML Extensible Markup Language. 124

List of Figures

1.1. Robot execution cycle . . . 4
1.2. A robot's capabilities . . . 5
1.3. Schematic illustration of the contributions of this dissertation . . . 9
1.4. Dissertation Outline . . . 11

2.1. Balancing two common engineering needs in robotic software development . . . 24
2.2. Middleware mappings . . . 28

3.1. The step-change from lab prototypes toward high-quality products . . . 33
3.2. Challenges of overall software development meet application-related needs . . . 35
3.3. Three robotic domain clusters . . . 35
3.4. Software complexity sources . . . 36
3.5. Motivating navigation scenario . . . 40
3.6. Required tools to support interfacing and handover of knowledge and artefacts between distinct development-workflow steps . . . 44
3.7. Vertical decomposition by defining appropriate abstraction layers . . . 46
3.8. CDL component schematic example . . . 47
3.9. Base-Laser-CDL-Base component chain . . . 49
3.10. Common aspects of MDSE (terminology is borrowed from [Voe13]) . . . 52
3.11. The main Eclipse modeling tools used in this dissertation . . . 54
3.12. Ecore packages overview . . . 55

4.1. Component concept . . . 57
4.2. Component container . . . 59
4.3. Models of computation . . . 61
4.4. Generic Lifecycle of a Component [SLS11a] . . . 63
4.5. The two Ecore-based component meta-model packages . . . 64
4.6. Component Ecore meta-model . . . 66
4.7. Performance Extension meta-model extends the component meta-model through derivation . . . 69
4.8. Component Performance Extension Ecore meta-model . . . 69
4.9. Generic Task-Trigger UML Class Diagram . . . 73

5.1. Conceptual overview of the three core system integration views . . . 77
5.2. The three system-integration views (realized as Ecore meta-model packages) . . . 82
5.3. The system-configuration view . . . 83
5.4. System-configuration Ecore meta-model . . . 84
5.5. The deployment view . . . 89
5.6. Deployment Ecore meta-model . . . 90

5.7. The system-performance view . . . 93
5.8. Performance Ecore meta-model . . . 95
5.9. From component models to task net models . . . 96
5.10. Relation between Activation-Constraints, Activation-Source and End2End latencies 105

6.1. SymTA/S artefacts . . . 110
6.2. SymTA/S Base Ecore meta-model . . . 112
6.3. SymtaConfig package . . . 121
6.4. SymTA/S Configuration Ecore meta-model . . . 122
6.5. Performance => SymtaConfig model-to-text transformation . . . 123
6.6. PythonScript generation step . . . 124
6.7. SymTA/S Project Explorer . . . 125
6.8. Class-diagram for the two generic logging classes and a UserTask class . . . 128
6.9. Sequence-diagram showing the logging of task cycle's start/end times . . . 130

7.1. Component diagrams for the SmartPioneerBaseServer (on the left) and the SmartLaserLMS200Server (on the right) . . . 141
7.2. Component diagrams for SmartJoystickServer (left) and SmartJoystickNavigation (right) . . . 143
7.3. Component diagrams for SmartMapperGridMap (left) and SmartPlannerBreadthFirstSearch (right) . . . 144
7.4. Component diagram for SmartCdlServer . . . 146
7.5. System-Configuration Diagram . . . 148
7.6. System-Deployment Diagram . . . 149
7.7. System-Performance Diagram (see also [Lot+16]) . . . 150
7.8. System-Performance Diagram (Deployment-Reference view) . . . 151
7.9. Pioneer P3DX moving in a home-like environment . . . 156
7.10. Histogram plots of the logged execution times for the CDL task with two different configurations . . . 160
7.11. Histogram plots of the simulated execution times (using Symbolic Timing Analysis for Systems (SymTA/S)) for the CDL task with two different configurations . 161
7.12. Histogram plots of the overall end-to-end data-flow times (based on logged raw data that has been aggregated using a MATLAB script) for the FastReactiveNavigation loop with two different configurations of the CDL task . . . 163
7.13. Histogram plots of the simulated overall end-to-end data-flow times (as result from the SymTA/S performance analysis) for the FastReactiveNavigation loop with two different configurations of the CDL task . . . 164

8.1. Refined component meta-model including upcall handlers and task/upcall observer pattern . . . 170
8.2. Component diagram for SmartPioneerBaseServer . . . 170

8.3. SmartPioneerBaseServer (modeled with the AADL OSATE2 tool) 172
8.4. Navigation-scenario system-configuration view (modeled with AADL OSATE2) 173
8.5. Navigation-scenario extending tasks (top) and defining data-flows (bottom) in the system-performance view (modeled with the AADL OSATE2 tool) 174
8.6. Query communication - schematic example 176

A.1. Histogram plots of the logged execution-times for the BaseOdometry task (the two plots on top) and the BaseVelocityCommand task (bottom plots) 199
A.2. Histogram plots of the logged execution-times for the JoystickServer task (the two plots on top) and the JoystickNavigation task (bottom plots) 200
A.3. Histogram plots of the logged execution-times for the Laser task (the two plots on top) and the Mapper task (bottom plots) 201
A.4. Histogram plots of the logged execution-times for the Planner task 202

List of Tables

3.1. Characterization of different timing requirements for typical capabilities of a robot 50

4.1. Mappings from component meta-model elements toward corresponding classes in the ACE/SMARTSOFT framework 72

5.1. Mapping from component and system models to the task net elements 98
5.2. Mapping from ActivationConstraints to ActivationSources 100

6.1. Mapping from the *performance* meta-model to the *SymtaBase* meta-model 113

7.1. Graphical notation of component's meta-model elements 141
7.2. Overview of Activation-Constraints for all Tasks of the presented components . . 142
7.3. Over-/under-sampling configuration of the different InputLinkExtensions 145
7.4. Graphical notation of the performance meta-model elements 149
7.5. Relation between the InputLink and the InputNode refinements 152
7.6. Relation between the availability of *InputNodes* of a *TaskNode* and the permissible *ActivationSource* types 153

8.1. Mapping between Component and AADL model elements 172

Part I.

Introduction and Fundamentals

1 Introduction

The overall objective of this dissertation is to identify required structures and abstractions that help to systematically design and develop complex robotic software systems. Specifically, this dissertation considers autonomous, mobile service robots that operate in open-ended, everyday-like environments. Service robotics is an interdisciplinary research field that combines and integrates achievements from various other domains such as embedded and real-time systems, electronics, mechanics, and software engineering. Recent movies featuring intelligent robots have excited high expectations in the public mind for future robotic technologies in application domains like elderly care, logistics, agriculture, search and rescue, and entertainment, as well as for robots as industrial general-purpose co-workers and household helpers. Moreover, robotic research fosters these expectations by showcasing impressive lab prototypes with isolated capabilities. Yet, only a few rather simple examples have been realized, such as the home-cleaning devices from iRobot.[1] The aspiration for multipurpose robots is rising, and ongoing technological progress enables increasingly complex robots to be built. However, it becomes challenging to combine and integrate the increasing number of isolated algorithms into coherent robotic systems operating in open-ended, dynamic, and unstructured environments with uncertain, unreliable, and incomplete information, while at the same time coping with limited (onboard) resources. As long as software development implies risks and efforts that are difficult to manage (see the EFFIROB-study [HBK11]), the software challenge will very likely remain an impediment to reaching the next level of robotic applications.

Therefore, systematically addressing the software integration challenge is no longer only a question of good software quality but also a make-or-break factor for future robotic developments, as recognized by the *European SPARC Robotics* [SPA] initiative and its Multi-Annual Roadmap (MAR) [Mar].

[1] www.irobot.com

1.1. Motivation

A service robot is a physical entity that has to keep pace with the dynamics of the real world. Real-world environments are inherently complex, unstructured, open-ended, and mostly unpredictable. Besides, service robots need to cope with limited capabilities, bounded resources, and uncertain information. As illustrated in Figure 1.1, a robot has sensors to perceive the environment and actuators to act within that environment. Software is the main element in between and realizes the robot's basic functions. During execution, the software continuously and reactively interacts with the environment.

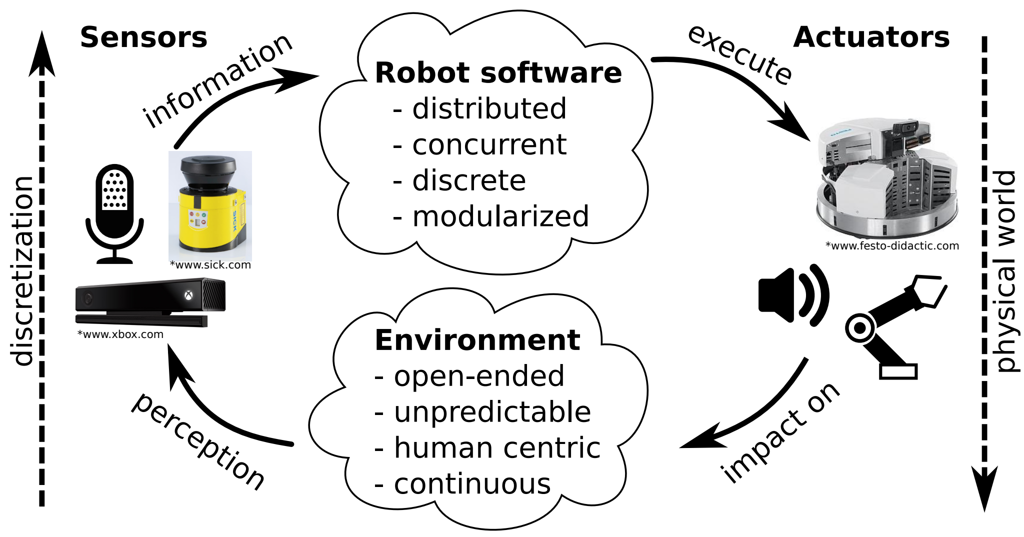


Figure 1.1.: The cycle of interaction between the software and the environment

The interactions with real-world environments impose (more or less hard) constraints on the overall response time of the robot's software in certain situations. This means that the robot system all together needs to react to stimuli within an adequate period of time. A reaction time that is too slow could lead to outdated world knowledge, late execution of actions, and ultimately to the robot failing to accomplish the designated mission. A faster than necessary reaction leads to a waste of scarce resources (such as the available energy in the batteries). Therefore, the overall response time of a system is an important design factor that must be managed throughout the entire life cycle of a robot software system (from design through implementation to runtime). Unfortunately, this aspect of a system, as well as other related non-functional quality aspects, has not yet received the necessary attention within the scientific robotic community. Addressing these aspects is a promising step toward robots becoming more dependable, reliable, predictable, and thus more trustworthy.

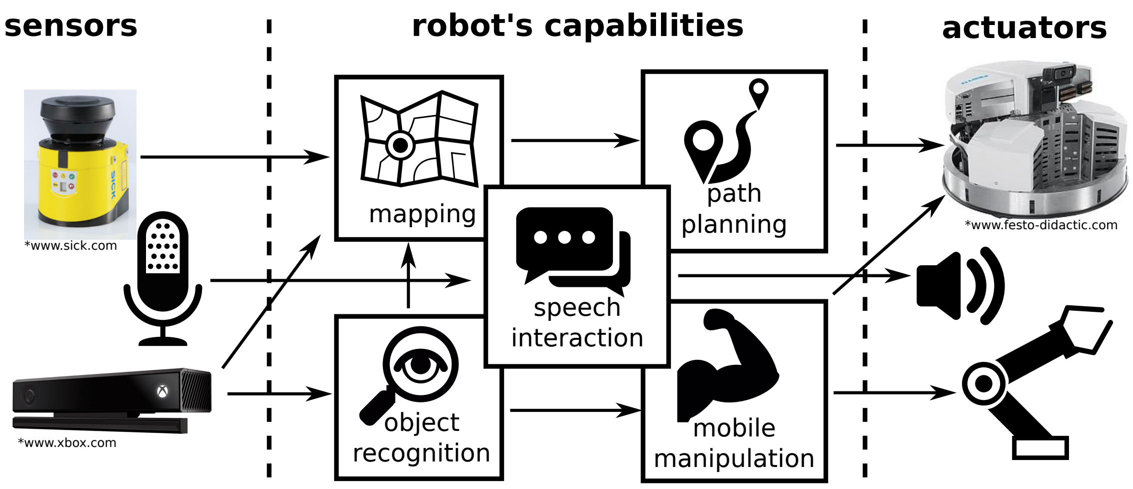


Figure 1.2.: A robot's basic functions and indicated flows of information (the arrows) in the system from sensors to actuators

Over the last decade, the robotic community has presented a wide range of impressive, yet isolated algorithms and basic functions (sometimes called *robot's (basic) skills* [Bon+97]) related to navigation, path planning, object recognition, mobile manipulation, and many others (see Figure 1.2). Individual *skills* depend on the professional knowledge of specialized experts who realize them as sets of software components. At runtime, individual components continuously interact with each other to collectively achieve the current goal and the overall mission. Moreover, some components are executed in parallel, while others are sequenced. Some components share information from the same sensors, while others aggregate or transform information. The information (or data in general) thus flows through the system starting from one or several sensor components, passing through several intermediate components, and finally resulting in actions executed in the actuator components. A typical robotic system consists of several concurrent data-flows forking and joining in related software components (see arrows in Figure 1.2). The need for adequate responsiveness discussed above requires methods for systematic design, development, and management of different communication properties such as latencies and jitters for individual data-flows in the entire life cycle of a robotic software system. Above all, these methods must support the development of individual software components and the integration of these components into new systems, as well as the adaptation of the software system to changing situations.

The design, simulation, and management of different runtime aspects (beyond purely functional needs) are not new in computer science. For instance, automotive and other Distributed, Real-time and Embedded (DRE) domains offer many approaches (such as AADL [AAD04] or MARTE [MAR11]) for modeling, simulating, and analyzing the overall end-to-end latencies and response times in a system. However, these approaches generally lack any sufficient means of composition, which is important in robotics. Systematic means of composition rely on a clear definition of flexibly reusable software components and a clear separation of responsibilities and concerns of the stakeholders involved [SSL12b]. Additionally, software tools are needed to support the design and development of individual software building blocks so that they later seamlessly fit together in different robotic software systems. Such tools provide dedicated views based on software models (see also [Sta+16]) that allow for a focus on local aspects at the right abstraction level. These tools accomplish this without presuming system-level properties that are not relevant in the current development phase. By linking these views, the tools enable consistent contributions to a system.

Hence, the general approach followed in this dissertation involves designing and developing tools and methods that provide a clear definition of communication semantics *between* individual software components and formalized processing schemes *within* individual software components. The resulting models are used for code-generation and as input for a systematic performance analysis using established analysis tools.

1.2. Research Question and Problem Statements

The overall problem addressed in this dissertation is twofold. On the one hand, current robotic software systems have reached a level of complexity that makes structured system-integration methods an imperative rather than an option. On the other hand, performance-related system aspects such as the overall responsiveness of a system are important design factors for achieving dependable, robust, and qualitative service robots. Responsiveness in turn depends on a systematic design of data-flows that form distributed, concurrent, and interacting data-flow chains in the system. It is imperative for the designer of these data-flow chains to have a view of the system with an adequate abstraction level that enables him/her to make appropriate design decisions without having to understand every individual detail of the overall system. All this leads to the following research question:

> *How can data-flow chains be systematically managed throughout the entire life cycle of a robotic software system without breaking established means of composition in a structured software-development workflow?*

There are several keywords in this research question that require further explanation. To begin with, systematic means of composition and a structured software development workflow are necessary for coping with the overall software complexity in robotics. Composition in this sense is the ability to combine several software components into a larger whole, namely the overall robotic software system. Thus, two follow-up questions are:

Research Question 1.1: *What exactly are the established means of composition that need to be implemented in a robotic software-development workflow? What are the important steps in such a development workflow?*

Having such a development workflow as a basis, the next follow-up question is:

Research Question 1.2: *How can collaborative and stepwise design of data-flow chains be enabled without conflicting with (or breaking) required structures and abstractions of that development workflow?*

In order to model data-flow chains, it is necessary to determine the exact system attributes to be modeled. This leads to the following question:

Research Question 1.3: *What are common patterns of typical data-flow chains and how can these patterns be formalized?*

Finally, one of the core motivations of this dissertation is not only to provide nice paper work but also to implement the ideas using model-driven tools to support consistent system design in all relevant development phases. This leads to the question:

Research Question 1.4: *How can such a workflow be effectively supported with integrated tooling, thereby supporting and guiding different developer roles in the development of realistic, real-world scenarios with real robots?*

1.3. Dissertation Objectives

This dissertation is guided by the following overall objectives:

Objective 1.1: *to free component developers from the need to anticipate any target application-related details with respect to system-level communication: This considerably improves the reuse of components in different applications*

Objective 1.2: *to enable system integrators to adequately design and manage non-functional system properties according to the current application using components as gray boxes (i.e., components with explicated configuration options): This clearly separates roles and concerns*

Objective 1.3: *to provide model-driven tools that support and guide all the developers involved in the overall development workflow in designing consistent and robust systems by finding possible design errors and misconceptions in early steps of the workflow: This improves the efficiency of the overall development workflow, because changes can be traced and global system properties can be preserved even if some local parts are exchanged*

Objective 1.4: *to use and integrate performance analysis tools from other software-intensive domains beyond robotics into an overall robotic development workflow: This allows reusing the vast engineering knowledge and time-tested analysis tools from external domains that bear similarities to robotics, such as automotive, embedded systems, and avionics*

Objective 1.5: *to generally facilitate the challenging transition from handcrafted lab prototypes to systematic design and development of high-quality robotic systems*

These objectives constrain and guide the methods and solutions used in this dissertation. While other approaches exist for addressing isolated aspects related to the above objectives, approaches that address the specific demands of robotics and these objectives in a harmonized way are rare. This dissertation aims to make decisive progress in this line of research by illustrating some coherent solutions that are detailed enough (yet not unnecessarily complex) to be useful for the development of real-world robotic systems.

1.4. Core Contributions

The core contributions of this dissertation lie in addressing two fundamental engineering needs. On the one hand, software complexity is effectively managed by a formalized, component-based, model-driven robotic development workflow that provides dedicated, domain-specific views for the involved developer roles. On the other hand, the robot's overall execution performance is systematically designed, managed, and analyzed at the model level before the robot's actual hardware is fully constructed. For each of these two engineering needs in isolation, there are many related approaches, both within robotics and in closely related domains such as automotive. The novel contribution of this dissertation is a systematic and consistent combination of these two engineering needs, specifically tailored to the service robotic domain. This combination requires a careful separation of concerns related to the developer roles involved so that individual developers can focus on those system aspects they are responsible for, while leaving open all other aspects that require further knowledge available earliest in the successive development phases. For each of these development phases and developer roles, dedicated, model-based views are defined and formalized using Model-Driven Software Engineering (MDSE) methods. Moreover, these views are interlinked at a meta-model level to define a consistent model-driven handover of knowledge between the various developer roles and development phases.

Figure 1.3 illustrates the core contributions of this dissertation. Previously, causal dependencies and other communication-related characteristics of interconnected components have been the result of hidden (i.e., implicit) design decisions from component development and were neither accessible for, nor adjustable during, system integration. Yet, such properties are important, application-specific system aspects that should remain open until the required domain knowledge becomes available. For instance, the "right" response time cannot be universally defined for a single component in isolation. Rather, it is a system-level property that may vary considerably across different systems. This issue is addressed by the following contributions:

Contribution 1.1: *Component developers should be able to focus on the component implementation only and not bother with system-level configuration issues. Therefore, this dissertation provides a component model that effectively decouples the component's internal functional implementation from the component's external communication semantics. The functional part within a*

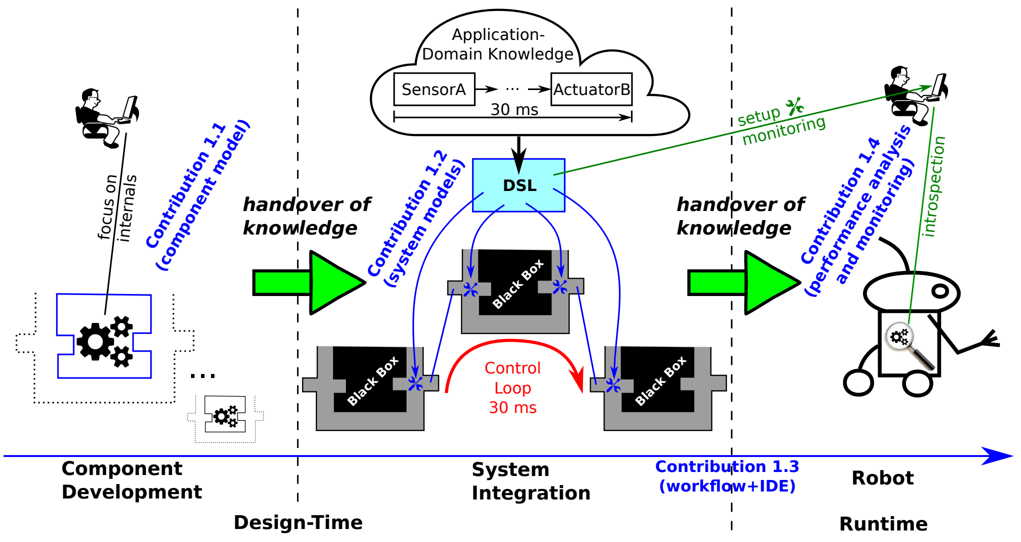


Figure 1.3.: Schematic illustration of the contributions of this dissertation

component implements against an abstract ***communication interface*** *(depicted on the left in Figure 1.3 as the component's inner container), which allows access to a component's input data and provides output data without prematurely binding a certain* ***communication scheme*** *for interaction between components (at the system level). This interface helps to postpone the configuration of certain inter-component communication properties to a later step in the overall development workflow, namely until system integration, where the required domain knowledge becomes available (see next Contribution 1.2). This contribution is addressed in the core Chapter 4.*

Contribution 1.2: *System integrators should be able to design system-level data-flow characteristics without the need to investigate (or even modify) components' internal implementation (i.e., components as gray boxes). Therefore, this dissertation provides a novel Domain-Specific Language (DSL) (pictured in the center of Figure 1.3) that allows defining and configuring system-level* ***cause–effect chains*** *within predefined functional boundaries. This enables system integrators to deliberately design causal dependencies and other communication-related configurations to better satisfy current application-specific needs. This considerably increases the reuse of components in different applications and eases the transition from multi-level plumbing toward designing dependable systems. This contribution is addressed in the core Chapter 5.*

Contribution 1.3: *The introduced DSL should not be stand-alone but form part of the overall robotic software development workflow. Therefore, this dissertation integrates the DSL into a model-driven Integrated Development Environment (IDE) for robotic software. This allows linking the DSL with the robotic component model to automatically generate the corresponding system-level configurations (based on models from this DSL) and to implement consistency* *checks, thereby supporting and guiding different developer roles involved in the overall development workflow. This contribution is mainly addressed in the core Chapter 3.*

Contribution 1.4: *It is important not only to support the design and development workflow but also to specifically add to the runtime robustness, dependability, and reliability of a robot performing in a real-world setting. These aspects, sometimes called* ***non-functional*** *or* ***extra-functional system properties*** *[Sen12], require management of the system's runtime execution performance. This is achieved through two contributions (mainly addressed in the core Chapter 6):*

- *A Compositional Performance Analysis (CPA) based on SymTA/S [Hen+05] is integrated into the overall robotic development process. This allows designing and analyzing the anticipated system performance.*
- *A dedicated logging and monitoring solution is presented that allows observing whether, when, and how often the configured performance aspects are violated at runtime (illustrated at right in Figure 1.3). This again helps to design robust error-handling strategies and to purposefully refine the overall system design.*

Contribution 1.5: *In order to evaluate the usefulness of the proposed solutions, this dissertation discusses a real-world scenario from the robotic domain. This scenario is modeled using the modeling tools developed here, and the performance of the scenario is analyzed with an integrated SymTA/S & Trace Analyser tool. The results of the performance analysis are compared with measured ground-truth values from the robot operating in a real-world setting. This comparison shows that the SymTA/S-based performance analysis sufficiently represents realistic robotic systems. This contribution is mainly addressed in Chapter 7.*

1.5. Dissertation Outline

Figure 1.4 provides an overview of the chapters in this dissertation, which are clustered into three parts: Part I: Introduction and Fundamentals, Part II: The Method, and Part III: Results and Conclusions. Chapter 2 collects related works and provides some fundamentals as a baseline. Next, Chapter 3 provides an in-depth analysis of the overall research problem addressed in this dissertation, and selectively discusses related approaches and solutions, focusing on identifying open scientific gaps to be filled. Subsequently, the three core Chapters 4 to 6 individually describe the three different system views related to *component development*, *system integration*, and *runtime* (the relevant contributions 1.1 to 1.5 are interlinked with the core chapters in Figure 1.4). The main focus within these three core chapters is to formalize important structures and semantics by designing and defining required meta-models. After that, Chapter 7 presents the realized component models and the system model of a real-world scenario. In addition, the results from a Compositional Performance Analysis (CPA) are compared with in-system measurements from a

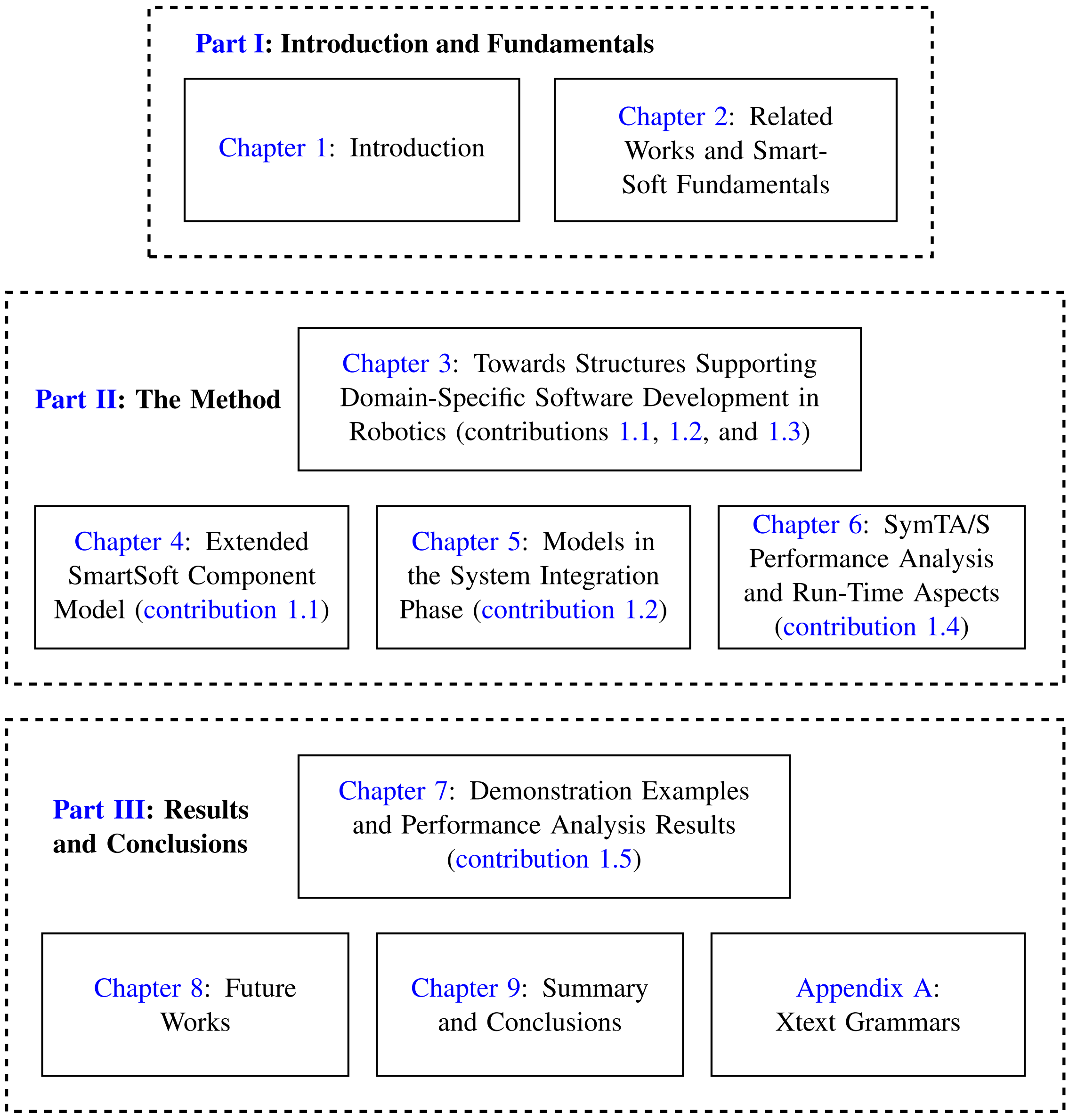

Part I: Introduction and Fundamentals
Chapter 1: Introduction
Chapter 2: Related Works and SmartSoft Fundamentals
Part II: The Method
Chapter 3: Towards Structures Supporting Domain-Specific Software Development in Robotics (contributions 1.1, 1.2, and 1.3)
Chapter 4: Extended SmartSoft Component Model (contribution 1.1)
Chapter 5: Models in the System Integration Phase (contribution 1.2)
Chapter 6: SymTA/S Performance Analysis and Run-Time Aspects (contribution 1.4)
Part III: Results and Conclusions
Chapter 7: Demonstration Examples and Performance Analysis Results (contribution 1.5)
Chapter 8: Future Works
Chapter 9: Summary and Conclusions
Appendix A: Xtext Grammars

Figure 1.4.: Dissertation Outline

robot operating in a real-world setting. Finally, Chapter 8 discusses potential future works that are promising follow-ups to this dissertation, and Chapter 9 concludes the dissertation. Appendix A lists the designed Xtext grammars.

Core Publications

[Lot+15] Alex Lotz, Arne Hamann, Ingo Lütkebohle, Dennis Stampfer, Matthias Lutz, and Christian Schlegel. "Modeling Non-Functional Application Domain Constraints for Component-Based Robotics Software Systems". In: *Sixth International Workshop on Domain-Specific Languages and Models for Robotic Systems (DSLRob'15)*. Hamburg (Germany), 2015.
URL: https://arxiv.org/abs/1601.02379.

[Lot+16] Alex Lotz, Arne Hamann, Ralph Lange, Christian Heinzemann, Jan Staschulat, Vincent Kesel, Dennis Stampfer, Matthias Lutz, and Christian Schlegel. "Combining Robotics Component-Based Model-Driven Development with a Model-Based Performance Analysis". In: *2016 IEEE International Conference on Simulation, Modeling, and Programming for Autonomous Robots (SIMPAR)*. San Francisco, CA, USA, 2016, pp. 170–176.
DOI: 10.1109/SIMPAR.2016.7862392.

[Sta+16] Dennis Stampfer, Alex Lotz, Matthias Lutz, and Christian Schlegel. "The SmartMDSD Toolchain: An Integrated MDSD Workflow and Integrated Development Environment (IDE) for Robotics Software". In: *Special Issue on Domain-Specific Languages and Models in Robotics*. Vol. 7. Journal of Software Engineering for Robotics (JOSER) 1. 2016, pp. 3–19.
URL: http://joser.unibg.it/index.php?journal=joser&page=article&op=view&path%5B%5D=91.

The main ideas of this dissertation have been published in three core publications [Lot+15; Lot+16; Sta+16]. The journal paper [Sta+16] provides the underlying foundations for this dissertation with respect to modeling tools and views of our SmartMDSD Toolchain, which is in productive use in several research projects, and for the *Festo Didactic Robotino3*[2] platform. The core extensions and refinements of this toolchain—as a core contribution of this dissertation—have been described and presented in two successive publications, [Lot+15] and [Lot+16].

Additionally, paper [Sta+16] provides a user survey conducted with our partners from previous research projects. The results of the survey reveal the great benefit provided by model-driven tools during the design and development of complex robotic systems. The two successive publications [Lot+15] and [Lot+16] incrementally describe some of the core ideas of this dissertation, particularly related to modeling *cause–effect chains* and integrating the SymTA/S-based performance

[2] http://wiki.openrobotino.org/index.php?title=Smartsoft

analysis into the overall robotic development process. The dissertation additionally provides a complete background for the published ideas with a much broader scope with respect to related approaches and potential solutions. Furthermore, it provides the latest, updated meta-models and implementations of the presented modeling tools along with a coherent robotic example. This example is extensively analyzed and discussed with an emphasis on architectural decisions made in different phases of the overall robotic development workflow. The results of the performance analysis are presented and discussed in greater detail than was possible in the published papers due to space limitations.

2

Related Works and SmartSoft Fundamentals

The overall goal of this chapter is to present clusters of related approaches and to set the foundations needed to provide the context for the core ideas and approaches in Part II of this dissertation.

2.1. Related Works

This section gives a broad overview of related topics and publications, only shortly relating them to the contents of this dissertation. A more in-depth analysis of selected approaches is given in Section 3.2, with the emphasis on identifying scientific white spots that are addressed in Part II of this dissertation. Moreover, the main method Chapters 4 to 6 individually relate some of the selected works at a more detailed level.

2.1.1. Robot programming and robotic frameworks

Various robot programming techniques have long been a major focus for robotic software development. This has led to various efforts to develop several robotic frameworks (sometimes also reduced to robotic middlewares). A comprehensive literature survey can be found in [ES12]. One early representative of such frameworks is Player/Stage [GVH03] (with its simulation backend Gazebo), which was influential at the beginning of this millennium. It provided at that time a rich set of device drivers for different robot platforms and a simple client/server communication mechanism.

While the Gazebo simulator is still maintained, the Player/Stage framework has been superseded by another more recent framework called Robot Operating System (ROS) [Qui+09]. ROS gained unprecedented acceptance within the robotic community. In its core, ROS provides an n-to-m publish/subscribe communication mechanism and some extensions such as the transformation frames (TF)[1] and action-lib[2]. Due to its wide acceptance within the scientific robotic

[1]ROS TF: http://wiki.ros.org/tf
[2]ROS Actionlib: http://wiki.ros.org/actionlib

community, there is a considerable number of contributed algorithms and device drivers. In recent years, an industry-driven initiative called ROS Industrial (ROS-I)[3] has emerged with the overall goal to make the rich body of algorithms from ROS usable for product development. Interestingly, this initiative struggles with the very liberal initial motivation behind ROS, namely to avoid any structures that might limit the programming freedom of ROS users (see [Lut+14] for more details). While this initial philosophy has been one of the main reasons for ROS becoming so popular in the first place, it also is its Achilles' heel, because the industry heavily relies on structures (according to E.A. Lee's *freedom-from-choice* [Lee10]) and standards for creating value chains and ecosystems (as will be argued in more detail later). Not only are such structures entirely missing in ROS, but they are also difficult to impose afterward, because now many overlapping and incompatible implementations coexist that require heavy redesign, reimplementation, and harmonization of individual ROS nodes. A (too late) initiative to resolve at least some of the initial ROS problems is the recently introduced upgrade called ROS2[4]. While there are some promising ideas, it remains to be seen whether this upgrade will be widely accepted because it proposes radical but badly needed changes, e.g. with respect to a generic middleware abstraction layer. Moreover, even ROS2 will not miraculously cure all problems of the initial ROS. Instead, ROS would need to undergo an even more drastic change toward a stable component model.

As opposed to ROS, such stable structures and a component model with a reusable set of communication patterns have been a major focus from the very beginning of the overall SMARTSOFT idea [SW99; Sch04; SSL12a; Lot+14], which evolved to a fully-fledged model-driven development methodology with the reference implementation called SmartMDSD Toolchain [Sta+16; SSL12b] (both will be introduced with more details in the next Section 2.2).

An interesting framework for this dissertation is the *Robot Construction Kit (Rock)* [JA11], which uses Orocos [Bru01] with its *Real-Time Toolkit (RTT)* as a basis. *Rock* allows specifying time- and data-triggered activation of components, which is comparable to some specifications in this dissertation (as shown in Section 5.4). However, *Rock* lacks any means to easily use e.g. performance analysis tools which, by contrast, is one of the core contributions of this dissertation.

An initiative driven by Japan's National Institute of *Advanced Industrial Science and Technology (AIST)* is the *Robot Technology (RT) Middleware* [And+05] with its open source reference implementation OpenRTM[5]. The interesting aspect about *RT-Middleware* is that it not only resulted in an Object Management Group (OMG) standard called Robot Technology Component (RTC)[6] but also provides an Eclipse-based development toolchain. Around 2005 when *RT-Middleware* was introduced, it soon became a leading initiative that advocated modelling of robotic systems and paved the way for some later approaches such as the BRICS Component Model (BCM) [Bru+13] and the Proteus' Robot Modeling Language (RobotML) [Dho+12] (both will be addressed with more details further below).

[3]ROS Industrial: `http://rosindustrial.org/`
[4]ROS2: `http://design.ros2.org/`
[5]OpenRTM: `http://www.openrtm.org/`
[6]OMG RTC: `http://www.omg.org/spec/RTC/`

Another notable and freely available programming environment is *Microsoft Robotics Developer Studio (MRDS)*[7] for building robotic applications based on .NET. It includes a lightweight, REST-style service-oriented architecture and a set of programming tools. However, due to the vendor lock-in, it never gained widespread acceptance within the robotic research community. *Yet Another Robot Platform (YARP)* [MFN06] is a thin robotic middleware mostly used for humanoid robots with the iCUB platform [Met+10]. A more extensive overview can be found in the survey [ES12] and a deeper history of past robot programming techniques can be found in Chapter 8 of the *Handbook of Robotics* [KS08a].

2.1.2. Model-driven engineering (MDE) and domain-specific languages (DSLs) in robotics

A recent survey [Nor+16] on Domain-Specific Languages (DSLs) in robotics identified 137 relevant publications. Interestingly, over 50% of the identified publications address aspects of architectures and programming of robotic software systems. The authors of the survey conclude that this problem domain must be well understood. However, there might be another (and more likely) interpretation. First, the field of architecture and programming is very wide and diverse encompassing various sub-problems. Therefore, it is only natural that many solutions have been developed. Second—and this is diametrically opposite to the survey's conclusion—this problem domain might **not yet** be sufficiently and satisfactorily solved, which still provokes new contributions. A high number of *component models* (see further below) supports this assumption. In other words, as the software complexity in robotics is rapidly increasing so is the pain, which emphasizes the need for efficient solutions to alleviate this problem. Moreover, using abstractions and thus model-driven approaches might be especially well suited for this kind of problems, which also explains the high number of contributions. In any case, it is safe to assume that model-driven approaches for architectures and programming are still needed, which motivates the contributions in this dissertation.

DSL engineering is also a growing trend in computer science in general. Various references in the literature such as [Voe13; BCW12; Fow11; Ste+11; Gro09] address this topic from a general perspective. In this respect, DSL engineering is a science that calls for a thorough understanding of the involved problem domain (including the required abstractions and structures) and requires a lot of experience in designing and implementing consistent, functional, and helpful model-driven tools. Robotics is no exception in this regard. This dissertation extensively uses model-driven techniques (see Section 3.3) and builds on over seven years of experience in building model-driven solutions for robotic software development problems.

A widespread problem within the robotic domain is the high diversity of available robotic algorithms. One related approach to handle diversity, specifically focusing on product diversity, is the Software Product Line (SPL) [Bos00; CN01] approach with a (rather recent) trend toward Dynamic Software Product Lines (DSPLs) [BHA12; Cap+14]. One of the notable approaches

[7]MRDS: `http://msdn.microsoft.com/en-us/robotics/default.aspx`

that have applied the SPL idea in robotics is the *Hyperflex* toolchain [GB11; GB14], which uses feature models to describe dependencies between sets of related components. SPLs are generally suited for well-understood domains where the reasonable combinatorial diversity of systems is known and can be globally expressed in advance. However, due to the natural heterogeneity of robotic systems, SPLs scale badly when systems need to be flexibly composed in a priori unknown and unpredictable ways, let alone dynamically adapt to changing conditions at runtime. While this issue is currently under discussion in the context of the DSPLs [BHA12], it remains to be seen whether this approach is applicable to real-world robotic applications of realistic size and complexity.

A frequently addressed problem in robotics is the increasing software complexity. There are hardly any approaches in robotics that do not tackle complexity management in one way or another, which is why a comprehensive overview of the related approaches would not be feasible. However, some selected approaches, most notably related to Component-Based Software Engineering (CBSE) [HC01] and Service-Oriented Architecture (SOA) [Erl07], specifically focus on this problem domain. CBSE has already gained widespread acceptance in the robotic domain as well, which is evident from the various *component models* (see below). SOA also gains ever more momentum in robotics and is a compatible way of addressing not only complexity by modularity (as with CBSE) but also diversity (as with SPLs) and workflow flexibility (in contrast to the traditional workflow models such as the V-model that are too inflexible[8]). A combination of CBSE and SOA thus makes sense. A notable approach addressing this combination is SMARTSOFT [Sta+16] with its open-source reference implementation called “SmartMDSD Toolchain” (Section 2.2 provides a more detailed introduction).

Over the last decade, several *component models* for robotic systems emerged such as the BRICS Component Model (BCM) [Bru+13], the RobotML [Dho+12], the Robot Technology Component (RTC) [And+05], and especially SMARTSOFT [SW99; SSL12a; Sta+16]. Two consecutive journal articles [BS09; BS10] provide a broader overview of component-based approaches in robotics. While each of these *component models* makes sense in one way or another, one of the distinguishing factors (from the scope of this dissertation) is their support for *system composition* and *runtime adaptation*. To the best of my knowledge, only the SMARTSOFT approach provides techniques that cover both areas (see [Sta+16] and [Lot+14; SS14; SLS11b] for more details), which makes SMARTSOFT and `SmartMDSD` attractive for this dissertation as an underlying foundation. However, the basic structures and abstractions in this dissertation are independent of any particular component model as long as the used component model provides rich enough semantics and allows extensions toward management of non-functional system aspects as advocated in Part II.

[8] https://harmonicss.co.uk/project/the-death-of-the-v-model/

2.1.3. MDSE in other domains close to robotics and MDSE community involvements

Talking about *software modeling* without mentioning the Unified Modeling Language (UML) [UML15] would leave an incomplete picture. On the one hand, UML has been one of the most influential developments that changed the way how software is engineered today and facilitated model-based approaches in general. On the other hand, there has been a lot of criticism about UML lately such as in [Bel04]. The core of the criticism is that UML has been (too often) treated as a "silver bullet" that miraculously solves any software-related problem. However, as with any approach, it is only as good as the engineers who apply it. Regardless of the opinion, even in the worst case, one still can learn from the mistakes made with UML for better designs in tailored DSLs. In the context of UML, some notable approaches are OMG's Model-Driven Architecture (MDA) and the System Modeling Language (SysML) [Sys12] specification. The former provides fundamental ideas with respect to platform-independent modeling and the definition of several meta-model layers, with Meta-Object Facility (MOF) [Mof] playing a significant role in specifying a meta-model for UML. The latter SysML—a subset and an extension of UML—is a more focused modeling language than UML with extensions toward requirement specifications and parametric diagrams (which are basically used to express physical constraints that later can be used for various model analyzes). Compared with UML, SysML puts the focus more on the overall system modeling, which makes SysML appealing for robotics. However, from the perspective of robotics, SysML might be too generic (i.e., too general purpose), thus missing the required structures related to non-functional properties such as those advocated in this dissertation.

Another well-known OMG standard is Modeling And Analysis Of Real-Time Embedded Systems (MARTE) [MAR11]. It provides a *General Component Model (GCM)* that includes a flow-port and a client-server port specification. Moreover, *MARTE* allows a detailed specification of hardware-related details and some timing-related annotations. However, as argued in [Lot+16], central concepts are often hidden in a freedom-of-choice philosophy offering all kinds of alternative, coequal, and overlapping concepts. These concepts are too fine-grained (e.g. read and write operations on buffers). This directs the focus of the *MARTE* user to minor aspects and leaves him/her alone with many system-level design choices, thus preventing him/her from "seeing the wood for the trees". This either leads to a refusal of the use of *MARTE* in the first place, or results in non-interoperable, hard-to-reuse components.

A recent OMG initiative is the Unified Component Model (UCM) [UCM]. Its recent "beta" specification looks promising and deserves further investigation in future works.

An interesting standard originating from the avionics domain is the Architecture Analysis & Design Language (AADL) [AAD04]. The interesting part about AADL is its flow-latency analysis specification [Han07], which allows a timing analysis of end-to-end data-flows for interacting threads. An ad hoc example of how to manually apply this analysis to a robotic wheelchair system is presented in [BFA14]. This makes the specification appealing for this dissertation as a promising candidate for realizing an integrated performance analysis, which is further explained in Chapter 6 and Chapter 8.

Another closely related domain is automotive with its industry-driven initiative called *AUTOSAR* [Aut]. Its goal is to support the design and development of automotive electrical and electronic (sub-)systems with independent software modules that are highly modular, scalable, transferable, and reusable. In this respect, the automotive domain faces the same challenges as those by the domain of robotics and draws the same conclusions with similar solutions. Eventually, the domains of robotics and automotive might converge into one unified field of research with shared solutions. However, now it is not yet clear in what ways this unification might happen as there are many concurrent developments and discussions going on around *AUTOSAR*.

A recent trend—mainly driven by German industries—concerns the umbrella term *Industry 4.0* (sometimes also called *factory of the future* or in German "Industrie 4.0") with its *OPC Unified Architecture (UA)*[9]. The focus of *OPC UA* is on providing a platform-independent and extensible communication infrastructure. Extensibility is realized by the so-called *companion standards*. A *companion standard* is a communication profile that extends the basic communication structures with additional communication semantics. Similar to UCM and *AUTOSAR*, *OPC UA* is a relatively recent initiative, which is worth keeping an eye on and which might become relevant for robotics in the near future as well.

Besides the industry-driven initiatives, there are several community-driven activities (both from research and from the industry). For instance, the euRobotics AISBL *Topic Group on Software Engineering, System Integration, System Engineering*[10] shapes the European roadmapping in software systems engineering for robotics. Another relevant community forum is the *Technical Committee on Software Engineering for Robotics and Automation (IEEE RAS TC-Soft)*[11]. Some notable research community activities are the *Domain-Specific Languages and Models for Robotic Systems (DSLRob)* workshop series[12], the *International Conference on Simulation, Modeling, and Programming for Autonomous Robots (SIMPAR)*[13], as well as the *Journal of Software Engineering for Robotics (JOSER)*[14]. The three core publications of this dissertation [Lot+15; Lot+16; Sta+16] are a direct contribution to these community activities.

A recent European initiative, starting January 1, 2017, is the *RobMoSys* project [Rob]. This initiative envisions a component-based robotic software ecosystem with composable models and systems for robotic systems of systems. The RobMoSys consortium comprises key research players from the robotic model-driven community, thus building on their accumulated knowledge for pushing the robotic software engineering methods to the next level. Moreover, RobMoSys encourages community involvement through open calls. This initiative is expected to have a severe impact on the overall robotic engineering landscape in the near future. The underlying ideas of this dissertation have been most influential in the *RobMoSys* project proposal phase and directly contribute to the underlying body of knowledge for *RobMoSys*.

[9]OPC UA: `https://opcfoundation.org/about/opc-technologies/opc-ua/`
[10]EU SSE TG: `http://www6.in.tum.de/Main/TG-Software-Systems-Engineering`
[11]RAS TC-Soft: `http://robotics.unibg.it/tcsoft/`
[12]DSLRob 2015: `http://www.doesnotunderstand.org/public/DSLRob2015`
[13]SIMPAR 2016: `http://simpar2016.org/`
[14]JOSER: `http://www.joser.org/`

2.1.4. Models of computation (MoC), non-functional system properties, and quality of service

Models of Computation (MoC) (sometimes also *data-flow models of computation*) is an umbrella term often used by E.A. Lee as in [Lee01]. In [BFG17], Bouakaz et.al. distinguish between *static* and *dynamic* MoC. One of the most widely known *static* MoC is Lee's Synchronous Data Flow (SDF) [LM87]. SDF defines synchronous interaction between components, meaning that each received input data directly triggers an update cycle of a currently receiving thread (in a current component). This policy is easy to understand and easy to analyze but not always reasonable. In some cases, it is necessary for the components to interact asynchronously using a different MoC such as the Kahn Process Network (KPN) [Kah74], which represents *dynamic* MoC. KPNs are more flexible and allow parallelism. However, for realistic robotic systems, it is not possible to only use one of these two MoC in isolation. Instead, robotic systems often need to use both MoC in any combination depending on the currently relevant needs. These two MoC play a central role in this dissertation for the design and for the analysis of end-to-end timings of the so-called *cause–effect chains* (see Section 4.1.2 and Section 5.3).

The general reason to address different MoC in the first place is closely related to the general need to manage *non-functional system properties* in robotics. As argued in Chapter 3, management of non-functional system properties is a fundamental prerequisite for a successful development of robotic products. Similarly, the management of non-functional properties has been a major focus in other domains beyond robotics. For instance, within the embedded domain, the OMG MARTE [MAR11] standard facilitates the management of non-functional aspects. However, as argued above, MARTE lacks the required abstraction level for robotics (i.e., it is too finely grained) that would allow the management of such properties in dedicated views and throughout several consecutive development steps.

Another notable work from the embedded domain is the dissertation by Sentilles [Sen12], which coined the term *extra-functional* properties. From its basic motivation, *extra-functional* properties are compatible with the ideas in this dissertation. However, while Sentilles addresses component-based systems, he is vague about how such systems can be systematically composed by system-integrators from different application domains (who are not necessarily technology experts). By contrast, the separation of component developers, system-integrators, and other developer roles is a major focus maintained in this dissertation.

A technical reference is Pletzer's Timing Definition Language (TDL) [Ple12], which is generally based on Logical Execution Time (LET) [Gho+04]. The general idea of LET is quite appealing for robotics as it increases platform independence by decoupling the logical execution time from the physical execution. Moreover, the globally required synchronization of clocks is not a hurdle anymore due to the availability of sophisticated time-synchronization mechanisms. However, the problem of LET, and thus also of TDL, is the global clock beat that requires pressing individual and concurrent executions into globally defined time frames. That simply scales badly for real-world robotic systems of realistic size and complexity. Moreover, while TDL directly supports synchronous interaction between components, asynchronous interaction is not handled by a real-time scheduler but is executed in the remaining spare time, which is a severe limitation on using different configuration alternatives.

Speaking of *real-time scheduling*, this topic has a long history in computer science with approaches such as [LL73], which uses *Rate-Monotonic Scheduling (RMS)*, and [Der74], which uses *Earliest Deadline First (EDF)* scheduling. A frequent challenge is to derive a realistic Worst-Case Execution Time (WCET). An overview of respective approaches is given in [Wil+08]. As argued in [BS05], safety-critical systems are traditionally developed using one specific scheduling technology for the entire system. However, as further argued in [BS05], new applications in the embedded domain—and this also holds for robotic systems—demand for more flexible and heterogeneous approaches where several scheduling mechanisms are composed and used in combination to collectively provide overall system functions with different levels of guarantees. One of such flexible and promising approaches is Reservation-Based Scheduling (RBS) [AB01]. Within the scope of this dissertation, the following conclusions can be drawn. First, robotic systems demand flexible, composable, and adjustable scheduling mechanisms as argued above. Second, roboticists should use already available and matured schedulability analysis tools such as SymTA/S [Hen+05] or MAST [MAS]. Last but not least, these tools and methods should be easily accessible to and usable by roboticists without the need to understand every detail of the underlying mechanisms which is one of the main objectives of this dissertation (as also stated in Objective 1.4 in Chapter 1).

Despite all the available approaches, it still is impossible in practice to systematically design even the most basic non-functional aspects such as end-to-end guarantees under consideration of the different responsibilities and concerns of the involved developer roles. Thus, the influence of local changes on the global system properties has to become explicitly known and traceable. Addressing this challenge promises a general improvement of *changeability* (i.e., simple exchange of individual parts) and *composability* (i.e., structured composition of systems out of reusable building blocks), and it is thus within the scope of this dissertation.

The above references address *non-functional* aspects in the sense of timings. Another (more general) way of addressing *non-functional* properties lies in the sense of the overall Quality-of-Service (QoS) of a system. QoS is a huge field of research and a cross-cutting concern that addresses many different aspect of a system at different levels of abstraction. One such QoS aspect is *robot safety* [TVS14], which (i) involves the robot hardware components that need to limit the physical forces and ensure electrical safety; (ii) requires functional correctness, determinism, and dependability of software; and finally (iii) requires a predictable robot behavior in operation. *Robot safety* is an emerging and increasingly important field of research with notable works such as [ISK15; AKS16]. While this overall topic does not directly fall within the main focus of this dissertation, improving the overall reliability and dependability of a system is considered important and can be seen as a prerequisite for *robot safety*.

Another QoS aspect is related to *fault tolerance* (i.e., robust system behavior even in the presence of failures). An overview of various fault-prevention mechanisms is given in [Bro+14]. A prerequisite for handling faults is the ability to detect them. [AC04] provides an overview of such fault-detection mechanisms. A common approach to handling faults involves equipping a system with redundant components that allow switching to a backup strategy, which again allows resuming operation even if sometimes with gracefully degraded performance. While fault handling is not a major focus of this dissertation, the monitoring solution discussed in Section 6.3 can also be used to detect faults.

A loosely related topic to *non-functional* properties concerns the terms *runtime adaptation* and *self-adaptive systems* [Wey+13; WIS13; IW15] (or more specifically, *architecture-based self-adaptation* [Ore+99]). The overall idea is that a system is divided into at least two layers, with the lower layer(s) implementing the *managed* system, which directly interacts with the environment and with the higher layer(s) implementing the *managing* system, which adapts the underlying *managed* system if needed. The *managing* system thus typically comprises the *MAPE-K* [IBM06; Wey+13] components (Monitor, Analyze, Plan, Execute, and Knowledge), which altogether implement the adaptation control loop for the underlying *managed* system. Adaptation in this sense means reacting to a changing availability of resources, system faults, or user inputs. While *self-adaptive approaches* already proved their usefulness in several software-intensive systems [WIS13], their application in robotic systems is rare (besides some notable exceptions such as [Edw+09]). Robotic systems typically require further structures as e.g. shown in [Lot+13; Lot+14; IR+12]. Given the scope of this dissertation, the definition of configuration options for the so-called *cause–effect chains* can be considered as a variation point that can facilitate an informed *runtime adaptation* (and more generally *behavior coordination* as in [SLS11b; Lut+14]).

2.1.5. List of own publications

The following list clusters my own publications[15] that contributed—some directly and some indirectly—to the ideas in this dissertation.

Monitoring: [LSS11] provides a generic solution for runtime monitoring and introspection of robotic software components. This topic is further addressed in Section 6.3.

Anytime: [LSS12] presents a mechanism to balance calculation costs and expected solution quality using a robotic example for object recognition. The proposed object recognition approach uses the *bag-of-words* algorithm, which is modified in such a way that it becomes an *anytime* algorithm. This work can also be used to improve the deterministic execution of individual components—or more precisely of tasks in a component—which is also mentioned in Section 5.3.

Robot behavior: Two works [SLS11b; Lut+14] address the problem of a flexible robot behavior coordination using a tailored internal DSL called *Smart Task-Coordination Language (SmartTCL)*. In this dissertation, this behavior coordination level is considered in the sense that the exposed configuration options of the so-called *cause–effect chains* provide an additional source of information and a variation point for an improved and informed behavior coordination.

[15] First-author publications are underlined; all other references are co-author publications.

Variability management: The three publications [Lot+13; Lot+14] and [IR+12] together address a topic that is closely related to the behavior coordination above with respect to the quality-aware operation of a robot. In other words, these works address a systematic management of functional and non-functional runtime variability in a system for improving the overall execution performance. On the one hand, these works improve the behavior coordination level in general. On the other hand, they provide a good example regarding how model-driven approaches can be adopted to robotic needs.

MDSE: Two book chapters [SSL12a; SSL12b] and three papers [Sch+09; Sch+13; Sch+15] broadly address the topic of applying model-driven engineering methods to robotics to deal with the ever-increasing software complexity. These works provide the underlying body of knowledge for this dissertation with respect to model-driven engineering in robotics.

Core-papers: The three core papers [Lot+15; Lot+16] and [Sta+16] directly address some of the main ideas of this dissertation. These papers are introduced with more details in Chapter 1 and are referenced several times throughout this dissertation.

2.1.6. Summary

In summary, the recurring overall challenges faced when building robotic software systems can be clustered into two main (sometimes opposed) engineering needs: the need to cope with the overall *software complexity* and the need to manage *non-functional* (sometimes also called *extra-functional* [Sen12]) *system properties* (see Figure 2.1).

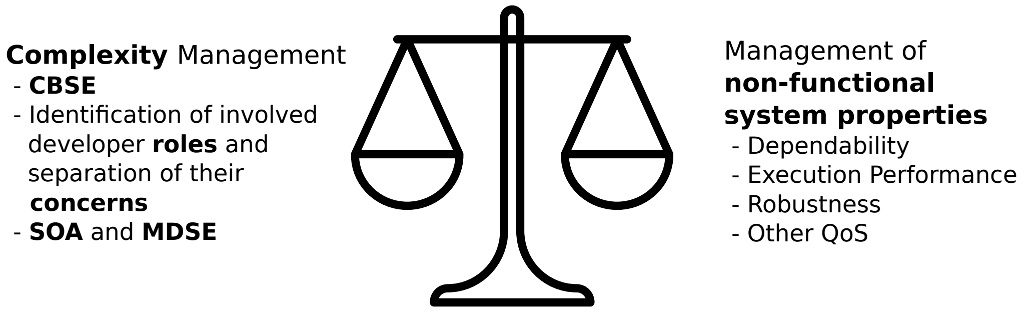

Complexity Management
- **CBSE**
- Identification of involved developer **roles** and separation of their **concerns**
- **SOA** and **MDSE**
Management of **non-functional system properties**
- Dependability
- Execution Performance
- Robustness
- Other QoS

Figure 2.1.: Balancing two common engineering needs in robotic software development

Realistic robotic software systems need to combine and integrate many functional blocks provided by different robotic experts. Ultimately, this integration challenge will lead to the use of a CBSE method in one way or another. As already demonstrated several times, e.g. in [SSL12b] and in [Sta+16], successful CBSE requires a structured overall development workflow that supports the involved developer roles in their individual work considering their individual needs and concerns. Moreover, SOA and MDSE methods become increasingly popular—also within the service robotic domain—to formalize methods for coping with the overall software complexity. Notable examples include RobotML [Dho+12], RTC [And+05], BCM [Bru+13], and SMARTSOFT [Sta+16]. Yet, as soon as the robotic demonstrators and showcases must be transferred into products (or at least to be robustly executed in a dynamic environment more than once), an additional engineering requirement comes up, which is related to managing the overall system quality aspects such as the overall execution performance, dependability, deterministic execution, safety, and other *non-functional* system aspects. These *non-functional* system properties become the main selling points of a product and must be designed and developed as an integral part of the overall robotic development process. However, as can be seen in related approaches such as AUTOSAR [Aut] (from automotive) and AADL [AAD04] (from avionics), the management of *non-functional* system properties often requires an extensive level of details of a system which contradicts the general CBSE principles related to encapsulation, information hiding, and loosely coupled, reusable building blocks. Consequently, finding an approach that effectively balances these two core engineering needs in a systematic and consistent way is an important Objective 1.2 of this dissertation (as further explained in Chapter 3).

2.2. The SmartSoft Fundamentals

This dissertation is influenced by the general ideas behind the term SMARTSOFT, which has been coined by the research group around Prof. Dr. Christian Schlegel. Consequently, this chapter provides some fundamentals of SMARTSOFT and relates them to the ideas in this dissertation.

The overall SMARTSOFT approach [SW99; Sch04] covers a rich set of techniques and methods developed and refined over the last decade in various research and industrial projects. Over the last decade, SMARTSOFT has been refined and extended by a novel Model-Driven Software Engineering (MDSE) approach [Sta+16], which is implemented using Eclipse-based modeling tools. The SMARTSOFT approach includes both a sophisticated robotic component model with explicated service definitions and a flexible development methodology particularly optimized for component-based software design. This facilitates the development of robotic systems supporting different roles and clearly separating concerns at all levels [Sta+16]. Moreover, the SMARTSOFT approach supports a stepwise variability refinement in the entire life cycle of a robot starting from individual component development, going through a consistent system integration, and finally leading to coordinated execution on the robot [Lot+14].

This dissertation seamlessly contributes to this field by providing a more abstract communication interface within components, which allow postponing the exact configuration of the component's communication characteristics until the system integration phase where the relevant application-specific requirements become known. Moreover, a novel DSL enables system integrators to effectively estimate and consistently design performance-related system aspects so that application-specific end-to-end guarantees can be met.

2.2.1. The SmartSoft communication patterns

The SmartSoft Communication Patterns are first presented in [SW99] and are extensively described in detail in the dissertation of Schlegel in [Sch04].

There are various general definitions in the literature for a software pattern. Most notably, Buschmann et al. [Bus+96] present the following descriptive pattern properties:

1. *"A pattern addresses a recurring design problem that arises in specific design situations, and presents a solution to it."*
2. *"Patterns provide a common vocabulary and understanding for design principles."*
3. *"Patterns support the construction of software with defined properties."*
4. *"Patterns help you build complex and heterogeneous software architectures. (. . .) Patterns help you to manage software complexity."*

Buschmann et al. [Bus+96] further define that a pattern typically consists of three parts: a *context*, a *problem*, and the *solution*. In this respect, a Communication Pattern [SW99] is an analogous way to provide encoded experience from skilled software engineers for recurring problems in the service robotic domain. That is, the *context* is the communication between software components, where a general-purpose middleware provides an infinite number of design alternatives. In this *context*, there are recurring communication *problems* (conforming to Statement 1), such as publishing data on a regular basis for several subscribers and querying data on demand. These *problems* lead to recurring *solutions*, such as publish–subscribe and request–response communication characteristics. These *solutions* further need to define the communication *properties* (according to Statement 3) such as involved buffers and the synchronicity of communication. In the terminology of SOAs, a Communication Pattern defines a contract between a service requestor and a service provider. Giving each Communication Pattern a distinctive name provides a *common vocabulary* (corresponding to Statement 2) to easily understand the involved design principles. Overall, Communication Patterns allow building complex, component-based software systems, thereby facilitating better coping with the overall software complexity (analogous to Statement 4).

In this form, Communication Patterns provide the basic vocabulary and the underlying semantics as a foundation for a successful component model specification using MDSE methods such as that demonstrated in [SSL12b]. This dissertation extends the previous set of Communication Patterns by an additional, generic *Push* Communication Pattern that provides enough details for component developers to fully implement the respective component's core functionality while leaving open those communication configuration options that need to remain open until system integration where they are fixed considering the then available domain knowledge. Hence, system integrators are enabled to adjust components so that they better go with their currently developed system and to purposefully design performance-related aspects (see Chapter 5 for further details).

2.2.2. Toward QoS and variability management in a robotic development process

A previous work [Lot+14] has presented an approach to manage design-time variability and to design runtime adaptability using a dedicated DSL. Runtime adaptability is an important ingredient in robotics to deal with unpredictable, open-ended, and unstructured environments while improving the robot's overall execution performance and QoS.

This dissertation contributes to this line of research by addressing an additionally important system property related to the robot's overall QoS, namely the design of end-to-end delays in a system. Since service robots have to act in real-world environments, adequate response times are important factors for the robot's QoS. Response time in this form is the end-to-end delay from sensing information of the robot's surrounding to executing relevant (re-)actions. The necessity to react and respond in time can impose different QoS guarantees depending on the current situation and the mission to accomplish. For instance, in the case where a robot has to avoid a sudden obstacle, the response time is safety-critical and needs to remain within sharp boundaries. In another case, where the robot reacts with an answer in a dialog with a human, response time is not that critical with a soft threshold leading to dissatisfaction with the robot's performance in the worst case. The key point addressed in this dissertation is that such system aspects should be easy to design in an overall robotic development process and should not result from hidden (i.e., code-defined) choices.

2.2.3. Platform-independent component model with different middleware mappings: from shared memory to DDS

One of the main requirements for a definition of a SMARTSOFT component, including its provided and required services, is the independence of a platform (i.e., middleware, operating system, and programming language). This platform-independence allows using any currently popular, widely applied, and time-tested middleware solution without affecting the communication semantics of the service definitions or being forced to adjust the component's internal implementation. The component's internal implementation can well be platform-specific[16]. However, the component hull and its service definitions must remain platform-independent. In SMARTSOFT, this is achieved by a generic middleware abstraction layer that allows mapping the SMARTSOFT Communication Patterns on any arbitrary middleware solution (as shown in Figure 2.2).

At the time of writing, SMARTSOFT has been mapped to several middleware solutions ranging from a very lightweight 8-bit micro-controller implementation using an embedded real-time operating system to an OMG DDS-based implementation. An overview is given below:

- **Embedded/SmartSoft (closed source):** a very lightweight version based on shared memory using global variables (demonstrated on an Atmel 8-bit micro-controller with a real-time OS) (see [Sch+09] for more details),

[16]This allows optimally exploiting the platform's specifics (wherever needed).

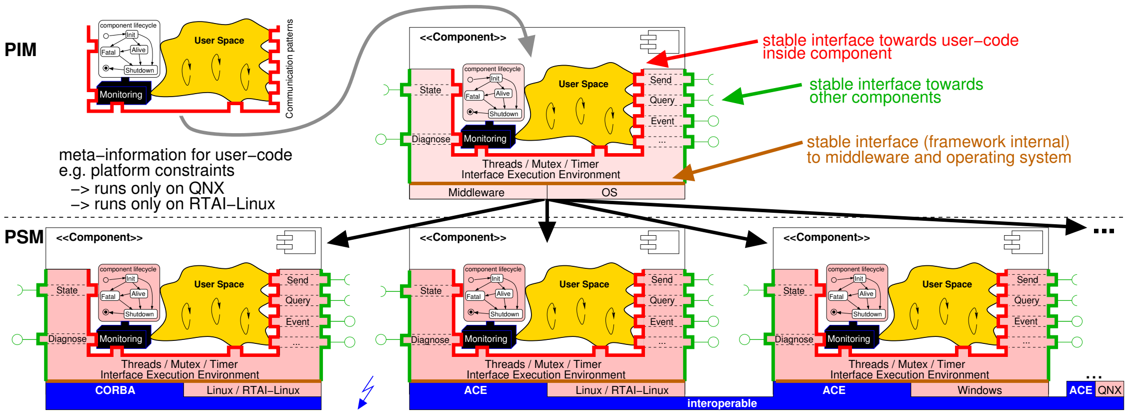

PIM
User Space
Communication patterns
Monitoring
meta–information for user–code
e.g. platform constraints
–> runs only on QNX
–> runs only on RTAI–Linux
<<Component>>
State
Diagnose
Send
Query
Event
Threads / Mutex / Timer
Interface Execution Environment
Middleware
OS
stable interface towards user–code inside component
stable interface towards other components
stable interface (framework internal) to middleware and operating system
PSM
CORBA
Linux / RTAI–Linux
ACE
Windows
QNX
interoperable

Figure 2.2.: Different middleware mapping alternatives and the platform-independent component container [SLS11b]

- **0MQ/SmartSoft (closed source):** A prototypical implementation based on 0MQ implemented on QNX OS,
- **ACE/SmartSoft[17] (open source):** a variant based on plain message passing using the Adaptive Communication Environment (ACE) [HJS03a], implemented and tested on both Windows and Linux OS (at the time of writing, ACE/SMARTSOFT is the reference implementation and is widely applied as baseline in several national and European projects involving external partners from both academia and the industry),
- **CORBA/SmartSoft[18] (open source):** OMG's Common Object Request Broker Architecture (CORBA)-based implementation using the ACE / TAO environment[19] (a time-tested and widely applied middleware mapping),
- **DDS/SmartSoft (closed source):** Prototypical implementation based on OMG's Data Distribution Service (DDS) using the commercial *RTI Connext DDS*[20] framework.

This dissertation extends the overall SMARTSOFT approach by an additional Communication Pattern that allows late configuration of a component at the model level. As a reference implementation, ACE/SMARTSOFT has been refined, extended, and published as open source on SourceForge[21].

[17]ACE/SmartSoft: `http://sourceforge.net/projects/smartsoft-ace/`
[18]CORBA/SmartSoft (including the Eclipse-based SmartMDSD Toolchain): `http://sourceforge.net/projects/smart-robotics/`
[19]ACE/TAO: `http://www.cs.wustl.edu/~schmidt/TAO.html`
[20]RTI Connext DDS: `https://www.rti.com/products/dds/`
[21]ACE/SmartSoft Version 3: `https://sourceforge.net/p/smart-robotics/smartmdsd-v3`

2.2.4. The SmartMDSD Toolchain

Apart from framework-level refinements, this dissertation particularly extends our (at the time of writing productive) SmartMDSD Toolchain [Sta+16]. The SmartMDSD Toolchain supports the overall robotic development workflow by providing dedicated modeling views for the different workflow steps (such as component development and system integration), thus directly supporting the involved developer roles. This dissertation extends both the component- and the system meta-models to enable the respective developers to design additional aspects related to system performance. The relevant extensions have been combined in a new implementation of our SmartMDSD Toolchain and are published as an open-source technology preview (i.e., a reference implementation) on SourceForge[22].

[22] SmartMDSD Toolchain V3: `https://sourceforge.net/p/smart-robotics/smartmdsd-v3/ci/master/tree/toolchain`

Part II.

The Method

*"Stop thinking of constraints as a universal negative—
freedom from choice!"*

—Edward A. Lee

3 Toward Structures Supporting Domain-Specific Software Development in Robotics

Nowadays, two main trends in robotics can be observed. On the one hand, the robotic research community showcases impressive lab prototypes with isolated robotic capabilities. On the other hand, there are only a few rather simple robotic products in the market. At the same time, there is an ongoing trend toward more automation in factories, and several potential application domains such as agriculture, automotive, logistics, and healthcare could benefit from robotic technologies. The lack of advanced robotic products suggests some impediments that must be removed and some challenges that must be overcome, which otherwise prevent companies from integrating and combining isolated capabilities into high-quality products. Although addressing this problem might involve lots of other (non)scientific efforts beyond this dissertation, the structures and abstractions mentioned in this dissertation decisively contribute toward this goal. The necessity to address the challenging transition from lab prototypes toward high-quality products (see Figure 3.1) is specified as a general Objective 1.5 in Chapter 1.

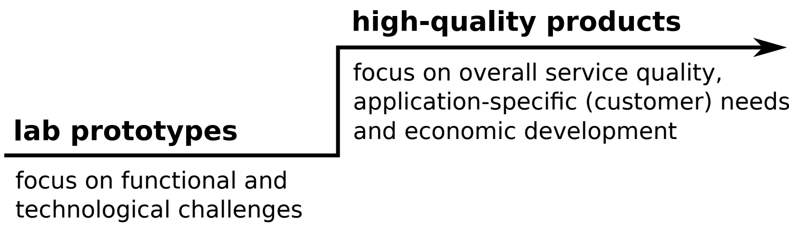

high-quality products
focus on overall service quality, application-specific (customer) needs and economic development
lab prototypes
focus on functional and technological challenges

Figure 3.1.: The step-change from lab prototypes toward high-quality products

First, to better understand the gap between lab prototypes and products, it is necessary to have a closer look into their individual focus (as illustrated in Figure 3.1). On the one hand, academic research tends to focus on fostering new robotic technologies and on improving isolated capabilities of a service robot in technology domains such as mobile manipulation, object recognition, and Human-Robot Interaction (HRI). On the other hand, developing products (robots or not) require a thorough understanding and purposeful satisfaction of customer needs while saving costs and efforts to remain economically competitive. Consequently, the step from lab prototypes to products involves a paradigm shift that affects all levels of a software system. For instance, common metrics to assess the maturity of a system are the Technology Readiness Levels (TRLs) from the European Robotics Multi-Annual Roadmap (MAR) [Mar]. Most lab prototypes get stuck at TRL 4 (or at the latest on the way toward TRL 5). Moving to a TRL 5 or even TRL 6 might require a complete redesign and reimplementation of the entire system, because required quality aspects were not considered from the beginning. This leads to great disappointments when industry seeks to use academic technologies because lab prototypes fail to meet even the most basic industrial quality expectations, resulting in high efforts to raise the TRL.

Common approaches in the industry to cope with complexity and to reduce costs and efforts involve the creation of value chains where individual companies become experts in sub-fields and supply high-quality components used by other companies. This allows sharing risks and efforts while reducing the overall time to the market. One of the fundamental prerequisites for realizing such value chains is a clear separation of responsibilities and concerns between individual companies. While the relevant collaboration terms can be defined on the bilateral contract level, examples such as AUTOSAR [Aut] demonstrate that standards and software structures implemented in tools allow the application of such terms more directly and efficiently.

Therefore, finding such structures and abstractions for robotic software development is one of the essential motivations for this dissertation. Interestingly, academia sometimes tends to consider structures as constraints that limit their freedom of choice. However, according to Edward A. Lee, constraints should not be seen as a universal negative but rather as an opportunity (*freedom-from-choice* [Lee10]). Structures allow focusing on the real problems at hand without being forced to make arbitrary decisions about system aspects that are not relevant at the moment and that can likely lead to incompatible system parts afterward. Therefore, structuring those system aspects, where the handover of knowledge and artefacts between domain experts takes place, can be of value for both the robotic market and academia.

This chapter is hereinafter structured as follows. First, Section 3.1 discusses some common challenges of robotic software development. After that, Section 3.2 provides a selected overview of associated approaches and relates them to the contents of this dissertation. Section 3.3 discusses the role of MDSE in formalizing the core concepts of this dissertation, which are described in the three successive core chapters.

3.1. Common Challenges in Robotic Software Development

In general, software development in robotics is associated with various challenges such as rising software complexity, expertise partitioned between different specialists, and the need to design robust robotic software systems considering constrained resources (see left in Figure 3.2). On the one hand, solving these challenges opens up new application domains. On the other hand, the application domains themselves impose specific needs on the overall software development such as the aspiration for multipurpose robots and natural responsiveness, as well as the required degree of robustness, reliability, dependability, and autonomy.

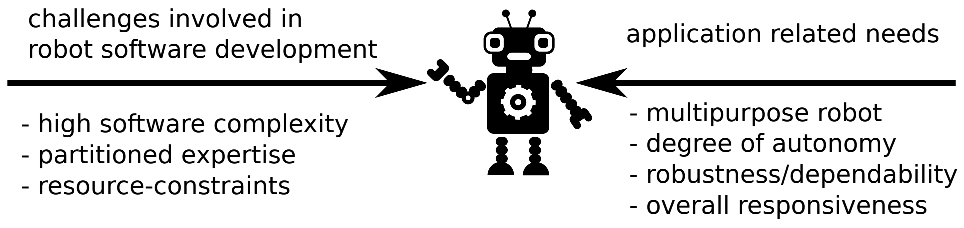

challenges involved in robot software development
- high software complexity
- partitioned expertise
- resource-constraints
application related needs
- multipurpose robot
- degree of autonomy
- robustness/dependability
- overall responsiveness

Figure 3.2.: Challenges of overall software development meet application-related needs

The domain of service robotics is a highly interdisciplinary research field involving many common computer science disciplines. As illustrated in Figure 3.3, the domain of service robotics can be further divided into three sub-domains, namely the robotic technology domains, the robotic application domains, and the robotic environments (which can be considered a domain in itself). While robotic technology domains have experts for individual robot capabilities such as navigation, mobile manipulation, object recognition, and human–robot interaction, the robotic application domains have experts in non-robotic domains such as agriculture, logistics, health-care, smart factories, and more.

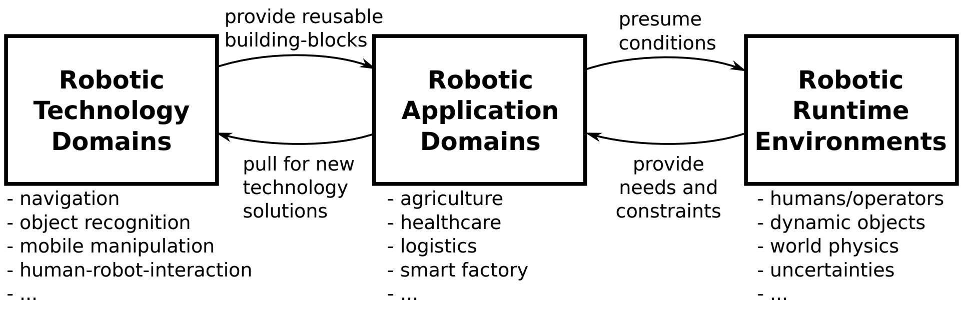


Figure 3.3.: Three common domain-clusters in robotic software development

There are push and pull relationships between the robotic technology domains and the robotic application domains. While individual robotic technologies and algorithms for robots still have a lot of potential to be further improved, the core robotic challenge is more related to the integration and combination of these algorithms into coherent and consistent robotic software systems. This integration challenge comes along with the necessity to carefully understand application-related needs of actual customers and the target environments where the robots are supposed to operate.

This section summarizes some common challenges involved in the design and development of robotic software systems. One of these challenges is related to mastering the overall software complexity. Therefore, the next two subsections analyze some common sources of software complexity and derive some general guiding principles to be followed. Thereafter, the two follow-up subsections discuss problems related to bounded rationality, partitioned expertise, and non-functional, application-specific needs.

3.1.1. Problem complexity in robotic application domains

One common reason for applying systematic software engineering methods in robotics is the need to master the challenge of an ever-increasing software complexity. However, before this challenge can be mastered, it is important to understand where software complexity comes from. One distinct cause is the diversity of robotic application domains with their inherent requirements for the robot's functionality and the anticipated environments where the robot is supposed to operate. Natural human-centric environments dictate certain constraints for the robot's shape, size, and mass, e.g. due to typical sizes of doors and general safety considerations. An application also defines the missions the robot should accomplish and thus the potential actions to execute. Executing actions means expending energy, which is limited due to the limited battery capacity of a robot and the amount of time the robot should last between charges. Additionally, diverse actions differently drain the robot's batteries. Altogether, the robotic application domain defines the overall (inherent) *problem complexity*.

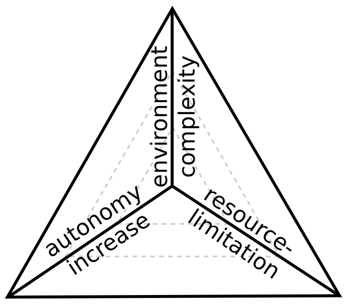


Figure 3.4.: Software complexity sources

Figure 3.4 shows three common sources of problem complexity that can individually (as well as in combination) impact the overall software complexity. First, environmental complexity has a direct impact on the way software needs to be built, because the more dynamic and unpredictable the environment, the more flexible the software needs to be in terms of reacting to changing situations. Second, the degree of autonomy also affects software complexity since high autonomy again requires adaptable and robust overall system behavior. Although resource limitations might not be as obvious at first glance, they have a major impact on the robot design, because systems with constrained resources need to become aware of and to dynamically manage their own resources. Combining these three sources makes it even worse. For instance, both environment complexity and a high degree of autonomy might impose high resource demands, which, if limited, make it more difficult to properly design the overall system. Moreover, complex environments lead to high efforts for making the robot autonomous, again provoking high resource demands. Finally, high autonomy means coping with many different exceptional situations, whose number increases in complex environments and which make it more difficult to find the right trade-off for acquiring the right resources. It is worth noting that, while the general relationship between these sources can be verbally described, to the author's best knowledge and despite some metrics such as lines-of-code, there is no direct way to combine the numbers from these three sources to calculate an objective value of software complexity.

Besides these three sources, there is another factor that severely impacts software complexity, namely multipurpose robots. On the one hand, single-purpose robots are easy to build, as their software only needs to serve one purpose only, leading to rather linear and repetitive executions. On the other hand, multipurpose robots typically need to be able to accomplish different missions in any combination and sometimes even in parallel. Obviously, this considerably adds to the above-discussed complexity.

In summary, while it is good to be aware of the inherent problem complexity, ultimately, realistic environments and real autonomy simply involve a certain unchangeable problem complexity (in contrast to design complexity, which is described next). Consequently, the overall development process must support a direct and easy way of incorporating application-specific requirements, rather than artificially trying to reduce this kind of complexity.

3.1.2. Design complexity in robotic software development

The design and development of robotic systems can be in itself a source of software complexity. Robot software involves many algorithms that need to exchange data and to be executed in a coordinated way. Complexity in this sense comes from interweaving these algorithms [Hic11]. Highly tangled algorithms cannot be considered and analyzed independently, making it cumbersome to integrate them into coherent and consistent robotic systems. Moreover, hidden assumptions between interacting algorithms easily lead to software bugs that are difficult to trace.

Another common problem results from the mingling of responsibilities and concerns. Software parts that are built for multiple purposes soon result in ambiguous semantics and implementations that need to handle many overlapping concerns. Again, this is highly error-prone and leads to arguably avoidable complexity. The opposite of complexity is simplicity, which, in this sense, means having a single purpose,[1] being considered independently (i.e., not interwoven with other parts), and having clear functionality and clear semantics. Consequently, a universal guiding rule is that software parts serving the same purpose and addressing a certain concern should be grouped and kept together, while unrelated parts should be separated and their coupling loosened

[1]Single-purpose software is not in conflict with multipurpose robots. Multipurpose robots can be composed of single-purpose software parts.

and simplified, but only up to a degree so that the relevant abstraction remains useful (i.e., still detailed enough) for the involved developer role.

While the *problem complexity* cannot be changed (as argued above), the *design complexity* can and should be reduced wherever possible. Therefore, a general objective of this dissertation is to focus on the overall reduction of design complexity using model-driven engineering methods and component-based software development techniques (see Objective 1.3).

3.1.3. Bounded rationality and partitioned expertise

Neither a robot nor any developer can predict the future. Nor is it possible to fully grasp the current global situation. Robots can try to improve the accuracy of the sensed world's properties by spending more processing resources. However, while the robot reasons about possibilities, the world continues to change and actions executed too late might lead to different, and possibly unexpected, outcomes. Therefore, the amount of deliberation time a robot should spend calculating a result should be traded off for the risks of actions that are too late [Zil93] and for the availability of resources. For cases that can be anticipated in advance, this trade-off is part of the overall robot design. However, there are also cases which cannot be efficiently predicted in advance, e.g. due to combinatorial explosion of alternatives. For these cases, the robot itself needs to be able to robustly and flexibly react to changing situations and contingencies. As a result, design-time variability and runtime adaptability must be mutually supported by a potential robotic software development process.

Furthermore, it is no longer possible—nor is it economical—to design and develop robotic software systems from scratch over and over again by a few omniscient robotic experts. Instead, realistic robotic software systems combine knowledge and integrate results from various experts in highly specialized fields. It is significant that these experts can focus on their individual field of expertise and can work on isolated sub-problems, solve them individually, and then systematically integrate and combine their results into a full robotic software system. The overall software complexity might remain the same or even be increased due to additional interfaces and abstraction layers. However, as long as the specialized developers are able to efficiently solve their individual software parts (i.e., individual complexity is low) and it is clear how these software parts are combined together as well as nobody ever needs to understand all the details of all software parts at once, the overall complexity does not count and the overall development process becomes manageable.

3.1.4. Non-functional, application-specific needs in robotic software development

As argued at the start of this chapter, one of the core differences between lab prototypes and products is the shifting focus from "purely" functional needs to non-functional, quality aspects of a system. For instance, non-functional system aspects related to reliability and dependability are critical factors for product development. Interestingly, the general difference between functional and non-functional aspects is not very clear. In the literature, the difference is often explained by the question about “what” a system is able (or should be able) to do (i.e., functional aspects) versus “how (well)” a system does it (i.e., non-functional, quality aspects). But then again, everything a system does is part of its overall function. Designing an improved (i.e., “better”) function means changing the function itself. Moreover, it is often difficult to pin-point the exact software parts that are responsible for certain quality aspects.

The difference between functional and non-functional system aspects can be analyzed from a slightly different angle. Individual software components tend to focus mainly on functionality, whereas system integration tries to satisfy application-specific (non-functional) needs after the right components, which provide the required functional features, have been selected. In other words, functional aspects often relate to reusable, generic software blocks (i.e., software components), while non-functional aspects are often intrinsically tied to the actual application-specific needs and requests of the involved customers. To make these non-functional aspects explicit—and more importantly manageable—it is essential to provide a clear grounding into the actual system parameters such as the end-to-end latencies and jitter of the data-flow chains in a system. While this certainly is a small part of the overall quality discussion only, it is particularly important for the design and development of dependable and reliable robotic software systems (as explained in more detail in the next section).

3.2. Use Case-driven State-of-the-Art Analysis

Structuring and formalizing the overall robotic development process has been a struggle for many years now, and many partial solutions have been proposed and presented within the scientific robotic community. Some approaches have become popular and are widely used (such as the Robot Operating System (ROS) [Qui+09]), while others have been forgotten (such as the Player/Stage Project [GVH03]). And still, it is hard to develop robust and reliable robotic products.

This section pursues a pragmatic approach based on a common use case for analyzing the involved problems, discussing common solutions (wherever available), and identifying scientific gaps to be solved. Therefore, Section 3.2.1 presents a navigation scenario with the focus on emphasizing the involved architectural design assumptions and decisions. Section 3.2.2 selectively presents some common best practices in building and engineering robotic software systems. After that, Section 3.2.3 presents common partitioning schemes that help in structuring the overall development workflow. Section 3.2.4 discusses some concrete structural needs at the component as well as the system level. Finally, Section 3.2.5 highlights the different QoS aspects from an overall application perspective.

3.2.1. The navigation use case

Powerful service robots rely on a range of basic functionalities to carry out anticipated tasks. While some functionalities such as the ability to interact with humans via speech can be considered as auxiliary amplification of the robot's basic functions, some other functionalities such as the ability to navigate in changing environments are fundamentally essential for any mobile service robot. This section thus chooses a navigation use case (earlier introduced in the core publication [Lot+15]) as one of the ever-recurring use cases in robotic software development with the emphasis on discussing the involved architectural design decisions.

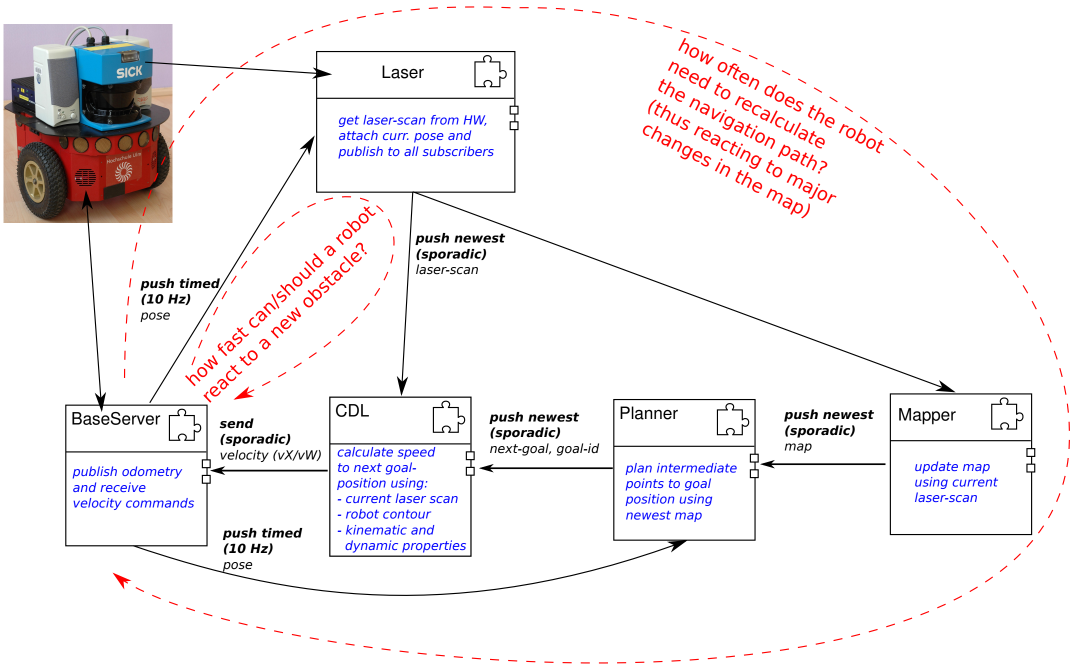

Laser
get laser-scan from HW, attach curr. pose and publish to all subscribers
how often does the robot need to recalculate the navigation path? (thus reacting to major changes in the map)
push newest (sporadic) laser-scan
push timed (10 Hz) pose
how fast can/should a robot react to a new obstacle?
BaseServer
publish odometry and receive velocity commands
send (sporadic) velocity (vX/vW)
CDL
calculate speed to next goal-position using: - current laser scan - robot contour - kinematic and dynamic properties
push newest (sporadic) next-goal, goal-id
Planner
plan intermediate points to goal position using newest map
push newest (sporadic) map
Mapper
update map using current laser-scan
push timed (10 Hz) pose

Figure 3.5.: Schematic representation of the navigation scenario

Figure 3.5 schematically illustrates the navigation stack that has been used in many research projects and whose features and limitations are well understood. The navigation stack comprises five main components that exchange information. Without focusing on the exact definition of a software component for the moment, we may consider a component just as a functional block (i.e., a cluster of related functions). The two components *Laser* and *BaseServer* directly communicate with the involved hardware devices, which in our case are the SICK LMS 200 laser-range finder and the Pioneer P3DX base platform. The *Laser* component acts as a pure sensor that continuously scans the environment and periodically provides laser scans. The *BaseServer* acts as both a scanner (that periodically provides odometric updates) and an actuator (that receives navigation commands). A first architectural particularity of this navigation stack is that the *Laser* component requires odometry from the *BaseServer* as pose-stamps for specifying the robot's position where the current laser scan has been recorded. While this might appear unnecessary at first glance, because the robot's exact position could also be determined by transforming the laser scans into the robot frame, stamping the laser scans with the position directly is just another (and potentially more efficient) solution for the same problem.

The remaining three components implement different functional aspects of the overall navigation stack. The *CDL* component implements an extension of the classical dynamic window approach called Curvature Distance Lookup (CDL) [Sch98], which considers kinematic and dynamic constraints and additionally the robot's shape. The *Mapper* component accumulates occupancy grid maps based on incoming laser scan updates, and the *Planner* component implements a simple breadth-first-search algorithm for calculating the shortest path to the next goal.

Overall, the five navigation components implement two basic capabilities of the robot: (i) reactive obstacle-avoidance and (ii) grid map–based path planning. Each of the two capabilities involves a chain of interconnected components representing a sensor-to-actuator control loop, also referred to as a *cause–effect chain* [Lot+15] (illustrated as the two red arrows with a dashed line in Figure 3.5). The role of the inner loop is to ensure collision-free navigation, while the role of the outer loop is to plan intermediate goals (within the *Planner* component) toward the overall destination and to command the next intermediate goal to the *CDL* component. While the inner loop needs to be fast in reacting to sudden obstacles, the outer loop can be considerably slower in re-planning the path in times of rather rare and substantial environment changes (e.g. closed doors or blocked hallways).

The design of the fast-reactive navigation loop might involve the following question: What is the maximum admissible overall response time in reacting to obstacles that suddenly appear in front of the robot? This question can be answered from two opposing viewpoints. On the one hand, the admissible reaction time depends on further application-specific needs such as the desired maximum navigation velocity (i.e., the faster the robot moves, the lower the reaction time needs to be) and the anticipated suddenness of the dynamic obstacles (e.g. humans appearing in front of the robot from around a corner at a certain speed). By knowing these aspects and the robot's physical kinematic constraints, a certain maximum reaction time can be determined along with a certain safety distance the robot needs to maintain while avoiding obstacles. On the other hand, the cause–effect chain in itself imposes a certain end-to-end latency with a jitter as a result from scheduling of intermediate links in such a chain, sampling effects, and network arbitration. Deciding for an admissible reaction time therefore requires the knowledge of application domain experts and the ability to influence the overall system configuration by adjusting components' internal *computation* characteristics (e.g. the periods at which individual components have to run) and adjusting *communication* characteristics between the involved components (e.g. whether data is communicated synchronously or asynchronously).

Moreover, analyzing the obstacle-avoidance loop and deciding on the right configurations for the involved components are not enough. Instead, it is common that several control loops (at least partially) involve the very same components. For instance, both the obstacle-avoidance and the path-planning loops use the *Laser*, *CDL*, and *BaseServer* components (see Figure 3.5). Similar to the obstacle-avoidance loop, the path-planning loop also depends on further application-specific needs such as the anticipated frequency of major map changes. Updating the current grid map and re-planning the path to the destination might be more expensive than the pure reactive obstacle avoidance. Therefore, deciding on an admissible update cycle of the path-planning loop means going for a trade-off between estimating a common frequency of substantial environment changes and minimizing calculation costs. Hence, the robot might use a suboptimal path for a short period of time before the map is updated and a potentially shorter path is selected. Configuring the three components of the obstacle-avoidance loop in such a way that their overall reaction time becomes lower might allow for more configuration flexibility for the *Mapper* and the *Planner* components, or respectively restrict the configuration options otherwise.

In summary, the following conclusions can be drawn. First, end-to-end latencies of chains of components need to be known (i.e., easy to determine) to enable application domain experts to reason about the overall application-specific system behavior. Second, computation characteristics within components and communication characteristics between components need to be easily modifiable by application domain experts at the system level. This allows directly applying application-specific requirements in a system composed of reusable and flexible building blocks.

3.2.2. Best practices in robotic software and system engineering

Over the last few decades, the robotic research community has been mostly focusing on improving isolated capabilities and algorithms for robotic systems. However, as mentioned in Section 3.1, software complexity is growing (presumably more exponentially than linearly) along with a rising number of isolated capabilities and the increasing performance of ever more miniaturized embedded hardware. This has recently shifted the scientific focus toward advanced software engineering methods for systematically designing and building robotic software systems.

The domain of service robotics provides a fairly long history of different software architectures, starting with a classical *sense-plan-act* [Nil80] paradigm, over the *subsumption architecture* [Bro86], up to several multi-layered architectures more or less related to the well-known *three-tier (3T) architecture* [Bon+97]. Kortenkamp states that *"the goal of an architecture is to make programming a robot easier, safer and more flexible"* [KS08b]. While this is true, there is ultimately no such thing as a single, generic reference architecture that equally goes with all robotic scenarios. Instead, an architecture is highly influenced by various objectives such as the desired level of modularity and reusability, the independence of hardware and software platforms (i.e., the operating system, the programming language, the communication middleware, the robotic framework, etc.), and, last but not least, the desired level of autonomy (i.e., the runtime adaptability). Although this dissertation nonetheless has an underlying architecture similar to 3T (or more specific similar to ATLANTIS [Gat92]), the focus is more on other engineering methods as described in the following.

As illustrated in the preceding Section 3.2.1, robotic software systems consist of a range of basic capabilities and thus lead to a natural modularity. Modularity under the term CBSE [HC01] has been one of the most widely researched and adopted software engineering principles in the robotic domain over the last decade for effectively coping with the overall software complexity. However, due to the huge diversity of robotic applications, it is unlikely to find one single approach that serves all the different engineering needs equally well. Nevertheless, there has been a big struggle to find that one generic CBSE solution for robotic software development. One recent example is the harmonization initiative driven by the BRICS project[2]. This initiative inevitably resulted in an oversimplified BRICS Component Model (BCM) [Bru+13], which is so generic that it lacks even the most fundamental structures that are required to successfully support the overall development process (see next subsection for more details).

Sometimes structures are perceived as restrictions that limit the developer's freedom and creativity. However, structuring those system aspects that are pivotal for ensuring the compatibility of individually developed system parts so that later they can seamlessly fit together is a prerequisite for successful system integration. Besides, structuring the "right" system parts frees the individual developer from the burden of caring about those system aspects he/she is not responsible for (in agreement with Lee's *freedom-from-choice* [Lee10] idea). Of course, developers should still have enough design freedom for their own system parts in order to be able to ensure their efficient implementations. Finding an adequate trade-off is a focus maintained in this dissertation.

Divide and conquer is often considered the core principle behind CBSE. However, the main challenge lies not so much in dividing the system into isolated parts as in the subsequent integration (i.e., composition) of those parts back into several consistent and functional systems. Successful integration depends on a clear definition of communication and interaction mechanisms between individual software parts in a system. In line with the general ideas of Service-Oriented Architectures (SOAs), dependencies between individual software parts need to be reduced (i.e., loosely coupled components), thus increasing their flexibility and reusability with respect to other systems. The IEEE Standard Glossary [Boa90] offers the following definitions of common terminology:

> *"A **Component** is one of the parts that make up a system, while a **System** is a collection of components organized to accomplish a specific function or set of functions. **Integration** is the process of combining software components, hardware components, or both into an overall system. **Flexibility** is the ease with which a system or component can be modified for use in applications or environments other than those for which it was specifically designed."* IEEE Standard Glossary [Boa90]

While these definitions still are valid, recent advancements in the area of Model-Driven Software Engineering (MDSE) considerably ease the definition of required structures and abstractions, including the implementation of respective model-driven tools that ensure compliance with these structures.

3.2.3. Vertical (layered) and horizontal (workflow) decomposition

The preceding subsection discussed the need for structuring pivotal parts of a system as a general means for coping with the overall software complexity. The question now is as follows: Which

[2]BRICS project: `http://www.best-of-robotics.org`

system parts are particularly vulnerable and thus need to be structured? Answering this question requires the identification of relevant architectural principles that provide partitioning schemes for organizing different views of a robotic system [Lut+14]. One of such architectural principles is the *separation and support of different roles* involved in the development of an overall robotic software system. Each of these developer roles has certain responsibilities and duties with respect to contributing to the overall system. Different developer roles must not necessarily work at the same time and at the same place. In an optimal case, they even do not need to know about each other and can rely on the knowledge, structures, and models provided by others. One distinctive factor that differentiates a role from others is a unique perspective and certain *architectural view* [Cle+10] of the overall system at an adequate abstraction level, thus restricting the focus to relevant system aspects only. The identification of unique responsibilities for such roles is often referred to as *separation of concerns*. Back in 1974, Dijkstra had argued in [Dij82] that the efficiency of a scientific thought could be increased by focusing solely on one specific *concern* at a time. This *separation of concerns* has since been refined and reapplied for different aspects within the discipline of computer science. The domain of service robotics is no exception in this regard. For instance, Bruyninckx et al. [Bru+13] propose the separation of the following five concerns: *Computation*, *Communication*, *Configuration*, *Coordination*, and *Composition*. Each of these concerns can mean different things depending on the actual system aspect they refer to. In order to put these concerns into the right perspective and context, it is necessary to have a closer look at some common development phases and views of a system.

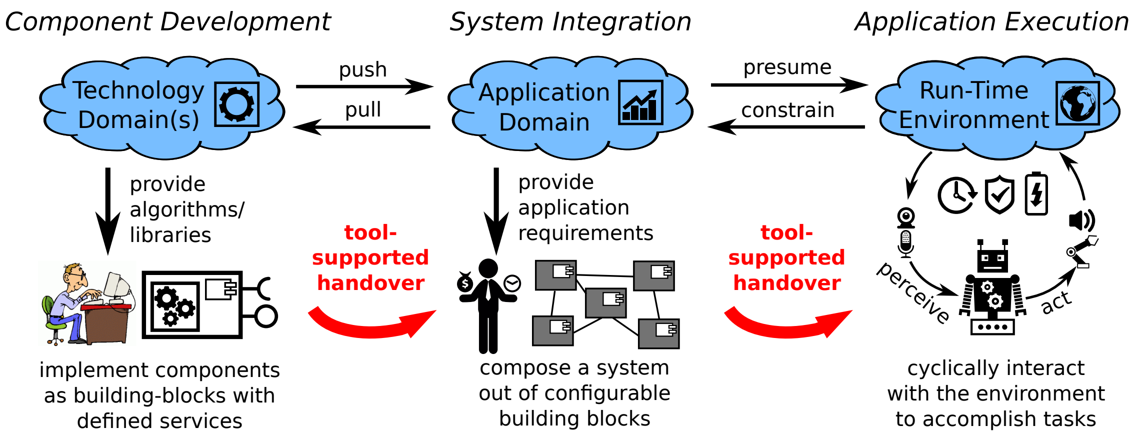

Component Development
System Integration
Application Execution
Technology Domain(s)
push
pull
Application Domain
presume
constrain
Run-Time Environment
provide algorithms/ libraries
tool-supported handover
provide application requirements
tool-supported handover
perceive
act
implement components as building-blocks with defined services
compose a system out of configurable building blocks
cyclically interact with the environment to accomplish tasks

Figure 3.6.: Required tools to support interfacing and handover of knowledge and artefacts between distinct development-workflow steps

Figure 3.6 illustrates the three most common development phases along with the involved developer roles. First of all, the sketched development workflow is neither intended to be all-encompassing nor statically ordered. For instance, it is possible to start defining a system out of already existing components and then to go back to component development for designing and implementing new (i.e., not yet available) components. Depending on the different roles, the distinctive factor for separating between different workflow phases is a different *architectural view* [Cle+10] of specific system aspects. For instance, a *component developer* (on the left in Figure 3.6) typically focuses exclusively on one particular component at a time without bothering about other components or target systems where this component might be used later. Similarly, a system integrator (in the middle of Figure 3.6) should focus on selecting the right components based on the application needs for the currently designed system using out-of-the-shelf components without having to understand the individual components' internal realizations nor being forced to modify their internal implementations. This is only possible if component developers are supported and guided to define all relevant aspects of a component such as the definition of the component's services that will become important composition factors in the later system integration phase. System integrators, on the other hand, need to be able to understand the *functional features* of the available components with their respective *quality attributes* and to adjust the components' parameters at a model level as part of the integration process. Model-driven tools can support these two developer roles in: (i) adjusting relevant aspects at the right abstraction level without risking system inconsistencies with respect to preceding and subsequent development phases, and (ii) passing the knowledge from one developer role and workflow phase to the next.

The separation of the two developer roles into respective development workflow phases might also be reasonable in many other software-intensive domains beyond robotics. By contrast, the third development workflow phase, shown on the right in Figure 3.6, is more distinct to the domain of service robotics. In general, the robotic software systems do not become static and fixed after the software has been deployed to the robot platform. Instead, the systems need to remain flexible and adaptable with respect to changing conditions and situations a robot might face during operation. While operating, a robot repeatedly interacts with its surrounding by continuously sensing the environment to determine relevant aspects of the current world-state, accordingly adapting its own strategy to accomplish a current mission and finally selecting and executing promising actions. For this to happen, the robot system requires further knowledge about available configuration options from the integration phase. Again, this handover of knowledge can be supported by relevant model-driven tooling.

The separation of roles and development phases is often referred to as *horizontal decomposition*. By contrast, *vertical decomposition* is often referred to as the layered decomposition with several abstraction layers, each clustering coherent concerns. A good example of such layered decomposition is the *OSI communication stack*[3]. Each layer addresses a distinct purpose at a certain abstraction level and builds on top of the respective lower layer. In robotics, such layers do exist, albeit not always explicit. Due to the separation of roles and concerns, the individual layers must not be solved entirely at one particular step in the overall development workflow but are refined in several successive steps. In other words, each developer role adds details at certain layers that are further refined by downstream developers.

[3]OSI model: `https://en.wikipedia.org/wiki/OSI_model`

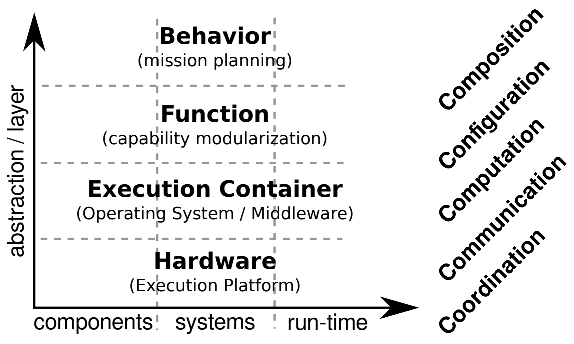


Figure 3.7.: Vertical decomposition by defining appropriate abstraction layers

Figure 3.7 illustrates different layers (vertical axis), the development phases (horizontal axis), and the different concerns (terminology borrowed from [Bru+13]) that can be found in different combinations in each of the layers and phases. For instance, *computation* on the hardware layer can refer to the processor load or the scheduling strategy on the execution container layer. The functional layer might require some *computational* resources and the behavior layer might require a certain response time with respect to *computation*. Moreover, component development might be mainly related to *composition* on the functional layer, or additionally also involve the behavior modeling of component-related behavior parts (if needed). It is clear that thinking about the different combinations of layers and phases eventually creates the required context for the individual concerns. Simply put, this is the thinking underlying the overall meta-model design process in the method chapters later in this dissertation. However, a general discussion of all the combinations (along with the implications) here would go beyond the scope of this dissertation (yet it will be very helpful in discussing the concrete problems in the subsequent core chapters).

The general focus of this dissertation is on the middle two layers, namely *function* and *execution container*. While the top layer *behavior* is not directly in focus, there is still an interesting link that needs to be explained. Several papers such as [Fir89; SS10; SS14; KB12; BC10] (just to name a few) specifically address this top layer. The link to this dissertation is that this top layer in any case needs to rely on the deterministic and reliable execution of all the lower layers. An interesting observation regarding this top layer is that it might require switching between different consistent system configuration options at runtime for the lower layer(s). While the general *behavior* coordination is beyond the scope of this dissertation, the design and development of consistent system configuration sets for selected aspects of a system are in focus here (as discussed in more detail in the successive sections). Finally, the *hardware layer* on the bottom provides an important foundation for the higher layers. In this dissertation, the hardware layer is considered given (i.e., solved) by the typical robotic platforms, which use common mobile Central Processing Units (CPUs) (such as the Intel's DualCore CPU), standard architectures (such as x86) and the widely applied communication stacks (such as TCP/IP).

To sum up, several partitioning schemes need to be considered in combination in order to properly identify system aspects that motivate specific structures and where tool support is required to increase the overall development efficiency, to automate knowledge handover between related developer roles and workflow phases, and to ensure the overall system consistency throughout the entire development life cycle of a robotic software system.

3.2.4. Relationship between components' internal processing and inter-component communication

Section 3.2.1 introduced an example use case with two main control loops (i.e., *cause–effect chains*) that depend on specific overall end-to-end latency and jitter specifications of messages starting from sensors, traversing intermediate component links, and reaching the actuators. In order to better understand the causes that affect the overall latency and jitter along a *cause–effect chain*, it is necessary to investigate further details of common structures within individual components, as well as common communication characteristics between several components.

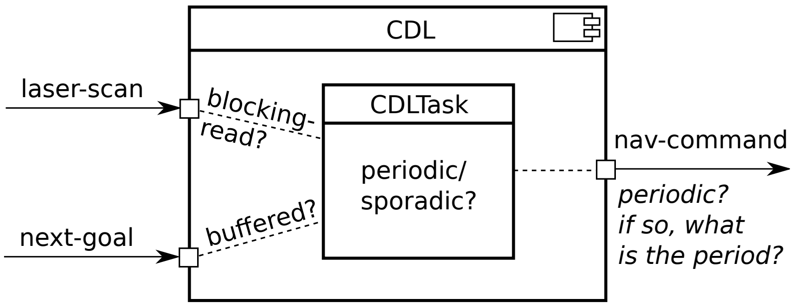


Figure 3.8.: CDL component schematic example

Figure 3.8 schematically illustrates some of the core structural elements of the *CDL* component (from Section 3.2.1). The obstacle-avoidance functionality of the *CDL* component comprises three steps. First, the *CDL* component receives the latest laser scan (through the respective in-port), which includes the odometry value of the robot. Based on this laser scan and the intrinsic kinematic constraints of the robot, all admissible navigation curvatures are determined that the robot could drive next (see [Sch98] for more details). Second, all those curvatures are filtered out of this list, which would lead to a collision with any obstacle visible in the current laser scan. Third, from all the remaining collision-free curvatures, the one that is selected leads the robot closest-possible to the next intermediate goal position (which is received through the other in-port named *next-goal* in Figure 3.8).

The obstacle-avoidance functionality is executed within the *CDLTask*. While the basic functionality is quite clear, the actual runtime execution behavior leaves a couple of open questions. To begin with: What triggers a new update cycle of the *CDLTask*? Is it maybe an internal periodic timer, or does the task wait on the arrival of new messages on one of the two (or both) input ports? From the functional point of view, it only makes sense to calculate new navigation commands **if new** laser scans are available. Calculating a navigation command based on an **old** (i.e., already processed) laser scan would yield the same result without considering that the robot has moved forward in the meantime and thus would be a waste of resources. Therefore, it might make sense to suspend the *CDLTask* until a new laser scan becomes available. However, this statically binds the update frequency of the *CDLTask* with the update frequency of the incoming laser scans. This leads to the following two problems. First, a uniform update frequency for the *CDLTask* alone cannot be generally specified without considering what is actually required in a target system (which might deviate from one system to another). Second, binding the update frequency with the input frequency of one of the input ports would degrade the flexibility of the *CDL* component to be used in combination with other components that provide the required input data.

For instance, some components might provide messages (e.g. laser scans) with an update frequency that is too high for the *CDLTask* to follow, because each cycle of the *CDLTask* takes a certain period of time to complete. Furthermore, each real robot has a certain mass leading to a certain inertia, thus resulting in absolute physical constraints. Calculating avoidance commands more frequently than the robot could physically execute does not make much sense and thus would be again a waste of calculation resources. Therefore, it might be useful to skip intermediate laser scans if they arrive at an extremely high input frequency and to use the available latest laser scan update.

Besides, what about the frequency of the incoming next-goal messages? It is safe to assume that the next-goal messages arrive at a much slower update frequency than the laser scans (regardless of which components exactly will be later used in the system). Moreover, if the next-goal updates would arrive at a higher update frequency than the laser scans, it would not make much sense to calculate new navigation commands blindly (i.e., without a new laser scan). However, calculating a new navigation command using a **new** laser scan but an **old** next-goal value still makes sense, because the robot might have to face a dynamic obstacle while approaching the next intermediate goal position. Therefore, it is reasonable to store the latest next-goal message in a buffer of size one that is overwritten each time a new next-goal update comes in, thus using the currently available next-goal value in each cycle of the *CDLTask*.

This rather simple example already involves a lot of reasoning and requires making several assumptions about the target system to properly define the synchronicity between input ports and the task, as well as the actual update frequency of the task. Selecting a certain update frequency for a task directly impacts the assigned out-port. For example, the update frequency of the *CDLTask* directly impacts the update frequency of *nav-command* (see Figure 3.8). Nevertheless, even after identifying all functional constraints, a definite update rate for the *CDLTask* cannot be universally selected for all (not yet known) target systems.

This altogether motivates a different approach. The activation source for the task's update cycles must remain a configurable variation point for the later system integration, where the then available knowledge about the system and application-related constraints help to determine the right configurations. However, it should still be possible to define some functional boundary conditions for tasks and thus to reflect implementation-specific constraints (see the next Chapter 4 for more details).

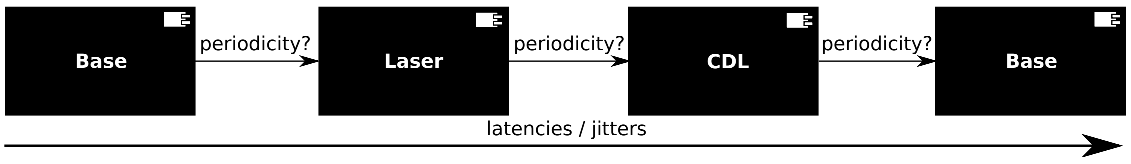


Figure 3.9.: Base-Laser-CDL-Base component chain

So far, the *CDL* component has been analyzed in isolation. The next interesting questions are as follows: How should this component behave in a *cause–effect chain* together with other components, and how can the overall latency and jitter of this *cause–effect chain* be determined? As discussed in the preceding Section 3.2.3, all relevant configuration options of individual components should be lifted to the model level in order to free the system integrator from the burden of investigating the components' internal implementation details. Figure 3.9 illustrates the obstacle avoidance component loop (from the navigation example in Section 3.2.1). Ideally, all the internal details of the components would be hidden at the system level, thus allowing the composition of a system out of black-box components. However, as argued above, some configuration options such as the selection of an adequate update frequency for a task in a component need to remain an open variation point until the system integration phase. Moreover, the individual links between the components' tasks and input ports need to be further refined so that application-specific latency and jitter values of the *cause–effect chains* are achieved. On the one hand, selecting synchronous links for the entire *cause–effect chain* will minimize the overall latency, but it might increase the jitter (because all the execution-time fluctuations of the intermediate tasks directly accumulate to the overall jitter). On the other hand, selecting asynchronous links for the components of the *cause–effect chain* will potentially stabilize the overall jitter at the expense of the potentially greater overall latency (because of the intermediate sampling effects).

In real-world robotic systems, *cause–effect chains* seldom appear in complete isolation (i.e., independent of each other). Instead, realistic robotic systems typically consist of several sensor-to-actuator couplings with many interconnected components in between, forming complex graphs with branches, forks, and loops. Selecting loop-free information-flows called *cause–effect chains* therefore makes sense. However, consequently, individual *cause–effect chains* are not necessarily independent because they might share the same components (i.e., branching or forking components). This makes hybrid *cause–effect chains* very common, consisting of a combination of synchronous and asynchronous links in between. For example, *CDL* is one of such branching components. It not only continuously receives new laser scan updates but also considers new navigation goals. Selecting the right update frequency for the *CDLTask* thus requires consideration of both *cause–effect chains*, namely local obstacle avoidance and map-based path planning, in combination.

3.2.5. Trading-off QoS requirements

The preceding two Sections 3.2.1 and 3.2.4 have introduced the notion of *cause–effect chains*. Thus far, the focus has been on rather technical aspects such as the overall latency and jitter of *cause–effect chains*. This section will focus more on the application-specific aspects such as predictability, deterministic execution, and execution quality (i.e., quality of service) in general.

	continuous (periodic)	**event-driven** (sporadic/on demand)
hard real-time	balancing robot	emergency stop
soft real-time	person following	humans detector
relaxed	map updating	speech interaction

Table 3.1.: Characterization of different timing requirements for typical capabilities of a robot

Multipurpose service robots need to combine lots of different capabilities implemented as software components. These components are part of different *cause–effect chains* with specific demands with respect to their individual responsiveness. For example, a robot that needs to balance on two wheels (at all times) requires a close coupling between the position sensor (e.g. an inertial measurement unit (IMU)) and the actuator (e.g. the wheel motors). Violating the response-time between the sensor and the actuator will likely result in the robot falling over and needs to be avoided at all costs. Deterministic execution behavior is a top priority here. Components of such a *cause–effect chain* are best implemented either directly in hardware using dedicated microcontrollers with predictable execution characteristics, or in software using real-time tasks and real-time communication.

In addition to such safety-critical system parts, other system parts might involve less strict timing demands (i.e., soft real-time). For instance, a speech-interaction capability does not need to respond particularly quickly in a dialog with a human. Longer response times (in rare cases) could lead to tolerable inconvenience with a fairly soft threshold. Therefore, applying hard real-time guarantees for the entire software system is neither practical nor necessary. Instead, realistic robotic systems are heterogeneous in nature with isolated safety-critical parts and other parts executed in parallel, only loosely coupled together in order not to interfere with (or violate) the hard real-time guarantees. Table 3.1 provides some examples with required timing demands (vertical dimension) and regularity in execution (horizontal dimension). These examples lead to the following conclusions:

- For the *hard real-time* cases (irrespective of whether *continuous* or *event-driven*), a classical worst-case real-time scheduling analysis with partitioned network bandwidth is appropriate for guaranteeing that in all cases the robot is able to react within a certain amount of time.

- For the *soft real-time*, *event-driven* cases, it is sufficient to approximate a rough latency and the jitter between a sensor and the actuator considering all the intermediate components with their individual delays. It is also possible to derive appropriate timeouts, allowing the implementation of backup strategies for cases where timings are violated to improve overall robustness.

- For the *soft real-time*, *continuous* cases, it is additionally necessary to design the update rates of the individual components and of the whole chain of interconnected components (i.e., overall responsiveness) between a sensor and the actuator.
- For all the *relaxed* cases, it is often sufficient to empirically determine the typical case response times and to accordingly provide timeout strategies that allow preventing infinite deliberation times of a robot.

To sum up, appropriate *service quality* means that the robot needs to interact sufficiently quickly with humans and the environment. Hence, the appropriateness depends on the current situation and mission. The environments can be either more static or more dynamic in nature. Situations change over time (e.g. empty areas become crowded with people during lunchtime) and different missions require different timing guarantees. Consequently, responsiveness results from specific system aspects, such as end-to-end latencies and jitter in chains of interconnected components, and requires application-specific knowledge in order to be specified appropriately. The important point addressed in this dissertation is that such system aspects become manageable in the design and development of a robot and that they later can be traced and controlled during robot operation.

3.3. The Overall MDSE-Based Approach and Method Overview

So far, this overall Chapter 3 has focused on the analysis of the general problem domain, the derivation of common guiding rules to be followed, and a selective discussion of related approaches. Hereinafter, the focus will shift toward the conceptualization, formalization, and realization of structures and abstractions that adhere to the general needs discussed so far and that address the identified scientific gaps. The overall concepts and methods of this dissertation are divided into three parts, which are discussed in detail in the successive three core Chapters 4 to 6. The follow-up Chapter 7 presents and analyzes a concrete robotic example to assess the usability of the developed modeling tools.

Before delving into the overall concept and realization, the follow-up Section 3.3.1 presents a formalization method for the definition and specification of the concepts and relevant meta-models in the successive core Chapters 4 to 6. Section 3.3.2 provides an overview of the MDSE tools used in this dissertation and Section 3.3.3 presents an overview of the designed meta-models.

3.3.1. The central role of MDSE in this dissertation

In recent years, Model-Driven Software Engineering (MDSE) has gained a lot of attention within the general software engineering community for coping with the ever-increasing software complexity by means of structuring and formalizing the overall software development processes [BCW12]. The domain of service robotics undergoes the same trend as can be observed by recent approaches such as the BCM [Bru+13], RobotML [Dho+12], and SMARTSOFT [Sch+15]. A recent user study published in [Sta+16] confirms the usefulness of model-driven approaches for robotic software development in realistic projects. However, there is still a lot of scope for improvement with respect to non-functional, application-specific needs. This dissertation directly adds to this plethora of modeling solutions by a novel Domain-Specific Language (DSL) that provides an additional, consistent view of the system with respect to modeling end-to-end guarantees. The design and development of useful DSLs (as one central aspect of MDSE in general) is a challenging task that requires careful consideration of several basic MDSE principles related to coherency, consistency, simplicity, and comprehensibility of languages. From now on, this section will discuss some of these principles with respect to their relevance for this dissertation.

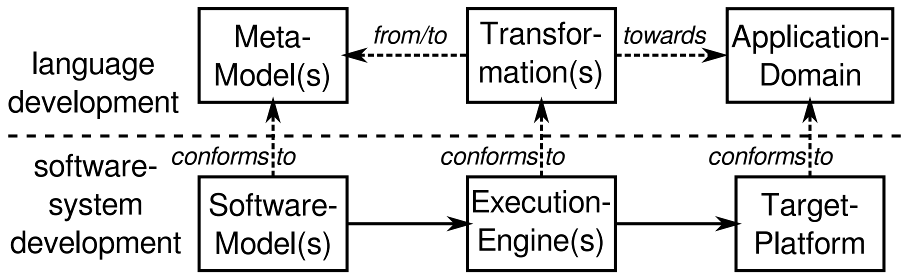


Figure 3.10.: Common aspects of MDSE (terminology is borrowed from [Voe13])

Figure 3.10 illustrates the general relationship between the development of languages and that of software systems using these languages. The main idea behind MDSE is that *software models* must drive the overall design, development, and evolution of software. This means that *software models* have to be (or to become) first-class citizens that directly influence pivotal aspects of an overall software system. This is also referred to as *prescriptive models* [Voe13; BCW12] (in contrast to *descriptive models*). Moreover, an *execution-engine* either directly interprets the *software models* in the context of the *target platform* or generates code for the *target platform* in a model-to-text transformation step.

Language development, on the contrary, is related to specifying *meta-models* and *model transformations*. A language implies a specific *view* [Cle+10] of a system under development with a certain abstraction level and aligned vocabulary (both specified in a *meta-model*) in the context of an *application-domain*. Moreover, useful languages are not separated but are interlinked either by using model references (specified at the *meta-model* level) or by specifying *model transfor-*

mations. Abstraction (e.g. of a *meta-model*) is a widely used principle in computer science (and in many other disciplines) that involves *"a cognitive process to create a mental representation of the reality"* [BCW12]. In this sense, creating and using languages and models is a natural process to cope with the vast complexity of software.

In the context of MDSE, two types of languages are distinguished: General-Purpose Languages (GPLs)[4] and DSLs. Both types allow modeling different aspects of a system. However, the former focuses more on flexibility with respect to the addressed problem domain, whereas the latter makes simplicity and clarity a top priority with respect to the modeling domain. Consequently, GPLs tend to become all-encompassing *lingua franca* (i.e., mixed languages with overlapping concerns), while DSLs often are combined into a family of several narrowed and interlinked languages, also called a *Modeling Language Suite* [Cle+10]. This dissertation prefers DSLs to GPLs due to the rather clear and specified problem domain, as well as due to the discussed need to separate concerns of the involved developer roles.

3.3.2. The main MDSE tools used in this dissertation

Recent advancements in MDSE tooling, most notably the Eclipse Modeling Framework (EMF) (with Ecore in its core), allow a simple and efficient specification and formalization of domain-specific concepts. Moreover, Eclipse provides a wide range of integrated tools (based on EMF) related to the design of graphical and textual notations, the definition of model transformations, and the management of several interlinked models. EMF is thus very attractive to be used as the main tool and the main notation for specifying the concepts and structures in this dissertation. More precisely, Ecore enables the specification of the *abstract syntax* of a language independently of any concrete textual or graphical syntax. Moreover, concepts specified with Ecore are independent of any particular execution platform (similar to e.g. the UML). In fact, Ecore is generally comparable to Essential Meta-Object Facility (EMOF)—a subset of MOF [Mof], which is basically the meta-model of UML. Unlike UML (and EMOF), the mapping from Ecore into a Java-based realization is absolutely clear and even fully automated in EMF. It is worth noting that while Ecore is the de facto standard for specifying meta-models, there are other valuable approaches, most notably JetBrains MPS[5], which can be used instead of EMF. Moreover, Ecore is particularly strong when it comes to specifying *static* structures (i.e., vocabularies and relations). Yet, structures alone are not sufficient to define all aspects of a language. Additionally, further semantic rules need to be specified, which define more *dynamic* behaviors of models. Therefore, model-checks, which directly extend the semantics of the Ecore-based meta-models, can be implemented. Again, there are several languages for this purpose, most notably the Object Constraint Language (OCL). This dissertation uses Xtend2 instead of OCL due to its better integration into the Eclipse ecosystem (and for other reasons explained in the next Chapter 4).

Figure 3.11 provides an overview of the main Eclipse Modeling Tools (EMT) used in this dissertation. At the top are the Ecore-based meta-models that specify the required structures and ab-

[4]Sometimes also called General-Purpose (Modeling) Language (GPML)
[5]`https://www.jetbrains.com/mps/`

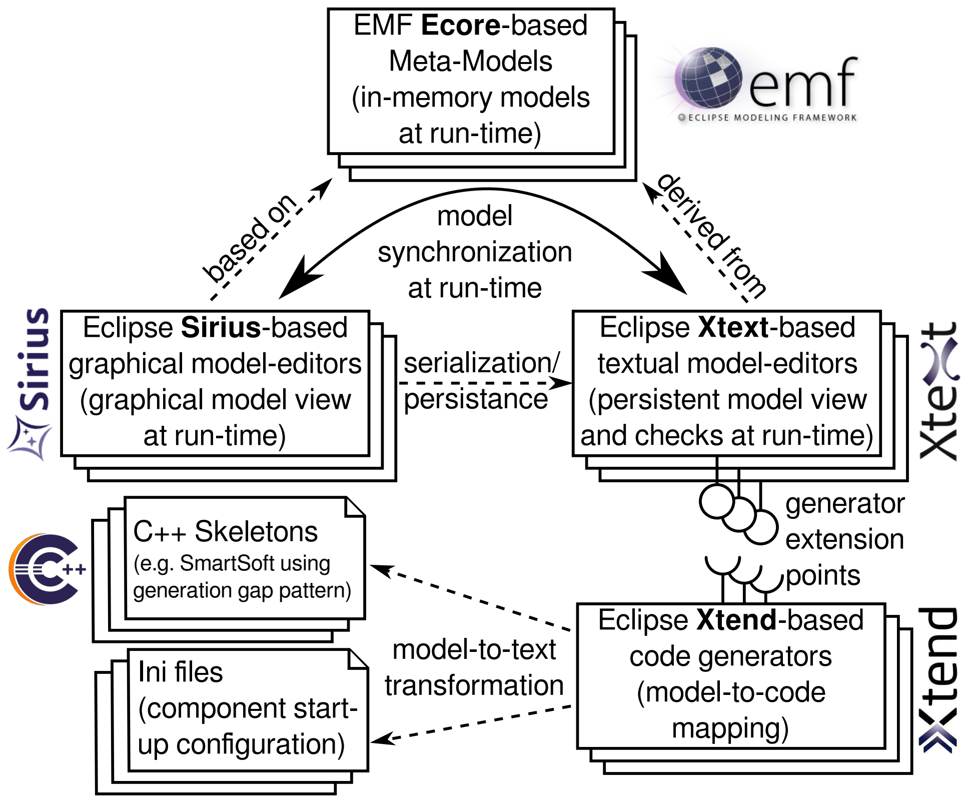

EMF **Ecore**-based Meta-Models (in-memory models at run-time)
emf
ECLIPSE MODELING FRAMEWORK
based on
derived from
model synchronization at run-time
Sirius
Eclipse **Sirius**-based graphical model-editors (graphical model view at run-time)
serialization/ persistance
Eclipse **Xtext**-based textual model-editors (persistent model view and checks at run-time)
Xtext
generator extension points
C++ Skeletons (e.g. SmartSoft using generation gap pattern)
Ini files (component start-up configuration)
model-to-text transformation
Eclipse **Xtend**-based code generators (model-to-code mapping)
Xtend

Figure 3.11.: The main Eclipse modeling tools used in this dissertation

stractions of different aspects of a system in different phases of a robotic development workflow. In conformity with these meta-models, textual grammars (based on Eclipse Xtext) and graphical notations (based on Eclipse Sirius) are derived. The default serializers from the Xtext grammars are used to make the models persistent. In addition, graphical model editors, based on Eclipse Sirius, allow a graphical specification of models using an intuitive graphical notation for each meta-model element with appropriate tool palettes to ease the creation of models. Both model representations (graphical and textual) are automatically synchronized in Eclipse based on their shared, Ecore-based, in-memory representation called Abstract Syntax Tree (AST). Moreover, the Xtext-based textual models are used to trigger code-generators in a model-to-text transformation step. Several code-generators are flexibly integrated by means of Eclipse Extension Points. Code-generators are realized using the Eclipse Xtend2[6] language. For this dissertation, a C++ code-generator for ACE/SMARTSOFT, a Python code-generator to interface with SymTA/S, and a text generator for ini-files have been implemented and integrated in a consistent model-driven

[6]http://www.eclipse.org/xtend/

tool suite called SmartMDSD Toolchain V3 (Technology Preview)[7].

3.3.3. Summary and overview of the meta-model packages

Figure 3.12 provides an overview of all the identified development views that are realized as respective Ecore meta-model packages. Chapter 4 addresses the component-development phase with the *component* and the *performExtension* packages (on the left in Figure 3.12). Chapter 5 addresses the system-integration phase with three views (again, realized as Ecore packages): *system*, *deployment*, and *performance* (in the middle of Figure 3.12). After that, Chapter 6 has the focus on the integration of a SymTA/S-based performance analysis into an overall robotic development process that is technically realized by the two Ecore packages *symtaBase* and *symtaConfig* (on the right in Figure 3.12). Chapter 6 also discusses some runtime aspects with respect to logging and monitoring.

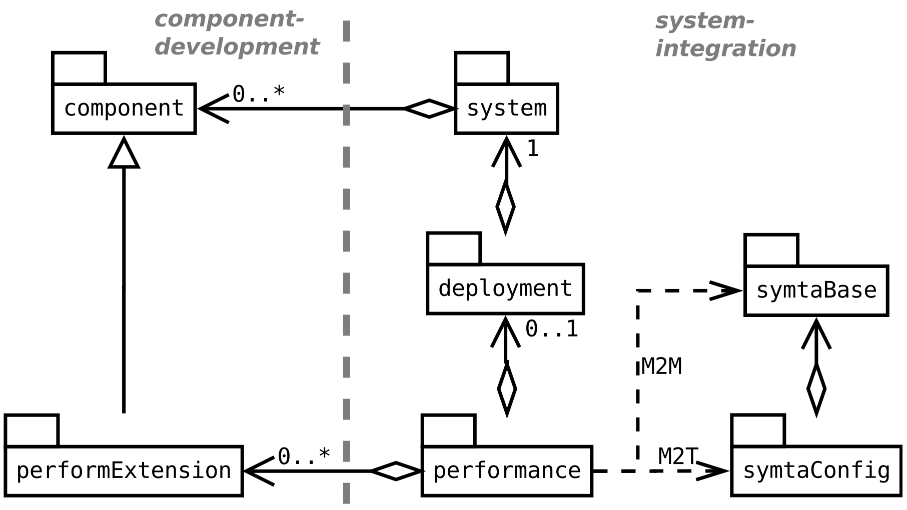


Figure 3.12.: Ecore packages overview

Each of the following three core chapters addresses aspects of one specific development phase at different levels of abstraction starting with conceptual, theoretical background discussions, over designing the suitable meta-models, and finally discussing relevant implementation-related design aspects. Thus, each core chapter groups and clusters coherent concerns. Therefore, each of the core chapters has its own brief section at the end that discusses related works of the respective development phase. Chapter 7 provides a consistent robotic example that uses the proposed development tools in all three phases of development.

[7] https://sourceforge.net/p/smart-robotics/smartmdsd-v3/ci/master/tree/

"Everything should be made as simple as possible, but no simpler."
—Albert Einstein

4 Extended SmartSoft Component Model

As stated in Chapter 2, the origins of this dissertation lie in the ideas of the overall SMARTSOFT approach, which provides, among other things, a sophisticated component model with clearly defined communication semantics. It is only natural to build on this expertise by extending and refining the underlying ideas. Therefore, this chapter uses the SMARTSOFT component model, which is refined in such a way that performance-related aspects can be modeled later in the *performance view* (see Section 5.3). The resulting simplified component model is extended so that functional boundaries for the end-to-end delays can be specified, thus restricting permitted choices for the subsequent system-integration phase.

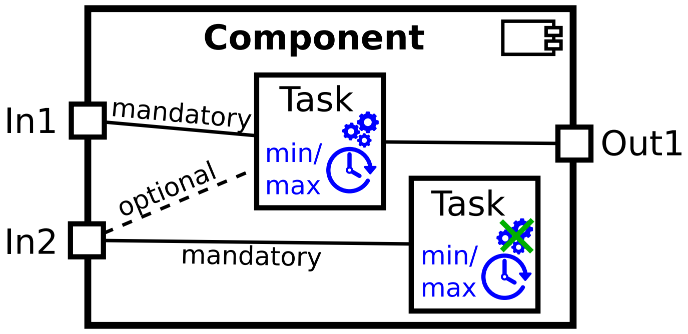


Figure 4.1.: Component model conceptual overview

Figure 4.1 illustrates the concept of the component model as it will be constituted in this chapter. Basically, a component model must specify the component itself, its communication and interaction interface characterized by input- and output-ports, as well as its internal tasks. Moreover, the model should indicate which tasks depend (strictly or optionally) on data received through respective input ports and/or generate data that is published on relevant output ports. The definition of tasks in a component should reflect implementation-specific execution characteristics, most notably the execution-time boundaries. Some task implementations might require a fixed (i.e., unmodifiable) specification, while others might allow a more flexible configuration with predefined boundaries. This latter flexibility is important for the subsequent system-integration phase where the component's tasks are configured so that application-specific needs are satisfied.

This chapter is structured into three sections. First, Section 4.1 discusses some theoretical foundations of the component development phase including the component interaction semantics, component internal (functional) view, as well as a small excursion into the configuration and coordination of a component (from a higher orchestration level). Next, Section 4.2 discusses and defines the *component* Ecore meta-model, thereby trading off different design choices along the way. At the end of that section, a design for a flexible task-trigger implementation is presented as a core technique to effectively separate the task implementation from its configuration. Section 4.3 concludes this chapter with a brief discussion of existing works related to component models.

4.1. The Component Developer View

Component development is the typical development phase for robotic technology experts, who provide configurable building blocks for later integration into various systems. The focus is on the internal structures and implementations of one single software component at a time without presuming or prematurely defining application-specific aspects that are not releveant in this phase.

In theory, it would be ideal to develop a software component in pure isolation. However, the functionality within a component typically either depends on information from other yet-unknown components and/or itself provides information to be used by other potential components. It is the very nature of any component to eventually form part of a bigger system and then to interact (i.e., communicate) with other parts (i.e., components) of that system. Section 4.1.1 provides some insights into what needs to be considered with respect to communication from the component's internal view.

Besides communication, a component also needs to manage its own computational resource demands. Such demands can result from the component's internal tasks (e.g. threads) or any other active parts implemented either in software or hardware. Resource management in this respect can be manifested in one of the following two alternatives. For proprietary implementations, resource consumption might be hardcoded (e.g. a proprietary hardware-driver internally implements threads or thread pools). In these cases, it is at least desirable to make the consequent resource consumption explicit for downstream developers. In other cases where the implementation is under control, it is better to postpone the exact task configuration until system-integration, which provides the required application-specific knowledge. Section 4.1.2 describes some common models of computation within a component that can be consistently configured in a later system-integration phase.

Finally, multipurpose robots can impose additional requirements on component flexibility and variability. Since multipurpose robots often need to execute many different tasks in any combination and in different situations, it often is necessary to reconfigure the software components dynamically at runtime. This situation-dependent reconfiguration is also called *system orchestration* [Lut+14]. Section 4.1.3 provides some further details about how *system orchestration* can impact communication and computation.

4.1.1. Toward middleware-independent service definitions

There is a huge community behind distributed computing with a long history of designing all kinds of general-purpose middleware solutions for virtually any imaginable purpose. A summary of the current state of the art of middleware design would be unrewarding and go far beyond the scope of this dissertation (although a few selected representatives are mentioned in Section 4.3). The important point—which is often inappropriately neglected in robotics—is to respect the knowledge of this community and to build on top of existing robust middleware solutions. Although the domain of robotics certainly has some domain-specific needs with respect to communication, these needs should be addressed by narrowing the general middleware semantics to fixed and reusable sets of domain-specific communication semantics and otherwise build on top of the deep knowledge, wide experience, and matured implementations provided by the various middleware communities. As a result, the communication semantics and implementations of robotic software components should be made independent of the underlying communication technology, thus allowing linking the components with any currently popular middleware technology that provides the required quality and performance attributes.

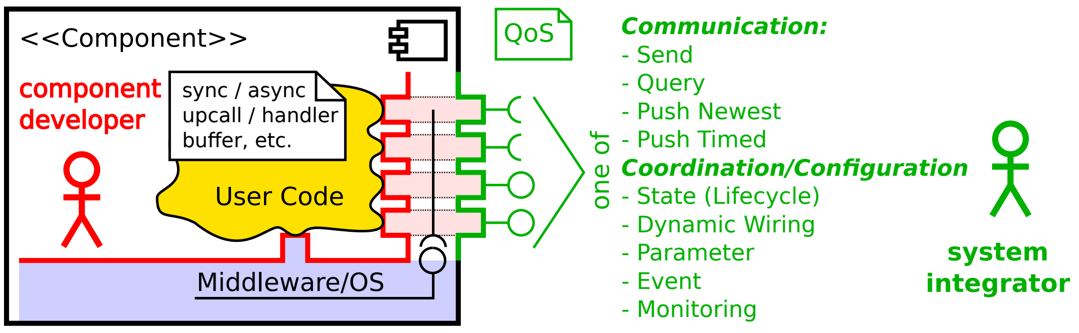


Figure 4.2.: Component's internal communication interface and external communication semantics [Sch+15]

Middleware independence is achieved by gaining control over the component hull [SSL12b]. This means that the communication semantics between components and the communication interface within components should be specified independently of the implementation details of the used middleware. With that in mind, a communication service can be specified from three distinct views, namely: (1) the communication semantics between components, (2) the component's internal communication Application Programming Interface (API), and (3) the mapping toward the underlying communication middleware. Figure 4.2 illustrates these three views of a service.

This dissertation uses the SMARTSOFT definition of a service for the following reasons. First, SMARTSOFT defines a fixed set of Communication Patterns with unambiguous communication semantics. Second, the SMARTSOFT Communication Patterns are independent of any middleware technology (in fact, several implementations are available, see Chapter 2 for more details). Finally, the abstraction level of the SMARTSOFT Communication Patterns is detailed enough to manage all required aspects with respect to intermediate buffers and involved synchronicity. However, the ideas presented in this dissertation are neither limited nor specific to the SMARTSOFT approach. Instead, this dissertation only requires an adequate abstraction level that exposes enough details with regard to the above mentioned communication aspects.

4.1.2. Component's internal models of computation

Seen internally, a component receives data from input services (in short: inputs) and provides calculated results through output services (in short: outputs). The intermediate part between inputs and outputs is the task definition. All three entities (inputs, tasks, and outputs) may or may not have different cyclic execution characteristics. Figure 4.3 illustrates the reasonable interaction combinations of one task receiving data from one input service and pushing the result to one output service. More precisely, the three input squares (on the left) represent the same input service but with different execution characteristics (periodic/sporadic and blocking/non-blocking). Similarly, the different rectangles for a task (in the middle of Figure 4.3) represent different task configuration alternatives of the same task, while the two output squares (on the right) represent the periodic and sporadic cycle types of the same output service.

First, for an input service, two basic execution characteristics can be distinguished—strictly periodic (i.e., with a stable update rate) and sporadic (i.e., with a varying update rate, typically bounded by at least the maximal update rate and sometimes also by a minimal update rate). This periodicity is the result from a preceding component providing the respective service. The other characteristic, blocking/non-blocking, results from the interaction of a task with that input service (see next).

The update rate of a task can be affected by various trigger sources. For instance, a task might internally use e.g. a sensor driver that blocks the task execution until new sensor values become available. This case is also referred to as self-triggered. In this case, the task follows the update rate of the sensor driver (as long as no other triggers are used). It would be technically possible to use several different triggers in one task; however, this would lead to very complex (i.e., hardly predictable) task execution characteristics, where slower triggers would dominate and faster triggers would lead to fluctuating jitters. Additionally, from the author's practical experience in building up several systems, there is no real reason for allowing multiple triggers at the same time. Allowing multiple triggers would cause more problems than providing benefits. Therefore, the task will hereinafter use only one trigger at a time (with the only exception of composite inputs, see below).

Another trigger source for a task is an input service. There are two possibilities how tasks can access data from inputs, namely synchronously (i.e., blocking) or asynchronously (i.e., non-

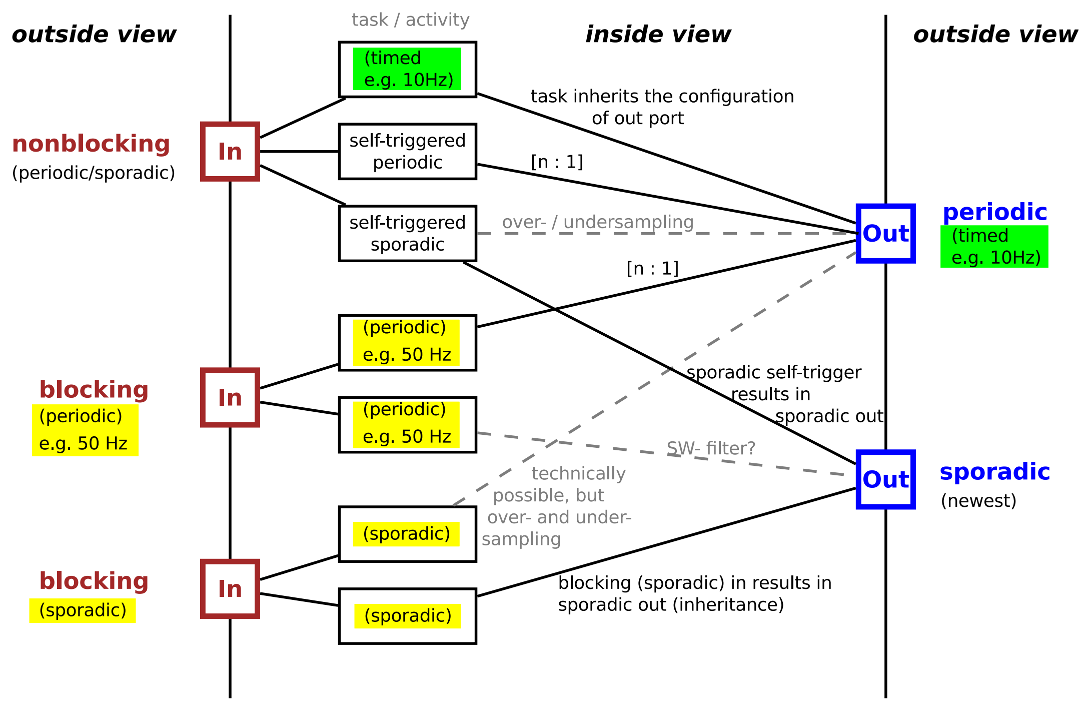


Figure 4.3.: Overview of all combination with respect to different models of computation within a component

blocking). The synchronous case means that each arriving data value immediately triggers a subsequent task cycle (assuming that the task's deadline is short enough, including the task's core execution time and the schedule, so that the task can follow the update rate of the incoming messages). As a result, the task cannot be executed independently of the input updates and thus inherits any update-rate characteristics from the input service (i.e., periodic/sporadic). This case is also referred to as blocking read and is directly related to the well-known Synchronous Data Flow (SDF) [LM87]. SDF provides a sound theoretical foundation; however, in practice the problem is of choosing the right queue size for the input messages. A queue for input messages could help to compensate for occasionally longer execution times of a task, assuming that a task in average can still drain the queue. While this might seem helpful at first glance, in real-world robotic cases, it hardly makes sense to process an old input message if a newer message already is available on the input port. For this reason, and to simplify the execution semantics, hereinafter an input buffer of size one is assumed. Thus, a message is stored in an input buffer until the next update arrives, which overwrites the buffer (also in the asynchronous case below).

Furthermore, it is worth noting that commonly a task might require data from more than one input. With respect to the task trigger, two cases can be distinguished. First, as argued above, at most one input should act as the task's trigger, meaning that all the other inputs are accessed asynchronously (i.e., non-blocking, see below). If more than one input should trigger the task's execution cycles (which is a relatively rare case), then the well-known *AND-* and *OR-activation semantics* [Hen+05] should be used to combine several input ports, which would again appear as one single and consistent trigger.

The other option to access data from inputs is the asynchronous case (i.e., non-blocking read). The task is executed with its own update rate independently of any inputs. As a result, depending on the individual update rates between the input(s) and the task, data might be either *oversampled* or *undersampled*, meaning that sometimes the same data values are used in several successive cycles of a task, or some intermediate data values are skipped. In practice, whether oversampling and/or undersampling is acceptable or not depends largely on the actual algorithm used in a task. In general, algorithms that can handle over-/undersampling allow for a higher reusability because the implementation of a task then becomes independent of the actual inputs and is thus more flexible and preferable.

For the interaction between a task and an output service, the following options need to be considered. An output service can appear to other components as being either periodic or sporadic. The periodicity might be the result of the periodicity of the associated task that calculates the respective results for that output service. Another (rather theoretical) option is that the output service itself is an active entity and thus can be executed independently of the linked task. This means that the output might publish old messages multiple times (i.e., oversampling the task's results) or skip some messages in between (i.e., undersampling the task's results). This leads to the following question: Why should a service actively do that without knowing the actual needs of the later connected subscribers? One might argue that undersampling in the output could save communication bandwidth in case none of the subscribers needs such a high frequency. But then again, if none of the subscribers needs such a high update rate, why should the producer task execute at such a high update frequency in the first place? In practice, it is sufficient to synchronously link a task with an output service and to configure the task directly if an update rate needs to be adjusted for the downstream subscribers. As a side benefit, this leads to a simplified computation model. Consequently, from the semantics point of view, the task and the output service can appear as one single entity (although the technical separation still makes sense as shown in a later Section 5.3).

In summary, the execution semantics of an output service directly depends on the execution semantics of its interlinked producer task. A task can be configured to either synchronously follow the update frequency of one input service, or be triggered by its own internal trigger (which is either an internal hardware trigger or a periodic timer). All the other input services are accessed asynchronously, thereby always using the newest available message.

4.1.3. Component's orchestration interface and lifecycle state automaton

The main purpose of every component is to be eventually integrated and then later executed in a bigger system. During operation, the system will need to coordinate the execution of several components in such a way that they altogether realize a system function (e.g. navigation, human-robot interaction, etc.). Multipurpose robots will implement several such functions and execute them in any order and some of them in parallel. Each system function might acquire certain system resources such as memory, CPU time, and devices. Some resource acquisitions and thus system functions might be mutually exclusive. For example, a camera mounted on a pan-tilt unit can only look in one specific direction at a time. In some other cases, it might not make much sense to combine certain system functions. For instance, it might not be possible (nor reasonable) to search for objects to manipulate while the robot navigates from one location to the next. For these reasons, and to save resources, it makes sense to switch off those system functions that are not needed for the currently executed activity. The main entities within a component that are responsible for resource consumptions are the tasks. Technically, an active task acquires certain resources which are released if a task becomes inactive. Therefore, controlling the activation and deactivation of the component's tasks also controls the component's resource consumption.

There is another link between the component's resource consumption and the system functions. A system function is realized by several interacting components. As shown in Section 4.1.1, components interact through services. Moreover, Section 4.1.2 shows that a service of a component is directly linked to one of the component's tasks and is therefore directly linked to the component's resources. Therefore, coordinating the (de-)activation of tasks of a component also coordinates the (de-)activation of the component's services.

This coordination is the main concern of the so-called *sequencer* [SLS11b; Lut+14]. In short, a *sequencer* is the master entity on the robot and is responsible for determining appropriate sequences of actions for accomplishing assigned missions. Single actions involve the execution of relevant system functions and thus the coordination of components with their services, resources, and tasks. This results in a clear control hierarchy, which is also called *system orchestration* [SLS11b; Lut+14].

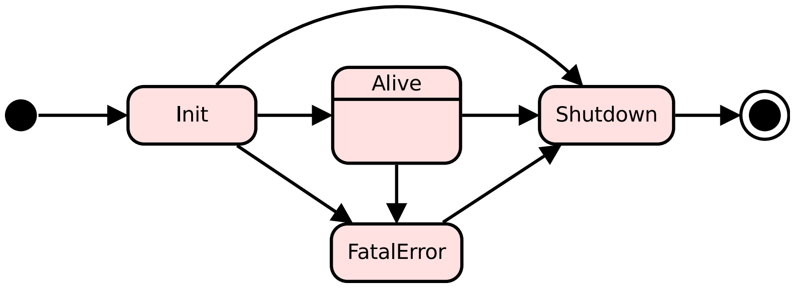


Figure 4.4.: Generic Lifecycle of a Component [SLS11a]

The core mechanism to control the component's tasks and services is the component's *life-cycle state automaton* [LSS11; SLS11a] displayed in Figure 4.4. The component's *life-cycle state automaton* controls the component's coordinated initialization, execution, and destruction. Only within the *Alive* state is the component able to provide its services. The *Alive* state in itself can contain several configuration modes of the component, each controlling whether relevant tasks are activated or not (see [LSS11; SLS11a] for more details). This makes the component's current state explicit, effectively prevents inconsistencies in execution, and increases the overall system robustness in execution. In practice, there are typically only a few defined modes a component can be switched into. Within the scope of this dissertation, this means that, for all those modes which might be critical in the sense that certain end-to-end guarantees must be met, relevant task configuration sets need to be specified and analyzed. This means that each individual task-configuration set might involve its own independent performance analysis (which is not in conflict with the approach presented here).

At this point, it is worth noting that *system orchestration* requires further standard services of a component, namely *parameter* [Sta+16], *dynamic wiring* [Sch04], and *events* [SSL12b]. However, these standard services are not relevant for analyzing and designing the component's execution performance. Hence, they fall outside the scope of this dissertation.

4.2. Extended SmartSoft Component Meta-Model and Flexible Task Implementation

This section presents the realization of the component properties discussed so far in an extended SmartSoft component meta-model. The extended component meta-model is separated into two Ecore meta-model packages (shown in Figure 4.5) to better distinguish between the component's basic structures described in Section 4.2.1 and the performance-related extensions described in Section 4.2.2. Section 4.2.3 additionally describes the framework-level extensions related to a flexible task-trigger implementation.

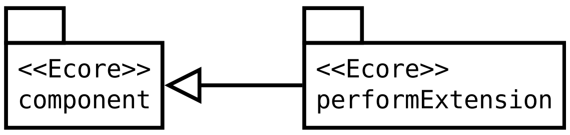


Figure 4.5.: The two Ecore-based component meta-model packages

At this point, it is worth mentioning that, while Ecore became a de facto standard for formalizing software meta-models (as also argued in Section 3.3), the presented structures and abstractions are neither limited nor specific to Ecore and can be easily redefined using any other preferred notation for specifying meta-models. Nonetheless, using a formal notation such as Ecore is important to ensure consistency and feasibility of the discussed structures and abstractions. However, Ecore alone is not enough to fully specify all the aspects (i.e., semantics) of a language. Therefore, this section additionally provides some technical details that complement the meta-model definition. At the time of writing, these details are specific to the Ecore-based meta-model definition, which can be considered a limitation of the Ecore-based approach. However, to the best of my knowledge, Ecore still is the most matured and widely used approach for specifying meta-models (despite its limitations).

4.2.1. Simplified component model with configurable services

The *component* Ecore package (on the left in Figure 4.5) is realized as the *component* meta-model in Figure 4.6. Appendix A.2 additionally provides a derived Xtext grammar that demonstrates one possible textual syntax. The component meta-model provides common elements which can be found in virtually any other popular component model. The general idea is not to reinvent the common elements from other component models but to provide an abstraction level which can also be mapped or even exchanged by the original model elements of any other preferred component model. In other words, this meta-model is a common denominator among several popular component models (see Section 4.3 for an overview). It is important to note that the core value does not come from this simple meta-model itself but from the interconnection with the other subsequent models. It is this very interconnection between several model views that enables a step-wise development of the entire system while guaranteeing overall system consistency.

At this point, it is worth mentioning that there is a common confusion about the meta-ness of a model. According to Voelter [Voe13, pp. 26–27], a meta-model defines the basic elements that can be instantiated in a model. A model therefore conforms to a meta-model. However, a meta-model itself can be a model conforming to a yet another, more abstract meta-model and so forth. In this dissertation, the term “meta-model” always refers to the Ecore-based meta-model definition (such as the Ecore diagram in Figure 4.6), whereas the term “model” refers to either an instantiated model or yet another meta-model (particularly when referring to related approaches, where these terms are also used interchangeably).

The root element of the component meta-model in Figure 4.6 is the EClass *ComponentModel*. A *ComponentModel* can have zero or one *Component* element. This restriction is important for the following reasons. First, it emphasizes the guideline for component developers to focus on one component at a time. Technically, each component project in Eclipse will instantiate exactly one component model with only one component definition. Second, if more than one *Component* element were allowed, one might be easily tempted to also define some connections between *InPort* and *OutPort* elements, which would be in violation of the component developer's responsibilities. A component must remain independent of any other component until the system integration phase (which is explained in detail in the next chapter).

Since the *Component* element is to be referenced in later system-integration models, it must be a named model element. Therefore, the *Component* element has an EAttribute *name* of type EString. It is interesting that the Ecore type EString is realized by the Java class `String`, which is a weak type for identifiers. A Java `String` thus acts as a technical primitive data type without any syntactic restrictions. In order to make the identifiers compatible with the Java identifiers, the type EString is further restricted by the notion of the Xtext's common terminal rule `ID` (see specification in Listing 4.1).

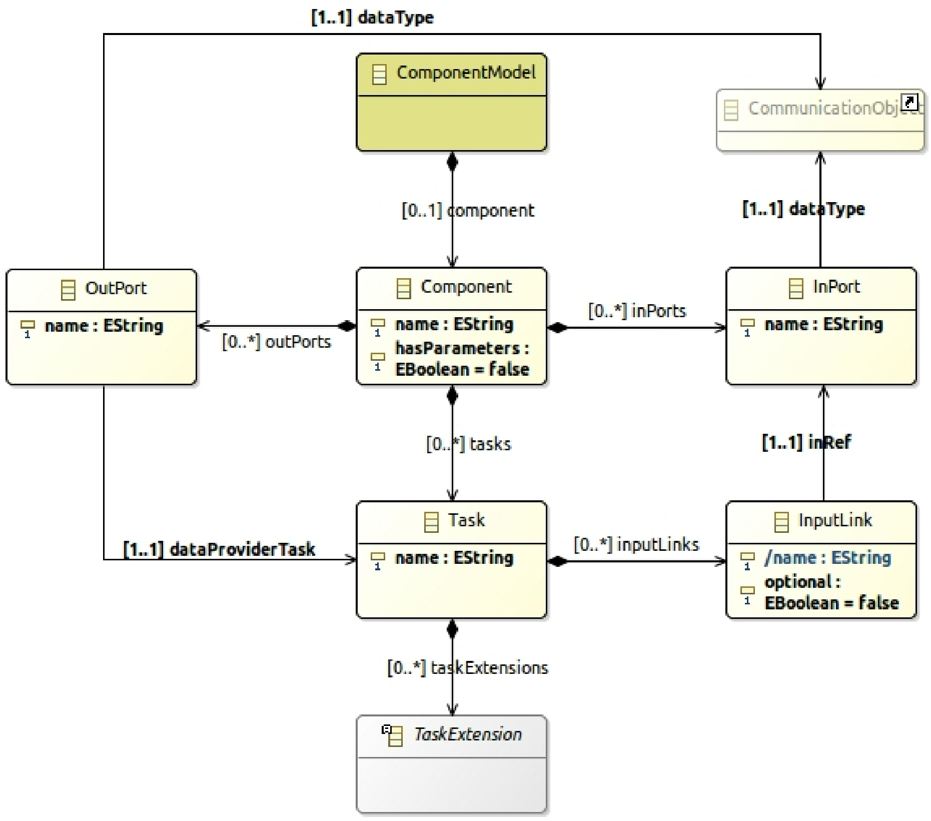


Figure 4.6.: Component Ecore meta-model

```
terminal ID: '^'?('a'..'z'|'A'..'Z'|'_') ('a'..'z'|'A'..'Z'|'_'|'0'..'9')*;
```

Listing 4.1: Xtext's common terminal rule for IDs

In general, each named element—an EClass with an EAttribute called *name* of type EString—can be referenced in other meta-models. To prevent name conflicts, the hierarchy of EClasses is translated into Java namespaces. Thus, only local elements—these are the infant elements defined from the same parent element—need to have unique names. For instance, all *InPort* elements in a component need to have a unique name, but *InPort* elements in different components can have the same name. By design, each Ecore element must have exactly one distinct parent element. As a result, named elements can be referenced using the so-called qualified names or respective qualified IDs (see specification in Listing 4.2, and its usage in the Xtext grammar in Appendix A.2).

```
QID:
  ID ('.' ID)*
;
```

Listing 4.2: Definition of a qualified ID

The second EAttribute of the *Component* element is *hasParameters* of type EBoolean. This attribute is a flag specifying whether a component additionally has component-related parameters, which are defined in a separate Xtext grammar (see [Sta+16] for more details).

Now, the three core elements that a component is comprised of are *InPorts*, *OutPorts*, and *Tasks* (see Figure 4.6). *InPorts* and *OutPorts* specify interaction points to other potential components in subsequent system-integration models. Hence, an *InPort* defines the requirement of this component to receive specific data messages, while an *OutPort* itself provides certain data messages to other potential components in a later system. In other words, *InPorts* and *OutPorts* define the interaction points between other components in a system and the current component's internal structures. Which other components exactly that might be should be irrelevant at this stage in the development. The only required semantic information is that certain message types are received and certain message types are produced. Message types define the communicated data structures. There are many suitable languages for defining message types such as the DDS Interface Definition Language (IDL). The presented component meta-model uses the Communication Object specification, which is defined in the Xtext grammar provided in Appendix A.1. However, any other convenient IDL can be used, which only needs to provide named elements (as described above) for all defined message types.

At this point, one might be curious about the abstracted definition of a service. As described in Section 4.1.1, a service has three distinct views: (1) the communication semantics between components, (2) the component's internal communication API, and (3) the mapping toward the underlying communication middleware. The only concern for a component developer should be to provide an implementation which uses the stable communication API of view (2) and otherwise leave the views (1) and (3) to the system integrators (see Chapter 5). Technically, the API of view (2) boils down to a getter method for requesting incoming messages and a setter method for publishing a message. Section 4.2.3 provides additional framework-level implementation details with respect to the API of view (2). These methods can be considered as an abstract API whose concrete mapping into the middleware is postponed until the system-integration phase. Depending on the later system-specific needs, these methods will be mapped to either a synchronous or an asynchronous call of the communication API. This decision should neither be prematurely made by the component developer, nor be part of the component model.

The connecting element between *InPorts* and *OutPorts* within a component is the definition of *Tasks* (see Figure 4.6). A *Task* represents, on the one hand, a functional block implementing the core functionality of its component, and on the other hand, a configurable active object. As an active object, a *Task* has a separate thread of execution with the execution characteristics specified in Section 4.1.2.

By design, a component can have more than one *Task*. Many popular component models such as e.g. RTC [And+05] restrict the usage of tasks to a maximum of one task per component, or even to sharing one task between several sequenced components. Although the rather simple RTC design might be appealing at first, it ultimately leads to rather fine-grained components more resembling active classes than independent components. The usage of sophisticated libraries such as OpenCV and MRPT demands a certain implementation flexibility within a component. Such libraries are naturally multi-threaded. Moreover, it is not always feasible to distribute the threads over several components because some threads might have very tight dependencies and lots of interactions that would lead to unreasonable resource overhead if these threads were to be distributed. Restringing the number of threads within a component would thus unavoidably lead to users defining their own threads beyond the component model, which would defeat the purpose. One of the common arguments for prohibiting multiple threads involves avoiding common multi-threaded programming errors such as race conditions and deadlocks. While there is some truth in this statement, there are also powerful design patterns such as the *observer pattern* and *active message queues*, which can be even automatically generated into a component (see Chapter 8). The point is that restricting the number of threads simply is **not** an option if the component model is to be accepted and adopted by real users—preferring an engineering model to a scientific model.

By definition, a component is never completely isolated (otherwise it would be an independent system). Instead, a component has at least either one *InPort* or one *OutPort* to interact with the rest of a system. Furthermore, a *Task* within a component might either require data coming from one or several *InputPorts* and/or itself generate data that is published over an *OutPort*. Messages arriving on one particular *InPort* can be shared between several *Tasks*. Moreover, a *Task* can use internal data sources such as a sensor driver. At the end of each *Task* cycle, calculated results are either directly propagated to an internal data sink (such as an actuator driver) or published through exactly one distinct *OutPort*. As described in Section 4.1.2, an *OutPort* cannot exist on its own without a *Task*, but a *Task* can be defined independently of an *OutPort*. This dependency is specified as *dataProviderTask* reference between an *OutPort* and a *Task* in the component model in Figure 4.6. Individual *InPorts* do not directly depend on specific *Tasks*, but a *Task* might require data from certain *InPorts*. More precisely, a *Task* might either strictly depend on certain *InPorts* or be able to optionally use some of the *InPorts* if available in the system and otherwise skip them during execution. This variable dependency is specified by the Boolean attribute named *optional* of the *InputLink* element between a *Task* and an *InPort*. For later referencing in other models, the *InputLink* must be a named element. However, from the component developer's perspective, it would be cumbersome to define an artificial name just for the link. Therefore, the *name* of the *InputLink* is automatically derived from the associated *InPort* reference (which is possible due to their parent–child relationship).

The remaining yet-to-be-described element in the component model is the abstract element named *TaskExtension*. Initially, this element might not make much sense on its own, because abstract elements cannot be instantiated and need to be derived by concrete elements. However, the element *TaskExtension* is only a placeholder for additional (not yet defined) properties of a task which are defined later in the meta-model extension described next. Therefore, it makes sense to specify an abstract element without children to indicate a specific extension point.

4.2.2. Component-performance extension

This subsection presents the *performance extension* (see Ecore diagram in Figure 4.8) of the component meta-model from the previous subsection. Technically, the extension is realized by derivation as shown in Figure 4.7.

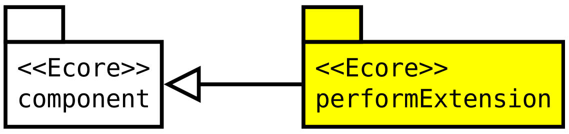


Figure 4.7.: Performance Extension meta-model extends the component meta-model through derivation

The *performExtension* meta-model inherits all the elements from its parent *component* meta-model and thus can be used interchangeably. Some of the *component* meta-model elements are refined by accordingly derived elements to provide additional model details. The Xtext grammar provided in the Appendix A.2 is based on the *performExtension* meta-model, including all the elements from the *component* meta-model. One of the reasons for separating the core elements in the component meta-model from their extensions is the need to keep the original meta-model simple and clean, and thus exchangeable with other popular component meta-models.

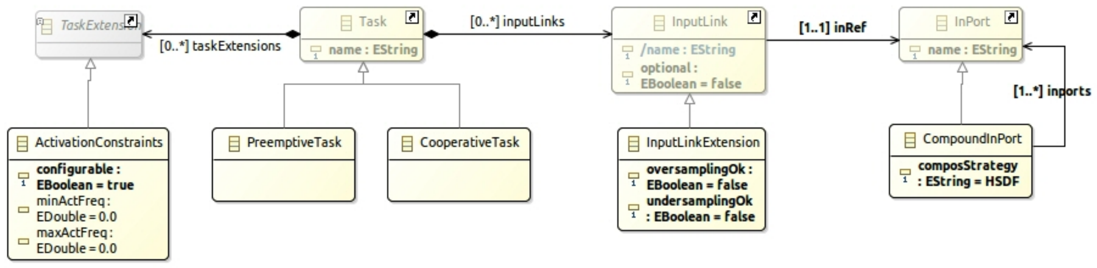


Figure 4.8.: Component Performance Extension Ecore meta-model

Figure 4.8 shows some of the imported core elements from the *component* meta-model (the faded-out elements) with their extensions (all the derived elements). The first extension is the EClass *ActivationConstraints*. This element is derived from the abstract core element *TaskExtension*, thus providing an initial specific extension of a *Task*. In other words, a *Task* composes

ActivationConstraints elements, which was not yet possible in the original Ecore meta-model (see preceding section). Unfortunately, there is no direct way to specify (or respectively restrict) the cardinality of *ActivationConstraints*. Because an arbitrary number of *TaskExtension* elements can be created for each *Task*, an equally arbitrary number of *ActivationConstraints* can be created, and Ecore does not provide any direct way to restrict the number of derived elements. However, for the *ActivationConstraints*, it does not make sense to define more than one element per *Task*. In general, this is a common problem when extending Ecore meta-models, and there is a rather recent initiative within the Eclipse ecosystem called *EMF Facet*[1], which addresses several issues related to model extensions in Ecore. However, at the moment *EMF Facet* is not yet matured enough to be integrated, and the *performExtension* is a rather simple extension with only one single "problematic" element. Therefore, a hand-crafted solution is preferred based on an Xtext model check. The Xtext infrastructure by default allows specifying additional model checks which are executed while the model is created in the model editor or is parsed from a character stream. Listing 4.3 shows the respective model check (implemented with *Xtend*[2]) that prints an error in case more than one *ActivationConstraint* element have been defined for a *Task*.

```
@Check
def checkSingleActivationConstraintsPerTask(Task task) {
  if(task.taskExtensions.filter(typeof(ActivationConstraints)).size > 1) {
    error("There are several ActivationConstraints elements defined for the
    task "+task.name, ComponentPackage.Literals.TASK__TASK_EXTENSIONS)
  }
}
```

Listing 4.3: ActivationConstraints cardinality check

Now, the model semantics for *ActivationConstraints* is as follows. As shown in Section 4.1.2, the cyclic execution of a *Task* can be triggered from different sources. For instance, a *Task* can use an internal trigger provided by e.g. an internally used sensor driver. In this case, the *Task* execution characteristics cannot be arbitrarily changed but are bound to the execution characteristics of the sensor driver. Therefore, *ActivationConstraints* provide the Boolean flag called *configurable*, which in this case needs to be set to **false**. In addition, the maximal and/or minimal activation frequencies (of e.g. the sensor driver) can be specified using the respective optional attributes *maxActFreq* and *minActFreq* of *ActivationConstraints*. Another common use case is that the *Task* does not strictly depend on a specific trigger. A common mistake in this situation is to prematurely bind the *Task* to be triggered by one of the associated *InPorts*, or to even manually implement a periodic timer. The interesting point is that, in retrospect and from the perspective of the appropriate *Task* implementation, it does not make much difference whether the *Task* is triggered by one of the *InPorts* or by a timer. However, at the system level it is of great value to be able to choose between these different trigger sources (as is explained in more detail in Chapter 5). For this reason, neither an *InPort* trigger nor a periodic timer is allowed to be di-

[1]EMF Facet: http://www.eclipse.org/facet/ (last visited on 6th May 2016)
[2]Xtend language: http://www.eclipse.org/xtend/

rectly defined within a component model. Instead, a component developer can optionally specify execution boundaries for each *Task* using, again, the attributes *maxActFreq* and *minActFreq*. In addition, the Boolean flag *configurable* set to **true** indicates that the task's trigger can be later changed considering the minimal and maximal execution boundaries.

The next extensions are the two derived EClasses—*CooperativeTask* and *PreemptiveTask*. These two elements do not add any further attributes. Instead, they are placeholders for a refined semantic behavior of a task. More precisely, *PreemptiveTasks* within a component, on the one hand, are directly mapped onto actual system threads (if the system provides them), which can be executed and distributed across several physical CPU cores. *CooperativeTasks*, on the other hand, are not really executed in parallel. Instead, they are sequential and an internal scheduler determines their order of execution based on common scheduling strategies such as First-In First-Out (FIFO) or earliest deadline first (EDF). Both kinds of tasks have their pros and cons. For instance, *PreemptiveTasks* allow utilizing multicore CPUs, but, if data is exchanged between the tasks, the component developer needs to manually implement relevant mutual exclusion and synchronization mechanisms (thereby using mutexes, semaphores, guards, condition variables, etc.). *CooperativeTasks*, on the other hand, ease the implementation of a task and prevent potential race conditions because different tasks can never write and read the same data at the same time. There is no universal rule to prefer one task type over the other. However, if tasks within a component are independent of each other (i.e., no data is exchanged), *PreemptiveTasks* typically allow for more efficiency, because the system scheduler can then optimally schedule their execution considering all the other system tasks. However, if a component consists of many tasks with many interaction points and rather short individual cycle times, then *CooperativeTasks* might be a better choice. Ultimately, this decision is the responsibility of the component developer, because he/she is the one providing the respective task implementations and is aware of implementation-specific needs and implications.

Next, the EClass *InputLinkExtension* extends the *InputLink* from the component meta-model by providing two additional attributes: the Boolean flags *oversamplingOk* and *undersamplingOk*. Again, the component developer should not directly select a task-trigger, but he/she should define execution boundaries by specifying *ActivationConstraints*. As shown in Section 4.1.2, this means that the relevant task(s) might asynchronously read data from the linked *InPort(s)*. In other words, tasks might be executed with different update frequencies than the incoming data. This leads to either oversampling or undersampling of data. As further discussed, an *InPort* does not have an input queue but only a buffer of size one, thus only storing the newest data message. Analyzing common component implementations reveals that there are different cases. In some cases, it does not make much difference whether old values are used several times or some values are skipped in between. In other cases, data values should not be skipped or old values should not be reused. Ultimately, whether under- and oversampling are acceptable or not needs to be decided and specified by the component developer, since he/she has the relevant knowledge about the implementation-specific constraints (if any).

The last extension is the EClass *CompoundInPort*. This element allows grouping several *InPorts* together in order to use that group as one distinct *InPort* trigger for a task. In most cases, the different *InPorts* of a component should be independent of each other because their individual data typically result from different components in a system. However, in some rare cases it might make sense to synchronize the arrival times from different *InPorts*, for example, in order not to miss any incoming updates and to immediately react to incoming updates. For these rare cases, the *CompoundInPort* groups several (at least two) relevant *InPorts* of a component and is used in place of an *InPort*. The EAttribute *composStrategy* allows specifying the required composition strategy based on the *AND-* and *OR-activation semantics* [Hen+05].

4.2.3. Flexible TaskTrigger implementation and decoupled component's internal service interface

The preceding section described the component meta-model independently of any robotic frameworks. However, at some point a meta-model needs to be mapped onto a concrete robotic framework so as to provide a grounding into an implementation (according to the general MDSE ideas).

This section describes some design aspects of the flexible (i.e., configurable) task definition. Thus, the meta-model elements from the component meta-model presented above are mapped onto the ACE/SMARTSOFT's framework classes in a model-to-text transformation step (i.e., the code generation) according to mappings in Table 4.1.

Component meta-model element	ACE/SMARTSOFT framework class name
Component	`SmartComponent`
PreemptiveTask	`ManagedTask`
CooperativeTask	(not yet implemented)
InPort	`PushClient`
OutPort	`PushServer`

Table 4.1.: Mappings from component meta-model elements toward corresponding classes in the ACE/SMARTSOFT framework

Figure 4.9 shows a UML class diagram with selected C++ classes of the ACE/SMARTSOFT framework that implement the configurable interaction between a *Task* (implemented as `ManagedTask`) and the *InPorts* (implemented as `PushClient` ports).

The two classes `PushClient` and `ManagedTask` (colored yellow in Figure 4.9) are part of the ACE/SMARTSOFT API that a component developer can use to provide component-specific implementations. For each modeled *PreemptiveTask*, a `ManagedTask` is initialized and the component developer needs to implement at least the core task method `on_execute()`, which is cyclically executed according to a configured trigger. Besides, the two optional methods `on_entry()` and `on_exit()` can be overloaded and implemented to provide optional initialization and the respective clean-up procedures of a task (if necessary). It is worth noting that the code within `on_execute()` never directly calls methods from `PushClients`. Instead,

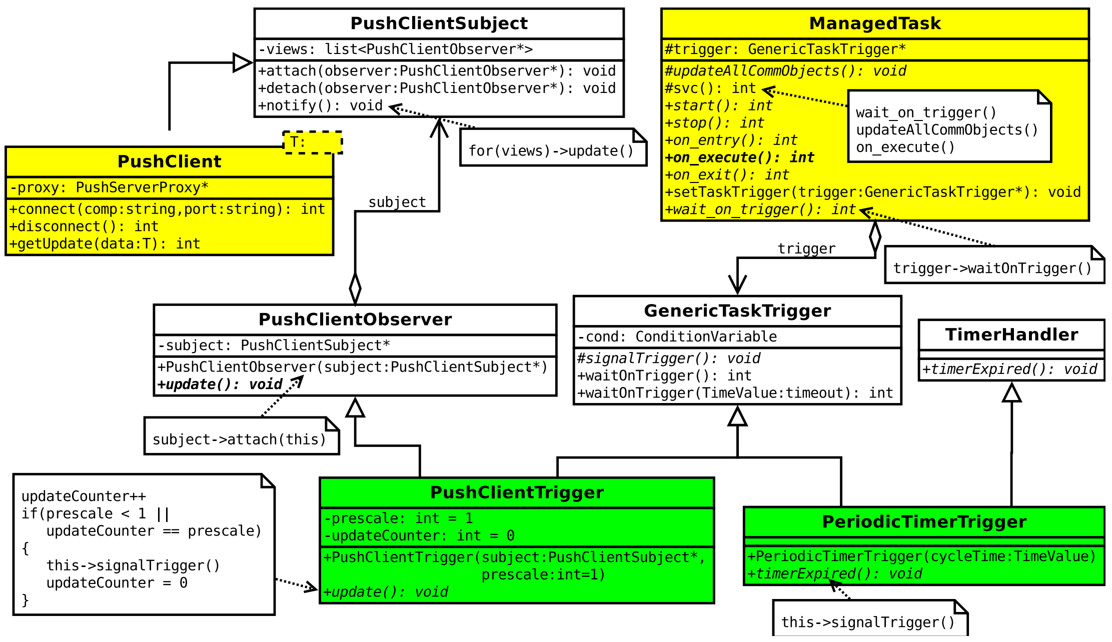


Figure 4.9.: Generic Task-Trigger UML Class Diagram

a `ManagedTask` automatically calls the `getUpdate(...)` method from each associated `PushClient` within the task's method `updateAllCommObjects()` at the beginning of each new task cycle. Each updated Communication Object is then stored as a member of the relevant `ManagedTask` instance and can be directly accessed at any time during a task cycle.

All `ManagedTasks` and `PushClient` ports are initialized properly from within a `SmartComponent` class (not displayed in the UML class diagram). There are three different alternatives to how a `ManagedTask` can be triggered, which are controlled using the task's `setTaskTrigger(...)` method. First, if the *ActivationConstraints* provide a *configurable* flag set to **false**, a custom trigger implementation needs to be provided by overloading the virtual method `wait_on_trigger()` in the derived task classes. If the *configurable* flag is set to **true**, it depends on the later performance model (see the next Chapter 5) whether the trigger will be a `PeriodicTimerTrigger` or a `PushClientTrigger` (see the two classes at the bottom in Figure 4.9 in green). In technical terms, this decision is stored in an ini file, which is read during the initialization of the component and the respective trigger-kind is passed to the task. Both trigger classes derive from the `GenericTaskTrigger` base class, thus inheriting the interface of the base class. The method `waitOnTrigger(...)` of the `GenericTaskTrigger` is used by a task to wait on the next task cycle according to the individual trigger strategy. The individual trigger strategy is specified by the relevant base class `TimeHandler` or `PushClientObserver`. The `TimeHandler` is triggered from a configured system timer class (not displayed in the diagram) that periodically calls the virtual method `timerExpired()`. The derived `PeriodicTimerTrigger` class implements the method `timerExpired()` in such a way that the method `signalTrigger()` from the parent `GenericTaskTrigger` class is called, which again releases the blocked method `waitOnTrigger(...)` and thus initiates the next task cycle.

The `PushClientTrigger` implements the *observer design pattern* [Bus+96]. The `PushClient` class thus derives and implements a `PushClientSubject`, which is the subject part (also referred to as model) of the observer pattern. Each time an update arrives in the `PushClient`, the `notify()` method is called, which again calls all `update()` methods of the attached observers (also referred to as views). One peculiarity of the `PushClientSubject` is that it can be optionally configured to call the `signalTrigger()` method upon each nth update, specified by the `prescale` factor. By default, the `prescale` is set to one, meaning that each time the `update()` method is called, the `signalTrigger()` method is executed.

Overall, an advantage of this design is that *InPorts* and *Tasks* are effectively separated from each other and their coupling becomes configurable at the system level. Thus, implementation-specific requirements can be considered, and a new degree of freedom is provided for later system integration, which makes the components more flexible, reusable, and thus generally more useful.

4.3. Related Works and Conclusion

This section selectively lists some other works related to component modeling particularly in the domain of service robotics. This list is not meant to be exhaustive in the sense of reflecting all general model-driven approaches in robotics, but it focuses only on components (according to the focus of this overall chapter). The other approaches are listed in the related works sections of the successive core chapters. A general overview of related works is given in Chapter 2.

4.3.1. Component models

Several attempts have been made to unify and to harmonize several component models to find the one that is generally and universally applicable. The OMG's CORBA Component Model (CCM) [CCM06] and the component definition of UML [UML15] are just two prominent examples in this category. In the entire history of computer science, no attempt to unify component models has achieved a satisfactory result that can be widely accepted. There are many reasons for that. For example, the unification of component models typically leads to one of the following two extremes. First, more and more details are thrown away until a very generic component model remains which does not conflict with the widely accepted constraints. However, this component model is so generic that it is practically useless; it neither provides relevant properties of a component, nor can it be transformed into a helpful code. What typically remain are entities called components with ports, which are basically what UML [UML15] offers. The other extreme is that many overlapping details are included, many of them optional and some even conflicting.

CCM [CCM06] is an example for this category. These component models inevitably become so complex that ultimately nobody really understands or uses them.

Consequently, component models need to be domain-specific. Although this means that each domain comes up with at least one component model specification of its own, this approach is much more rewarding and has produced several helpful domain-specific component model standardizations. For instance, for the DRE domains[3] AADL [AAD04], MARTE [MAR11], and SysML [Sys12] are good examples. Furthermore, RobotML [Dho+12], BCM [Bru+13], RTC [And+05], and SmartSoft [SW99] are popular examples for robotic component model specifications. Although these models are definitely valuable contributions, it is important to note that isolated component models alone are not enough. Instead, component models essentially need to be connected with system (integration) models.

Another rather recent initiative is the Unified Component Model (UCM) [UCM]. It has been at the beta stage during the time of writing. UCM provides a flexible and extensible component model based on the component-port-connector idea, which allows additional annotation of policies and contracts at the modeling level. While this recent development is promising, it remains to be seen in what ways this standard develops and whether it gets widely accepted. For future studies, it could be interesting to investigate whether the presented structures and abstractions (which are independent of any concrete component model) can be expressed with UCM.

But then again, component models should be interconnected with system models so as to ensure a systematic and consistent handover of knowledge between successive development phases with their individual views on the system and involved developer roles. Furthermore, linking modeling views is possibly the core ingredient for making the step from a model-*based* (i.e., models just for documentation) toward a model-*driven* engineering approach that includes helpful code generation, stepwise system refinement, and inherent consistency checks.

4.3.2. Robotic frameworks and communication middlewares

Component models alone are not enough to realize a truly model-*driven* approach. Instead, the semantics of a component model are partially defined by its grounding in the implementation of an actual component. There are basically two ways to realize this mapping. First, a component can be entirely generated out of a component model. While this might be an appealing option in theory, it is quite unrealistic in practice. Real component implementations use sophisticated and efficient libraries that sometimes do not have any software models. Moreover, components should rely on powerful middleware implementations, which also sometimes do not have a model-base. Therefore, component realizations (in short- or mid-term) will likely remain a combination of model-driven parts (i.e., models and code generation) and other parts directly implemented in a robotic framework and communication middleware.

The robotic community has presented a several robotic frameworks such as ROS [Qui+09], OROCOS [Bru01], RT-Middleware [And+05], and SmartSoft [SW99] (to name just a few

[3]DRE domains are considered generally related and similar to the domain of service robotics.

prominent ones). Within the scope of this dissertation, the proposed component model is independent of a specific framework implementation. Yet, the presented component model requires certain features of a framework to be exposed for modification (such as the internally used threads, exact communication characteristics i.e., synchronicity, and communication buffers). This dissertation uses the SmartSoft framework due to its rich and flexible interface, which has been further refined to allow an even more flexible configuration of threads in a component. Other frameworks such as ROS might provide valid alternatives for future works. This, however, is yet to be assessed.

Another important aspect of components is related to communication. There is a huge community providing many powerful and general-purpose middleware solutions. The Object Management Group (OMG) alone provides two common standards—DDS [DDS07] and CORBA [Cor]. As stated in Section 4.1.1, it is important that a component model can be mapped on any existing middleware implementation if it offers the required performance and quality properties. This also means that the component model itself should be freed from any middleware specifics, which is also the case in this dissertation. The dissertation of Schlegel [Sch04] discusses in detail how this abstraction level can be achieved and which issues need to be solved. Hence, the middleware abstraction layer for robotic component models can be considered solved. Consequently, a more in-depth comparison of different middleware solutions would be pointless for this dissertation.

4.3.3. Conclusion

This chapter has presented the component developer's perspective as a distinct phase of the overall robotic development workflow. The relevant needs and concerns of a component developer are discussed in Section 4.1. Thereafter, Section 4.2 discusses a design of a meta-model for this development phase. Thus, the individual design choices are explained along with the involved semantics. Section 4.2 also presents a flexible task-trigger design at the framework level, allowing a direct mapping from the model level. Finally, Section 4.3 lists some related works with a restricted focus based on the scope of this chapter.

Overall, the component meta-model presented in this chapter allows the complete implementation of a component in such a way that the component's code can be considered as closed-source in the next development phases. Simultaneously, as will be shown in the next chapter, the component model reveals enough details and configuration options at the model level, which enable system integrators to adjust the components so that the then available application-specific, system-level needs can be satisfied.

"Everything is theoretically impossible, until it is done."

—Robert A. Heinlein

5 Models in the System Integration Phase

The preceding Chapter 4 described the component model as the main view during the component development phase. The objective of that phase is to design, develop, and implement flexibly usable building blocks. These building blocks are now used as the main elements in the next development phase, namely *system integration*, which forms the focus in this chapter hereinafter. The building blocks are configured using modeled (i.e., predefined) configuration options only (i.e., gray box components).

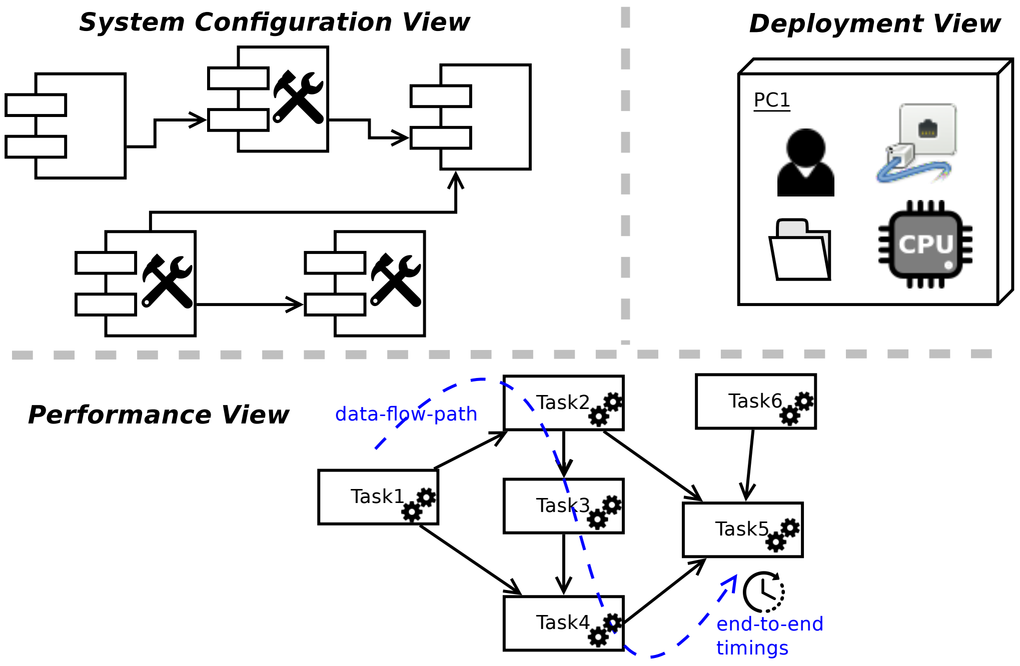


Figure 5.1.: Conceptual overview of the three core system integration views

The system integration phase combines several views with specific concerns. In general, system integration is about building up an entire system out of reusable components. The main focus in the *system-configuration* view (see top left in Figure 5.1) is to instantiate components, to specify initial wiring, and to refine component configurations. Moreover, system integration needs to prepare the designed system for deployment. Therefore, platform-specific details need to be specified such as the distribution of component artifacts over several network nodes on the robot and the specification of the execution environment. These platform-related details are related to the *deployment* view (see top right in Figure 5.1). These first two views have been inspired by our previous SmartMDSD Toolchain (version 2). Additionally, these two views have been refined and extended in such a way that they can provide a consistent foundation for the next novel view named *performance* (see bottom in Figure 5.1).

The *performance* view allows modeling performance-related aspects of components at the system level. In this view, the interaction links between individual tasks are refined and configured in such a way that desired overall end-to-end response times are satisfied. Therefore, individual tasks have to be configured only at the model level within their individually predefined configuration boundaries, ensuring that the configurations do not conflict with component implementations. Furthermore, the overall end-to-end latencies and jitter of data-flow paths must be calculated to analyze individual configuration choices. This calculation requires simulating runtime conditions, such as the scheduling of tasks considering their mutual dependencies (e.g. synchronous triggering) and the occurring sampling delays. Therefore, external analysis tools can be used as described in the next Chapter 6.

This chapter hereinafter is structured into three parts. First, the next section discusses some common aspects of the system integration phase. This discussion emphasizes the role of system integration in an overall robotic development workflow that involves the interactions with the preceding component development phase and the successive runtime phase. Section 5.2 presents the meta-models from the first two views (on the top in Figure 5.1), and Section 5.3 introduces the meta-models of the novel *performance* view (on the bottom in Figure 5.1).

5.1. The System-Integrator View

System integration is about combining the hitherto independent parts (i.e., the software components) into a coherent whole (i.e., the overall system) and to make the system ready to be deployed and systematically executed on the robot. The core responsibility of system integrators in this phase is to design the overall system in such a way that all relevant application-specific needs are satisfied. The system integration phase acts as a bridge between individual component development and coordinated execution on the robot.

The first common step in this phase involves the selection of the right components that provide the required functionality with required quality and performance properties. Thus, integration means defining initial connections between interacting components, the initial parametrization of components, the platform details, and the mapping of components to platforms and the actual hardware. System integrators are application domain experts with a deep knowledge about application-specific needs and constraints. The driving force for system integrators is the aim to satisfy customer needs. Therefore, system integrators need to understand technical features, performance attributes, and quality properties of components without the need to go into too many technical details. This enables system integrators to make informed and balanced decisions for finding the right trade-offs between what customers really need, what is technically possible, and what is economically reasonable.

System integrators rely on component models rather than on component implementation. Thus, component models should expose open configuration options as well as those properties that have a system-level impact. Consequently, system integrators can select the right components and tailor them to meet the application-specific needs, without being forced to understand all the internal, low-level implementation details of each individual component. Hence, anything else that is not relevant at the system level should be hidden (i.e., encapsulated) within relevant components. This additionally means that, technically, the component and the system-integration models must be interlinked at the meta-model level so as to automate and formalize the handover of knowledge from component developers to system integrators and later to the executed system. In brief, this allows implementing model-driven tools that guide and support different developer roles in different development phases with proper abstraction levels.

The following section discusses the role of system integration with respect to runtime adaptation. Section 5.1.2 discusses the meaning of the system's overall responsiveness as one of the core qualitative system properties. Finally, Section 5.1.3 introduces the notion of *cause–effect (task) chains*.

5.1.1. Variability management and stepwise system refinement

Bridging the gap between component development and runtime involves two related concerns for system integrators. On the one hand, system integrators can exploit available design variability in a component model to adjust the components so that they fit into the current system, by restricting or binding some of the provided configuration options. Model-driven tooling can support in ensuring the overall consistency between the individual configurations. On the other hand, the system must not be completely fixed before deployment, instead, system integrators can leave open some of the configuration options that can be exploited by the robot itself for autonomously making own decisions at runtime and thus for adapting to contingencies and changing situations (see [Lot+14] for more details on this topic). The role of the model-driven tooling thereby is to ensure the overall system consistency at design-time and to transfer a consistent system configuration into the runtime (either through code generation or through runtime model interpretation).

In this sense, system integration is about finding the right trade-off between binding (i.e., restricting) design-time variability and equipping the robot with sufficient runtime adaptability. This trade-off will shift between different systems, as it depends on the target application with its inherent environment complexity (i.e., dynamic or static), the tasks to perform (i.e., multi- or single-purpose robot), and the desired degree of autonomy. Interestingly, realistic robotic systems always need to effectively strike a balance between two extremes; it is neither reasonable to pre-program every decision of the robot in advance, because the robot then simply is too inflexible, nor is it feasible to let the robot itself learn everything on demand due to the unbounded complexity of the real world. As a result, system integration is not the final step in robotic software development. In fact, runtime variability also has a severe impact on the overall system design and on analyzing the overall system performance. This particularly means that one single and static performance analysis for an entire robotic system is not enough, and that runtime reconfigurations need to be considered. In other words, a typical system will require the specification of a finite number of configuration sets and the performance of each configuration set needs to be individually analyzed.

5.1.2. Designing reactivity and responsiveness of a robotic software system

Another common concern of system integration is to define the flow of data between interconnected components that form networks of components. Compared with the component development view (discussed in Section 4.1) there is a systematic shift from component-internal view—where data flows from the component's *InPorts* through the component's *Tasks* toward the component's *OutPorts*—to the component-outer view—where data flows from one component's *OutPorts* toward another component's *InPorts* and after a "magic" internal transformation within that component, new data appears on the *OutPorts* of that component, which again can be used by yet another component's *InPorts* and so on. This means that components form interconnected chains or even networks of components.

Because service robots operate in dynamic environments (sometimes involving unpredictable interactions with humans), they always need to cyclically scan the environment using their sensors to determine relevant aspects of the world's current state, which continuously changes. Moreover, a robot needs to execute appropriate actions to achieve its designated objectives. Real-world actions are not atomic but take a certain amount of time to complete and they can fail while executing or can produce unexpected results (respectively changes in the world). Consequently, for the robot, executing actions means cyclically monitoring the environment and the progress of the current action to react to anomalies and contingencies that deviate from the expected results.

This particularity of robots has a direct influence on the overall robot's software-system design. For instance, some components implement sensor functionality of the robot. These components usually do not need any *InPorts* but use an internal driver that directly communicates with a sensor device mounted on the robot to regularly receive new sensor updates and to publish the sensor-data using the component's *OutPorts*. Other components receive data from one or several such sensor components (through their *InPorts*) to derive new information by aggregating and transforming data using all kinds of powerful robotic algorithms. These components publish their produced information through their *OutPorts*. Finally, yet other components are responsible to operate the robot's actuators. These components typically have some *InPorts* to regularly receive action commands and these components typically do not require *OutPorts*, but directly communicate with an internal actuator driver.

An interesting observation is that individual components do not always implement one single function; instead, they regularly cluster and implement several functions that belong together. For instance, a component can act as both; an actuator and at the same time as a sensor because many actuators also directly provide their internal state (such as the odometry). Nevertheless, it is highly desirable to be able to deliberately design the reactivity and responsiveness of the overall system because this has a direct impact onto the robot's safety and execution quality (as is also discussed in Section 3.2.5). The overall system's responsiveness is defined by the end-to-end time from receiving sensor updates, traversing several intermediate components, and finally commanding an action.

There are many system factors that directly or indirectly influence this end-to-end time. For instance, the available processing power and communication bandwidth of the execution context provide the foundation for the possible execution and communication times. However, it then depends on how many tasks are executed in parallel on the same processor, the used scheduling policy, and how much data is communicated on the same communication channel within a certain period of time, which, altogether directly influence the overall delays from sensors to actuators. Designing these factors alone is a great challenge, which nevertheless might be manageable using common real-time scheduling and real-time communication approaches. However, there is yet another aspect that needs to be considered, which is discussed next.

5.1.3. Cause–effect (task) chains

As mentioned in the preceding section, designing end-to-end delays from sensors to actuators is a crucial factor for the system integration phase. Therefore, the execution of threads needs to be simulated and analyzed. Individual threads in a system are specified by individual component's tasks. Since components have mutual dependencies and need to interact (i.e., communicate) with each other, some tasks might also depend on each other, which needs to be considered in a performance analysis (see Chapter 6 for more details). In other words, some tasks might trigger other follower tasks in a kind of execution chain. Therefore, the next part of this section discusses those system factors that influence the interaction options in such execution chains.

From the computation point of view, the component's boundaries are not important. A component is meant to provide a container for the component's *InPorts*, *OutPorts*, and *Tasks*. *InPorts* and *OutPorts* represent passive interaction points to other components. *Tasks* are the only active parts of a component. In other words, *Tasks* are the actual physical entities in a system that interact with each other. For this reason, the management of end-to-end times demands a more physical view of the system by exposing the component's internal *Task* specifications and their interaction possibilities. These two concerns seem contradictory at first glance, because encapsulation means hiding certain details, while exposing *Tasks* means exposing the component's internal details. However, there are two mitigating factors that are discussed next.

First, as shown in Section 3.2.5, not all system functions of a robot are time critical (nor even safety-related) and thus would need to be considered for the responsiveness of a system. This reduces the actual number of *Tasks* to be considered in a performance analysis. Second, not all details of *Tasks* need to be exposed at the system level. For instance, the component meta-model, as defined in the previous chapter, already specifies *Tasks* with links to *InPorts* and *OutPorts*. The remaining configuration options relate to further specifying the synchronicity between *InPorts* and *Tasks* and to specifying the activation semantics (i.e., the trigger) for individual *Tasks*. All this allows the definition of a lightweight task-configuration meta-model which does not overwhelm a system integrator with otherwise unimportant details. Such a meta-model view enables system integrators to design the overall system's responsiveness properties at the right point in time where the relevant knowledge about the application domain becomes available and before redesigns might become too costly.

The management of the different task-configuration options is the main concern of the *performance* meta-model described in Section 5.3. This *performance* view provides a novel degree of freedom that can be exploited by system integrators for refining the overall system's design without conflicting with the component's internal implementations. In this view, *Tasks*—connected through refined interaction links—form task nets where individual loop-free paths starting from sensor-tasks and ending in actuator-tasks are called *cause–effect task chains* (or more shortly *cause–effect chains*), which are important system aspects for the later performance analysis (described in Chapter 6).

5.2. The System Integration Meta-Models

As introduced at the beginning of this chapter, the system-integration phase is subdivided into three separate views, which individually are realized as the three Ecore packages (illustrated in Figure 5.2): *system-configuration* (short *system*), *deployment definition* (short *deployment*) and *performance specification* (short *performance*).

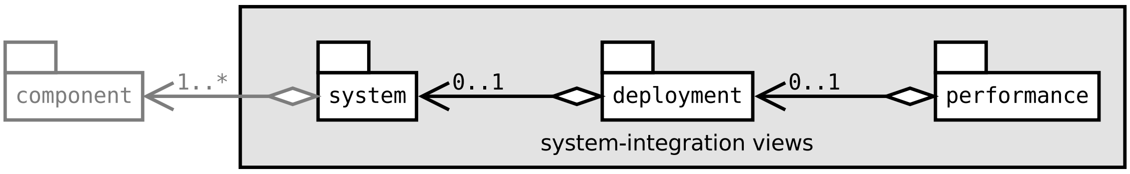


Figure 5.2.: The three system-integration views (realized as Ecore meta-model packages)

The three *system-integration views* (in Figure 5.2) build on one another. Respectively, the overall system model is extended and refined throughout these sequenced views. This section thus is structured as follows. First, Section 5.2.1 presents the *system-configuration* view that allows selecting components that build up a system. In contrast to the component-development step from the previous Chapter 4, *system-configuration* is the first step where the overall system takes shape and all the components are considered in combination (rather than individually). Next, Section 5.2.2 discusses the *deployment* view, which is about adding platform-specific details and mapping the previously specified components to hardware. These first two views can be considered generic and might be available in a similar form in some other frameworks or component models. The idea in this dissertation is not to reinvent these models but to provide a consistent representation and a consistent realization as a basis for the new *performance* view presented in the next Section 5.3. This new view bridges between the system-level views based on components and the lower-level, task-based view without conflicting with the already provided specifications in the preceding models.

5.2.1. Simplified system-configuration meta-model

The section hereinafter presents the *system-configuration* view (see Figure 5.3). The *system-configuration* view is realized as an Ecore meta-model shown in Figure 5.4. This meta-model directly references the *component* meta-model (described in Section 4.2.1). The relevant *system-configuration* Xtext grammar is provided in the Appendix A.3.

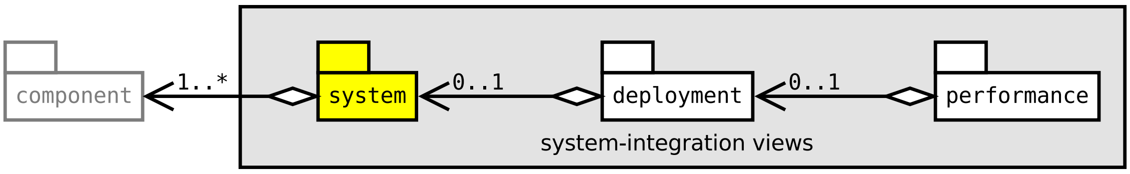


Figure 5.3.: The system-configuration view

The focus of this *system-configuration* view is to define component instances, to specify an initial wiring between components and to refine initial component configurations. Therefore, the *SystemModel* root element in Figure 5.4 creates elements of type *ComponentInstance* and *Connection*. The *ComponentInstance* element instantiates a distinct component, referenced by the *compRef* attribute and specifies a unique instance-name. Instantiating components is reasonable because there might be more than one instance from the same component in a system. For example, in this way it is possible to instantiate two (or more) laser scanner components of the same type for several laser scanners mounted at different places on the robot.

Each *ComponentInstance* also instantiates the *InPorts* and *OutPorts* of the referenced *Component*. From the semantics point of view, instantiating ports is not necessary because the ports cannot be different from those defined in the referenced component model. This is important because specified ports cannot be removed or new ports invented without affecting the component's internal functionality. However, for technical modeling reasons, the *Connection* element demands the existence of port elements as direct children of a *ComponentInstance*. For this reason, a *ComponentInstance* allows creating and composing *InPortInstances* and *OutPortInstances*. *InPortInstances* and *OutPortInstances* are named elements, but their names are automatically derived from the respectively referenced *InPorts* or *OutPorts*. The derived name already ensures that neither an *InPortInstance* nor an *OutPortInstance* can be initialized more than once in a *Compo-*

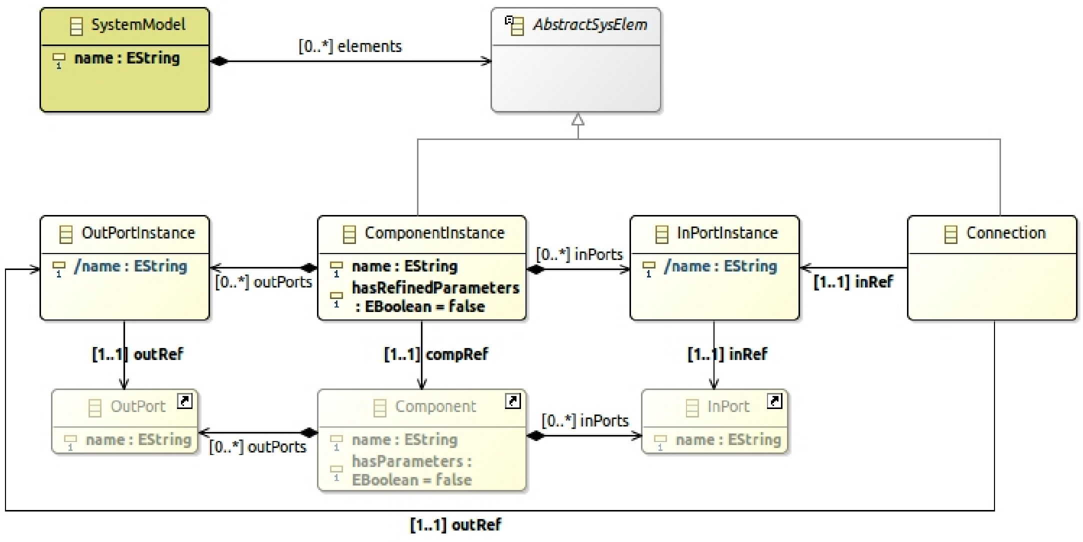


Figure 5.4.: System-configuration Ecore meta-model

nentInstance (which otherwise is automatically detected as a name conflict). However, this does not guarantee that **all** the specified *InPorts* and *OutPorts* of a component are also instantiated within the respective *ComponentInstance*. On the one hand, non-instantiated *OutPorts* are not harmful because if no component in the system ever needs to use data from that *OutPort*, it can also be omitted in the model. On the other hand, if non-optional *InPorts* are not instantiated and thus not connected to a relevant *OutPort*, then the component and thus the system will not function correctly. Unfortunately, Ecore does not provide any means to directly check this constraint. Formally, these two model constraints can be expressed using predicate logic as is demonstrated by the following two equations:

$$\begin{aligned}\forall\, comp \in ComponentInstance : \forall\, task \in comp.compRef.tasks \\ \exists\, inputLink \in task.inputLinks : \neg inputLink.optional \\ \Longrightarrow\ \exists\, inPortInst \in comp.inPorts : inPortInst.inRef = inputLink.inRef\end{aligned} \tag{5.1}$$

$$\begin{aligned}\forall\, comp \in ComponentInstance : \forall\, inPortInst \in comp.inPorts : \\ \exists\, conn \in Connection : conn.inRef = inPortInst\end{aligned} \tag{5.2}$$

While this formalism allows expressing the model constraints, they ultimately need to be integrated and implemented within the meta-model itself. One solution for this problem is to define invariants for the *ComponentInstance* element using the Object Constraint Language (OCL). For such cases, Eclipse provides the *OCLinEcore*[1] editor. Listing 5.1 shows the representation of the *ComponentInstance* element within the *OCLinEcore* editor.

```
   class ComponentInstance extends AbstractSysElem
 2 {
     attribute name : String[1];
 4   property outPorts : OutPortInstance[*] { ordered composes };
     property inPorts : InPortInstance[*] { ordered composes };
 6   property compRef : component::Component[1];
     attribute hasRefinedParameters : Boolean[1] = 'false';
 8   invariant allInPortsInstantiated('At least one non-optional InPort is not
       yet instantiated'):
       self.inPorts->collect(inRef)->includesAll(
10       self.compRef.tasks->collect(inputLinks)->flatten()
           ->select(optional=false)->collect(inRef)
12   );
     invariant allInPortsConnected('At least one InPort is not yet connected'):
14     self.inPorts->forAll(inPort:InPortInstance |
         self.oclContainer().oclAsType(SystemModel).elements
16         ->selectByType(Connection)->exists(inRef=inPort)
     );
18 }
```

Listing 5.1: OCL invariant ensuring that all InPorts are instantiated and connected

Besides of the regular properties and attributes of a *ComponentInstance* (lines 1–7), Listing 5.1 additionally demonstrates (in lines 8–17) the definition of two OCL invariants. The first invariant checks that all non-optional *InPorts* of the referenced *Component* also are instantiated within the *ComponentInstance*. The second invariant checks that each instantiated *InPort* is connected.

Technically, OCL is integrated within Ecore using EAnnotations. Therefore, the actual OCL invariants are stored as strings within the Ecore meta-model without any further syntactical constraints. This has several negative implications. First, the OCL invariants are not visible within the Ecore meta-model diagram (see e.g. Figure 5.4). Second, the actual syntax of the OCL invariants is not checked by a Java compiler. OCL invariants are only checked within the *OCLinEcore* editor and are later directly interpreted in a deployed Eclipse instance. This means that in case where the Java implementation of the meta-model is changed (for any reasons), the Java compiler will not detect inconsistent OCL constraints. Instead, an inconsistent OCL constraint leads to a runtime error in a deployed Eclipse instance. Consequently, this leads to increased accidental complexity and to a tool lock-in because of an additional language and tool. All these reasons motivate another approach. Because the actual model editor is based on Xtext and Sirius, the validation functionality from Xtext and/or Sirius can be used instead. Prior to that, the subtle difference between invariants and the model validation in Xtext/Sirius needs to be highlighted: While the invariants express the positive case (i.e., how the correct model is supposed to be), the

[1]OCLinEcore: `wiki.eclipse.org/OCL/OCLinEcore`

model validations check the negated facts (i.e., they try to find model inconsistencies). Therefore, the two invariants first need to be negated before being implemented as a model validation check. The following Equation (5.3) shows the negated version of first invariant from Equation (5.1), which has been additionally slightly modified to become more implementation friendly.

$$\begin{aligned}\forall\, comp \in ComponentInstance : \forall\, inPort \in comp.compRef.inPorts :\\ \exists\, task \in comp.compRef.tasks : \exists\, inLink \in task.inLinks :\\ inLink.inPort = inPort \wedge \neg inLink.optional\\ \implies \neg\exists\, inPortInst \in comp.inPorts : inPortInst.ref = inPort\end{aligned} \tag{5.3}$$

Listing 5.2 shows the implementation of the model validation check using the Xtend language for the negated invariant from Equation (5.3). This model validation check is part of the actual *system-configuration* Xtext grammar (provided in Appendix A.3).

```
  @Check
2 def checkComponentInstantiatesAllInPorts(ComponentInstance comp) {
    // check whether all non-optional InPorts of a component are instantiated
4   for(in: comp.compRef.inPorts) {
      if(comp.compRef.tasks.exists[
6         it.inputLinks.exists[it.inRef==in && it.optional==false]
        ])
8     {
      // there is at least one Task that has an inputLink to the current InPort
      marked as non-optional
10      if(!comp.inPorts.exists[it.inRef==in]) {
          warning("At least one non-optional InPort is not yet instantiated",
12          SystemPackage.Literals.COMPONENT_INSTANCE__IN_PORTS)
        }
14    }
    }
16 }
```

Listing 5.2: Xtend check that warns if not all InPorts are instantiated

Technically, the model validation check in Listing 5.2 searches for those cases where one of the non-optional *InPorts* (explained further below) is not yet initialized, and if so, generates according warning messages that are attached to the *InPortInstances* list of the corresponding *ComponentInstance* model-element. Only non-optional *InPorts* need to be connected and thus instantiated. The optionality is defined as follows: A *Component* can define several *Tasks*, which may share the same *InPort* through respective *InputLinks*. Each of these *InputLinks* can be either optional or non-optional. If at least one of these *InputLinks* (for the same *InPort*) is non-optional, then this makes the referenced *InPort* altogether as non-optional (regardless of whether other *InputLinks* might be optional). Therefore, the check in Listing 5.2 (in lines 5–7) first needs to check whether there is a relevant *InputLink* that is both non-optional and references the currently checked *InPort*. If so, the current *InPortInstance* is searched within the *InPortInstances* list (in line 10). If not found, a proper warning message is generated (in lines 11–12).

In a similar way, the negated version of the second OCL invariant from Listing 5.1 (and respectively Equation (5.2)) results in the following formalism (that has been slightly extended to include the differentiation between optional and non-optional *InPorts*):

$$\begin{aligned}\forall\, comp \in ComponentInstance : \forall\, inPortInst \in comp.inPorts :\\ \exists\, task \in comp.compRef.tasks : \exists\, inLink \in task.inLinks :\\ inLink.inPort = inPortInst.ref \wedge \neg inLink.optional\\ \implies \neg\exists\, conn \in Connection : conn.inRef = inPortInst\end{aligned} \tag{5.4}$$

Listing 5.2 demonstrates the appropriate Xtend-based implementation.

```
@Check
def checkConnectedInPortInstance(InPortInstance in) {
  val comp = (in.eContainer as ComponentInstance)
  val system = (comp.eContainer as SystemModel)
  if(comp.compRef.tasks.exists[it.inputLinks.exists[it.inRef==in.inRef &&
    it.optional==false]])
  {
    // there is at least one task that uses the current InPort as non-optional
    if(!system.elements.filter(typeof(Connection)).exists[it.inRef==in]) {
      warning("Non-optional InPort " + in.name +
        " seems not yet to be connected to any OutPort",
        SystemPackage.Literals.IN_PORT_INSTANCE__NAME)
    }
  }
}
```

Listing 5.3: Check whether all InPortInstances are connected

The **system-level** semantics of a purely optional *InPort* (i.e., all associated *InputLinks* are optional) is that it can be optionally connected in a system. However, this *InPort* technically becomes non-optional as soon as the system defines an initial connection between that *InPort* and a relevant *OutPort*. This is important, since ports at runtime should not arbitrarily disappear for any reasons. It would be unnecessarily difficult to implement a component that can deal with arbitrary *InPort* disappearance and thus connection losses. There is one exception, namely the *DynamicWiring* pattern (shortly mentioned in Section 4.1.3). This component-coordination pattern allows dynamically changing the wiring of the component's *InPorts* at runtime without provoking system wide failures.

The remaining element in the *system-configuration* model is the EClass *Connection*. A *Connection* allows specifying the initial wiring of an *InPortInstance* with an *OutPortInstance*. One *InPortInstance* can only be connected to exactly one *OutPortInstance* that uses the same *CommunicationObject*. This can be expressed as follows:

$$\forall\, conn \in Connection : \\ \exists!inPort \in conn.in : inPort.dataType = conn.out.dataType \quad (5.5)$$

Yet again, this invariant needs to be negated to be implemented as a model validation check:

$$\forall\, conn \in Connection : \\ \neg\exists!inPort \in conn.in : inPort.dataType = conn.out.dataType \quad (5.6)$$

The negated model check in Equation (5.6) is implemented as an Xtend model check shown in Listing 5.4. The meta-model specifies that several *InPortInstances* can be connected to the same *OutPortInstance*. This implements the classical publish–subscribe communication pattern with one publisher and several subscribers. It is interesting to note that a publisher (or respectively an *OutPortInstance*) can exist without any *InPortInstances* ever connected. However, a non-optional *InPortInstance* cannot exist in a system without a matching and connected *OutPortInstance*.

```
@Check
def checkConnection(Connection conn) {
  // check that inPort and outPort in the connection use the same
    communication object type
  if(conn.inRef.inRef.dataType != conn.outRef.outRef.dataType) {
    error("CommunicationObjects mismatch between the inPort "
      + conn.inRef.name + " and the outPort " + conn.outRef.name,
      SystemPackage.Literals.CONNECTION__IN_REF)
  }

  // check that an inPort is only connected once
  val sysConfig = (conn.eContainer as SystemModel)
  for(other: sysConfig.elements.filter(typeof(Connection))) {
    if(conn != other && conn.inRef == other.inRef) {
      warning("InPort "+conn.inRef.name+" is connected multiple times",
        SystemPackage.Literals.CONNECTION__IN_REF)
    }
  }
}
```

Listing 5.4: Connection Xtend check

A vigilant reader might have noticed one remaining attribute, which has not yet been described, namely the boolean flag *hasRefinedParameters* of the *ComponentInstance* element (see Figure 5.4). This optional flag allows further constraining the component's initial parameters based on the application-specific needs (if necessary). Therefore, an appropriate *parameter* [Sta+16] Xtext grammar is used (which is considered out of scope in this dissertation and thus not further explained). The typical outcome of the *system-configuration* model is an ini-file for each instantiated component, specifying the initial connections, component parameters and other possible component configurations.

5.2.2. Deployment meta-model

The *system-configuration* model mentioned above defines the overall system, independent of any specific platform. This section presents the successive *deployment definition* view (see Figure 5.5) realized as the Ecore meta-model illustrated in Figure 5.6. The relevant Xtext grammar is provided in the Appendix A.4.

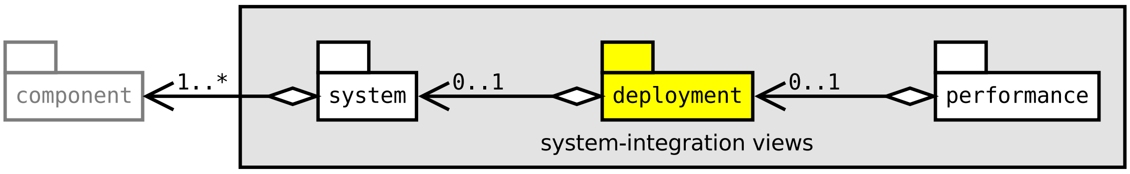


Figure 5.5.: The deployment view

The *deployment* meta-model seeks to provide the computer infrastructure of the target platform and to assign component artifacts to relevant computer devices. This meta-model is closely related to the standard UML deployment model. For each *system-configuration* model several *deployment* models can be defined, thus reflecting potentially different platforms. The root element *DeplyomentModel* in Figure 5.6 needs to be a named element to uniquely distinguish between different platforms. The *system-configuration* model is referenced through the attribute *systemModel* of the *DeplyomentModel* element. The optionality of the *systemModel* is a pure comfort feature that allows defining the devices and the network infrastructure in the *deployment* model independent of a *system-configuration* model. However, to assign relevant component artifacts, a distinct *system-configuration* model needs to be referenced.

The main two elements of a *deployment* model are *Devices* and *NetworkConnections*. A *Device* is a named element that represents a physical computer unit mounted on the actual robot platform. The three mandatory attributes of a *Device* are (i) the *loginName*, (ii) the *deploymentDir* and (iii) the *ip* address. The first two are self-explaining and define (i) the account name of the target computer, which is used as log-in account for deploying the component artifacts and (ii) the deployment directory, which is a system path where the component artifacts are deployed to on the target computer. The third element specifies the main network interface address that should be accessible from the host (i.e., the developer's computer). While a real platform might consist of more than one physical network devices, in practice, usually only one of them is used as the main communication interface, which then calls for a model-based configuration. All the other interfaces are typically configured to statically forward the requests to the main interface and are thus a matter of the overall network infrastructure. For simplicity reasons, only the main interface

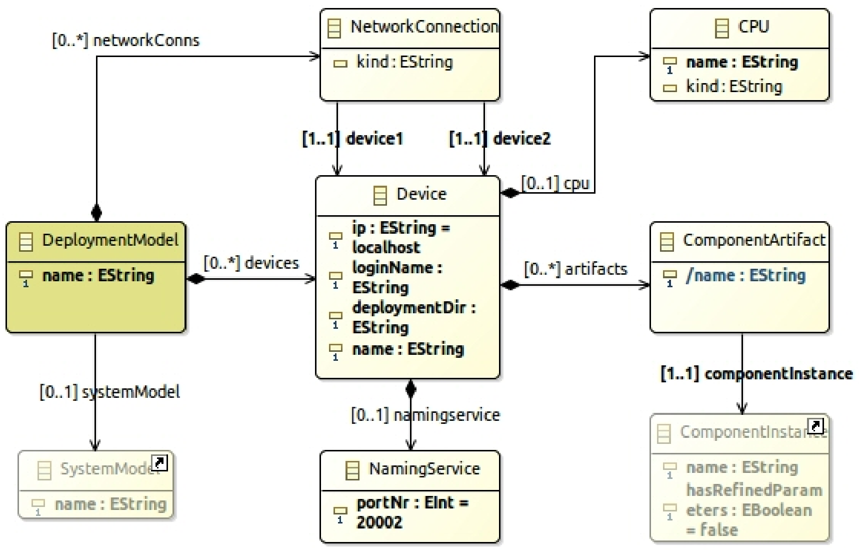


Figure 5.6.: Deployment Ecore meta-model

has been made explicit on model level. However, a proper extension for multiple interfaces is conceptually straightforward.

The two optional attributes of a *Device* are the *CPU* definition and the *NamingService*. The *CPU* definition is a generic element which is not imperative for deploying the component artifacts. This element is a placeholder for the device's main processor and is later referenced within the *performance* model (see next section). It should be noted that one could specify a much more detailed platform definition than that, including several *CPUs*, memory specifications, and further communication-related details. However, this huge amount of details is not (yet) required in typical robotic systems. For this reason and for reducing initial complexity, these details are omitted in the initial meta-model. As before, these details can be easily introduced within the *deployment* model.

The *NamingService* element represents a directory service daemon installed on a device. This directory service translates the logical *OutPort* names of respective components in a *device* into a platform-specific address representation such as a TCP/IP address with a port number. The actual implementation of the directory service depends on the used communication middleware such as e.g. the CORBA naming service or the ACE/SMARTSOFT naming service. Technically, the naming service can be used in two different ways. First, a naming service is installed as a central master on one of the devices and is shared among all other devices in the network. Second, a separate naming service instance is installed on each device (i.e., no central point of failure), thus allowing the individual naming service instances to have a local copy of the directory entries, which are typically automatically synchronized among the individual instances. The latter case is particularly reasonable if the network connection between devices is unreliable e.g. using a wireless network, which then allows the local components to continue working despite network dropouts.

The remaining element is a *ComponentArtifact*, which is a placeholder for the binary representation of a component including everything that this component might require (such as e.g. an ini-file, a grid-map file, etc.) for being deployed to and executed in the target device. The name of the component artifact is derived from the referenced *ComponentInstance* of the accordingly referenced *system-configuration* model, because there is no reason to define yet other names for the components different to what already has been specified within the *system-configuration* model. An additional model check needs to ensure that each *ComponentInstance* is only used once in all artifacts of all devices. The constraint can be expressed mathematically as follows:

$$\begin{aligned}&\forall ar \in Artifact, \forall dev \in Device : \\ &\qquad \neg\exists ar_2 \in dev.artifacts : ar_2 \neq ar \wedge ar_2.compInst = ar.compInst \end{aligned} \tag{5.7}$$

Simply speaking, this constraint states that no two artifacts must use the same *ComponentInstance*. Because the multiplicity cannot be expressed with this kind of statements, the constraint needs to state (slightly more complicated) that there is no artifact different from the currently checked one but uses the same referenced *ComponentInstance*. The negated model validation check in Listing 5.5 just needs to find all those artifacts which reference the same *ComponentInstance*.

```
@Check
def checkArtifact(ComponentArtifact ar) {
  if(ar.eContainer.eContainer instanceof DeploymentModel) {
    val depl = (ar.eContainer.eContainer as DeploymentModel)
    for(dev: depl.devices) {
      for(arOther: dev.artifacts) {
        if(ar!=arOther) {
          if(ar.componentInstance==arOther.componentInstance) {
            warning("Same artifact is deployed to multiple devices",
            DeploymentPackage.Literals.
            COMPONENT_ARTIFACT__COMPONENT_INSTANCE)
          }
        }
      }
    }
  }
}
```

Listing 5.5: ComponentArtifact check

Now all the duplicates have been filtered out, but this does not yet say anything about the completeness in the sense that **all** *ComponentInstances* from the *DeploymentModel* also are used by **any** of the artifacts. Again, this constraint can be expressed mathematically as follows:

$$\forall compInst \in ComponentInstance : \exists dev \in Device : \exists ar \in dev.artifacts : ar.compInst = compInst \quad (5.8)$$

The negated version of this constraint is implemented as the model check shown in Listing 5.6, which checks the absence of an artifact referencing the currently checked *ComponentInstance*.

```
@Check
def checkAllInstancesAreDeployed(DeploymentModel depl) {
  if(depl.systemModel != null) {
    for(comp: depl.systemModel.elements.filter(typeof(ComponentInstance))) {
      if(!depl.devices.exists[
        it.artifacts.exists[it.componentInstance==comp]
      ]) {
        warning("The "+comp.name+" ComponentInstance is not yet deployed "+
        "to any device",
        DeploymentPackage.Literals.DEPLOYMENT_MODEL__SYSTEM_MODEL)
      }
    }
  }
}
```

Listing 5.6: Checking that all ComponentInstances are assigned to a device

One of the interesting aspects about the *deployment* step is its outcome. A *deployment* model can be considered as the last model before the components are deployed to and executed on the actual hardware. In other words, this model handles the actual transition between design-time and runtime. This also is the last step where platform-specific adjustments of components can be made. For instance, to that point components might exist as source code only, or they were compiled without already being linked to a middleware. This means that the actual deployment might involve either a cross compilation of the component binaries, or just linking against the relevant communication middleware and execution environment. In addition, platform-specific start-scripts can be generated that allow easy startup of all involved components on the target. Optionally, the development tool might provide a functionality to copy the binaries to the target device over the network.

5.2.3. Further possible meta-models

So far, two core system-integration views *system-configuration* and *deployment* have been presented. The next section presents the novel *performance* view as a main contribution in this dissertation. It is worth mentioning that these three views individually address some of the fundamentally important concerns during the overall system-integration phase. However, there might be other views that complement this phase. For instance, other common views might be related to a more detailed hardware model of the target system including details related to the used HW devices (the actual sensors and actuators), as well as the physical relations between the HW devices such as the coordinate-system frames. Another important part of system-integration relates to designing and defining action-plots (also called *behavior models*, see [Sta+16]).

Overall, this dissertation does not claim to entirely solve the overall system-integration phase on all levels, nor to provide all encompassing meta-models. Instead, the focus in this dissertation is to provide and discuss all relevant views that are minimally required to design performance related aspects of a system. Each of the presented views separates the individual concerns of the involved developer roles. Thus, the individual complexity is reduced, while the overall system consistency is ensured by construction.

5.3. The Performance Specification View

This section presents the *performance specification* view (see Figure 5.7) realized as the Ecore meta-model illustrated in Figure 5.8. The related Xtext grammar is provided in the Appendix A.5.

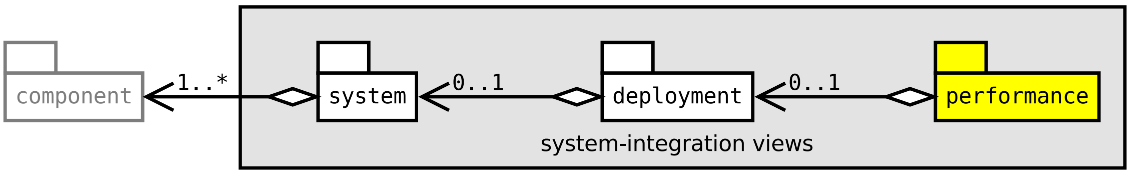


Figure 5.7.: The system-performance view

Before the *performance* view is explained, a subtle relation between the *performance* view and the other two preceding views (see Figure 5.7) deserves further explanations. While the preceding two views use components as the main building blocks, the *performance* view removes the component boundaries and uses the components' internal *Task* specifications as the basic building blocks. This does not contradict with the preceding views for the following reasons. First, one of the main concerns of a component is to cluster (i.e., to bundle) related functional parts. Moreover, functions are wrapped by *Tasks* within a component and offer specific configuration options, thus allowing modification of the functional behavior in a consistent way even if the component has been fully implemented and closed (i.e., compiled to a binary).

This feature can be exploited for injecting performance-related configurations (as explained in Section 5.1.2 and Section 5.1.3) into a component even after the component development phase. This allows modeling and refining the *performance* aspects in a separate view. The refinement of component internal structures without causing inconsistencies is only possible due to the specific

Component meta-model design in Section 4.2 that carefully avoids all those system related design decisions which relate to the here presented *performance* view.

Another delicate aspect is the chosen order between the three system-integration views in Figure 5.7. On the one hand, Section 5.2.2 describes the *deployment* view as the last step before the component artifacts are deployed to the real hardware for execution. Hence, one could infer that no further models can follow *deployment*. On the other hand, the *deployment* view can be further subdivided into the platform definition (with *Devices* and *ComponentArtifacts*) and the actual deployment action (i.e., the copy action that transfers the component binaries and other related artifacts to the actual execution platform). Semantically, the *performance* view would be in between these two deployment sub-steps. However, the *deployment* view has been kept united to keep related concerns together and to prevent too finely grained views. In addition, the actual triggering of the deployment action is a matter of the right tooling ensuring that the potential outcomes (i.e., further artifacts) of the *performance* view are also included in the deployment action. In fact, the outcome of the *performance* view is twofold. On the one hand, *performance* models refine the components' configurations (technically, by providing additional ini-file parameters). On the other hand, *performance* models are the basis for the later Compositional Performance Analysis (CPA) (see next Chapter 6). It is perfectly reasonable to define several alternatives of *performance* models for the same system to experiment with (i.e., to find a good trade-off between) different system designs. Moreover, different configuration sets for relevant runtime system configurations can be simulated and analyzed in this way. All these reasons lead to one conclusion that the *performance* view must follow the *deployment* view and must hook into the relevant deployment action.

The main concern of the *performance* view is to gain control over non-functional system aspects such as responsiveness (see Section 5.1.2 for more details). The overall idea is to make such aspects a deliberate design choice instead of accidental system behavior. Therefore, the *performance* meta-model illustrated in Figure 5.8 combines two related concerns. The first concern is to refine the individual couplings between *InPorts* and *Tasks* of a component by exploiting the purposefully left-open variation points of a component model. The other concern is to specify relevant information about the system so that a Compositional Performance Analysis (see next Chapter 6) can be performed. While these two concerns could also be separated into two separate meta-models, they have a lot in common with many overlaps, which suggest keeping them united. In the end, it is a trade-off between clustering related concerns (thus reducing potentially extremely finely grained views) and splitting up less related concerns (whatever this might mean in detail), thus reducing the complexity of individual views.

The *performance* meta-model illustrated in Figure 5.8 might appear slightly overloaded at first. However, the meta-model is just the basis for real model-editors, who by themselves can reduce model-complexity using e.g. model-layers (see model examples in Chapter 7). Moreover, the meta-model complexity might be further reduced (in future work) by splitting up potentially less related concerns like the definition of activation sources from the definition of task chains (indicated by the two different background colors in Figure 5.8).

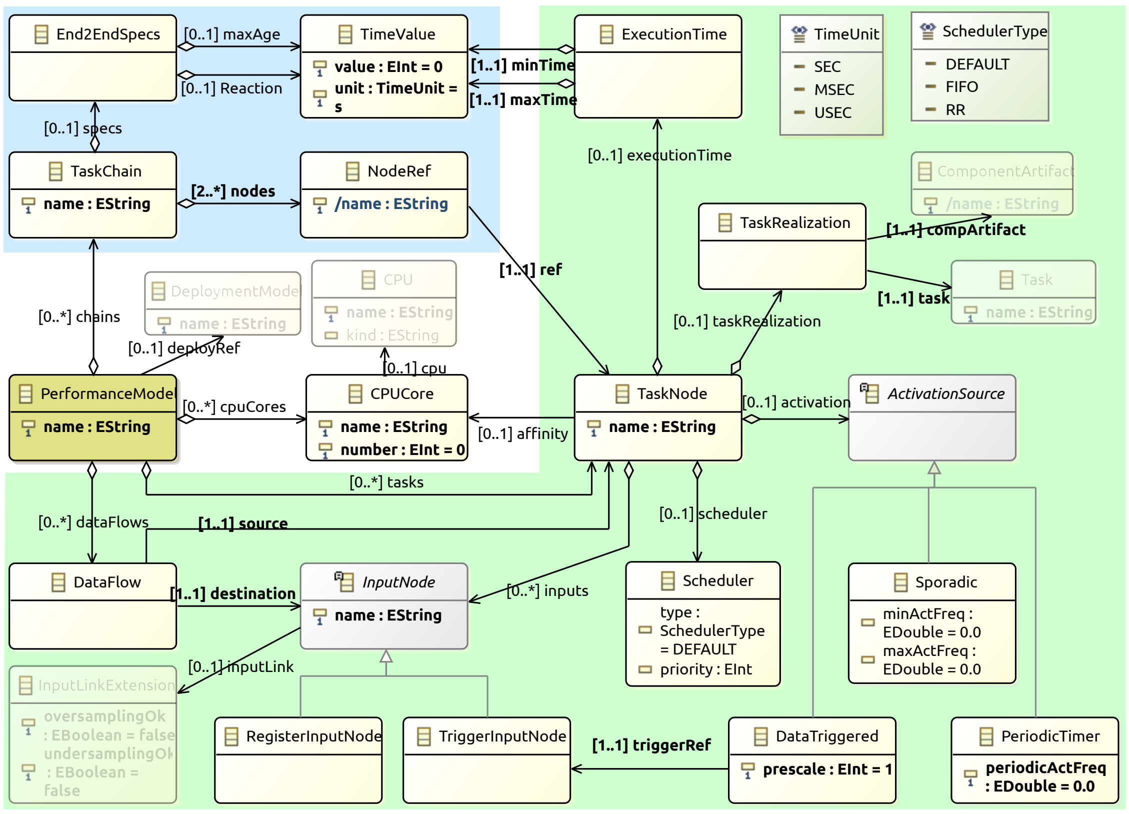


Figure 5.8.: Performance Ecore meta-model

The root element of the *performance* meta-model (in Figure 5.8) is the named element *PerformanceModel*. It can optionally reference a *deployment* model over the *deployRef* attribute. This optionality is important due to the idea of conducting hypothetical system performance analyses without fully specified system models. Hence, system-level performance aspects can be determined and designed early in the overall system integration step in parallel to the actual system specification. While the performance analysis can become more precise the more details about the actually used platform and the selected components become available through relevant *system* and *deployment* models, a hypothetical performance analysis itself can provide additional information and clues for selecting the right components for the current system (thus closing a loop to the beginning of the system integration step).

The child elements of the *PerformanceModel* root element can be clustered into two main groups. One for specifying task nets (described in Section 5.3.1), the other for selecting unique data-flow paths within that task nets also called *cause–effect chains* (see Section 5.3.2). Section 5.3.3 describes the remaining elements with some concluding remarks.

5.3.1. The definition of task nets

This subsection describes the elements related to defining task nets—these are all the elements in the lower and the right half of the Ecore diagram in Figure 5.8 (additionally marked with the light-green background color in the diagram figure). The two main elements are the *TaskNodes* and the *DataFlows*. *TaskNodes* represent individual nodes in a task net. Each *TaskNode* can consist of several *InputNodes*. Now, a *DataFlow* element represents a virtual communication channel between a *TaskNode* and an *InputNode* of another *TaskNode*. A communication channel abstracts away the actual communication technology and hides the crossings of the component's boundaries in between (as explained next).

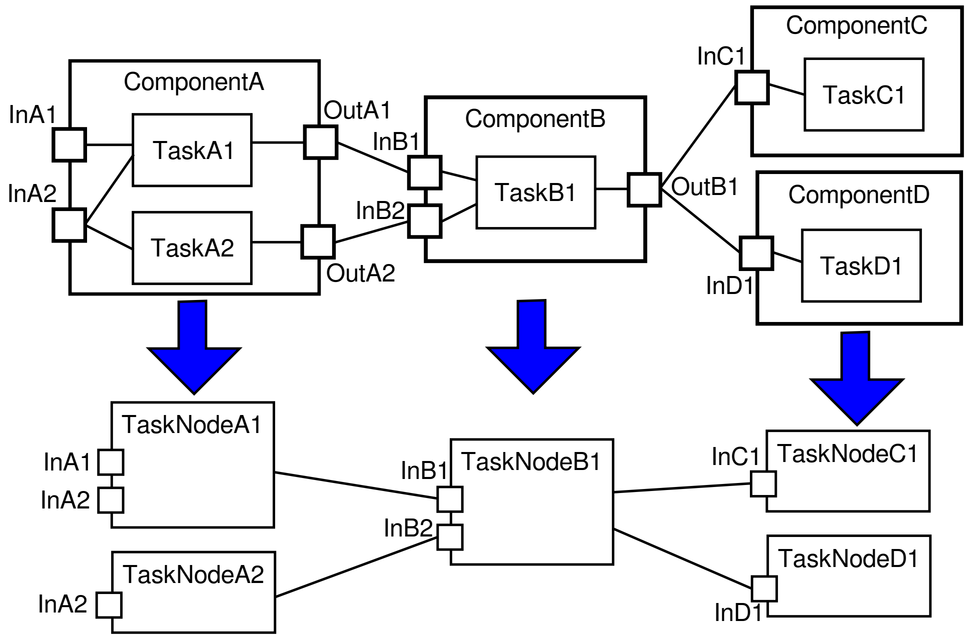


Figure 5.9.: From component models to task net models

Figure 5.9 illustrates the relations between task nets and component specifications. This figure only is a schematic representation used to illustrate the semantic transition and does not represent the actual graphical models (real examples are shown in Chapter 7). In this example, ComponentA (in upper left in Figure 5.9) consists of two *Tasks* TaskA1 and TaskA2, two *InPorts* InA1 and InA2 and two *OutPorts* OutA1 and OutA2. TaskA1 receives data from both *InPorts* InA1 and InA2 and propagates its results through the *OutPort* OutA1. TaskA2 receives data from the *InPort* InA2, thus sharing this *InPort* with TaskA1 and propagates its results through the *OutPort* OutA2. The two rectangles (in lower left in Figure 5.9) show the appropriate representations of the *Tasks* as *TaskNodes* with *InputNodes*. A *TaskNode* thereby semantically combines a *Task* and the associated *OutPort* in one element. An *InputNode* is not so much a representation of an *InPort* but rather the *InputLink* between a *Task* and an *InPort*. For example, both TaskNodeA1 and TaskNodeA2 use the same *InPort* InA2. The names of the *TaskNodes* and the *InputNodes* can be arbitrarily defined independent of the originating *Task*, *OutPort* or *InPort* specifications. This is important for avoiding potential name conflicts, because each component opens up a new namespace for *Tasks*, while all the *TaskNodes* are within the same namespace.

Next, the connection between a *TaskNode* and an *InputNode* is called *DataFlow*. A *DataFlow* semantically skips the intermediate *OutPort* and *InPort* specifications because they are not important for this view. A *TaskNode* can serve as a source for several *InputNodes* such as for example the TaskNodeB1 serves as a source for both InC1 and InD1. However, each *InputNode* must be served by at most one distinct *TaskNode*. At this point, it might appear that the definition of *InputNodes* is redundant compared with the definition of *DataFlows*. However, this redundancy is purposeful and reasonable because an *InputNode* just is an abstract placeholder for either a *RegisterInputNode* or a *TriggerInputNode* (see relevant elements in Figure 5.8). Only a *TriggerInputNode* can be used as an *ActivationSource* for a *TaskNode* (see below) and thus needs to be distinguished.

$$G = (N, E) \text{ where } N = \{N_1, .., N_n\} \text{ and } E = \{N_i \to N_j\} \text{ with } i, j = 1, .., n \text{ and } i \neq j \tag{5.9}$$

$$P = \{e_1, .., e_k\} \text{ where } e_i = \{N_i \to N_{i+1}\} \in E \text{ with } i = 1, .., k \text{ and } k \geq 2 \tag{5.10}$$

Overall, *TaskNodes*, *InputNodes*, and *DataFlows* allow the definition of task nets with branches, forks and even loops over several *TaskNodes*. A task net can be expressed mathematically as a directed graph G (shown in Equation (5.9)) consisting of *TaskNodes* N and *DataFlow* edges E where the edges connect different nodes. Furthermore, individual data-flow paths $p \in P$ (see Equation (5.10)) can be selected as a subset of all the edges E. Task-nets in this form are generally comparable to *Petri Nets* [Mur89]. Thus, *TaskNodes* represent *places*, *InputNodes* represent *transitions* and *DataFlows* represent *arcs*. Moreover, in some cases where for all *TaskNodes* in a task net the *DataTriggered ActivationSource* is selected, this task net additionally realizes the Synchronous Data Flow (SDF) semantics [LM87]. However, since individual *TaskNodes* can also be asynchronously connected (i.e., using the register semantics), the task net is not necessarily an SDF graph. The asynchronous case is more related to *Kahn Process Networks* [Kah74] where the initiating tokens come from a periodic timer rather than from communication. All these graph models (and there are many derivatives form them) have their pros and cons and resulted in different analysis tools for different purposes. Despite all the similarities or differences between these semantics, the main point in this dissertation is to reduce the overall (meta-)model complexity to a reasonable set of useful and sufficient semantics (in accordance with Lee's *freedom-from-choice* philosophy [Lee10]). From analyzing different real-world examples and scenarios in the domain of robotics (such as e.g. the one presented in Section 3.2.1) and from the discussion in Section 4.1.2, two main graph models seem to be sufficient, namely the SDF and a time-triggered

KPN. These two models will be used for activation-source semantics (see below) under the terms *DataTriggered* (for SDF) and *PeriodicTimer* (for time-triggered KPN). As will be shown in a later Chapter 7, these two semantics are sufficient for modeling all the required, performance related system properties of a real-world robotic scenario and for conducting a Compositional Performance Analysis (CPA). Nevertheless, adding further semantics is always possible, yet, increasing the flexibility should always be balanced with reduction of accidental model complexity.

Meta-model name	**original element**		**performance model element**
component	1 *Task*	→	1 *TaskNode*
performExtension	1 *InputLinkExtension*	→	1 *InputNode*
system-configuration	1 *Connection*	→	0..* *DataFlow*

Table 5.1.: Mapping from component and system models to the task net elements

One of the main purposes of the *performance* view is to specify task nets and to annotate properties related to the two data-flow semantics from the preceding paragraph. There are two alternatives how task nets can be specified. First, because there is a direct mapping between the elements in a task net and the elements defined in the preceding system and component models (see Table 5.1), it is possible to pre-generate an entire task net. It is a matter of the right tooling to provide this model-generation step. For instance, Xtext allows implementing code-completion handlers for such purposes. Listing A.6 in the Appendix demonstrates the appropriate completion proposal provider of the *performance* Xtext grammar (in Listing A.5) for generating *TaskNodes* with *InputNodes*. However, this pre-generation step just is an optional comfort feature. The second option involves specifying a task net upfront even before any system or deployment models are known. In this case a hypothetical task net is specified, which can be checked against the preceding model elements as soon as the relevant reference is specified over the *deployRef* attribute of the *PerformanceModel* root element. Having this reference specified, the individual *TaskNodes* can be linked to the relevant *Task* specifications by defining the optional *TaskRealization* elements for *TaskNodes* and by defining the *inputLink* attributes for the *InputNodes*.

The *InputNode* is an abstract element, which derives to either the *RegisterInputNode* or the *TriggerInputNode*. The semantics of the *RegisterInputNode* is that the parent task will always use only the latest available data values each time the next task-cycle is activated (see *ActivationSource* below). This means that all the intermediate values will be ignored that might arrive in-between while the task is in its current cycle. Or, if the update period of the task is higher than the update frequency of the *InputNode* then the task takes the old value for several task cycles until a new data value becomes available. This comes with some consequences for the selection of the *ActivationSource* (discussed further below).

On the contrary, the semantics of a *TriggerInputNode* is to define a potential data-trigger-source, which can be used to trigger the task's cycle activations. This means that each time a new data value arrives on the *TriggerInputNode*, this data value is directly propagated to the task and the task is triggered to execute its next cycle. In case the *InputNode* is linked with a relevant *InputLink* (from the component specification), the *optional* flag of the *InputLink* needs to be checked. Only non-optional *InputLinks* can be used as a *TriggerInputNode* because, as shown in Section 5.2.1, only non-optional *InPorts* need to be really connected and used in a system. In other words, an optional *InPort* might not be used at all and thereby not trigger the potential task, which would lead to undesirable behavior that should be avoided. This constraint can be mathematically expressed as follows:

$$\forall tr \in TriggerInputNode : tr.inputLink = null \vee \neg tr.inputLink.optional \tag{5.11}$$

A corresponding Xtend check in Listing 5.7 prints a warning message for all those model elements where this constraint does not hold (i.e., the negated version—using De Morgan's laws[2]—of the constraint from Equation (5.11)).

```
@Check
def triggerInputNodeWithNonOptionalLink(TriggerInputNode trNode) {
  if(trNode.inputLink != null && trNode.inputLink.optional == true ) {
    warning("TriggerInputNode should not be used with an InputLink defined as
   optional", PerformancePackage.Literals.INPUT_NODE__INPUT_LINK)
  }
}
```

Listing 5.7: Checking that TriggerInputNode is non-optional

The next attribute of a *TaskNode* is the definition of the task's *ExecutionTime*. As mentioned in [Lot+16] there are basically two main ways to define the *ExecutionTime*. The first is to simulate different execution paths of an algorithm, which includes branch prediction and estimation of shared resources to derive the overall Worst-Case Execution Time (WCET). As argued in [Lot+16] this approach is too limiting and too inflexible for robotics. Instead, another approach is to use profiling, i.e., to instrument the source code and to measure the execution times directly on the target platform. This latter approach seems to be much more straightforward and feasible for the robotic cases where algorithms might be heavily complex for a holistic analysis. In some cases, it is also possible to use *anytime* approaches such as e.g. demonstrated in [LSS12]. In any case, providing a realistic *ExecutionTime* is the responsibility of the *performance* model designer who has all the required knowledge about the basic component specifications and the execution context.

While the *ExecutionTime* defines the real calculation time of a task, the task's update-rate is defined differently. Therefore, the *ActivationSource* element of a *TaskNode* allows selection from among three different options—*PeriodicTimer*, *Sporadic*, and *DataTriggered*. In case, the *TaskNode* already references a component's *Task* over the *TaskRealization* element, the choice for the *ActivationSource* is not arbitrary but depends on the *ActivationConstraints* of the referenced task.

[2]De Morgan's laws: https://en.wikipedia.org/wiki/De_Morgan%27s_laws

ActivationConstraints	**allowed ActivationSource**
configurable == **false**	*Sporadic*
configurable == **true**	*PeriodicTimer* or *DataTriggered*

Table 5.2.: Mapping from ActivationConstraints to ActivationSources

More specifically, the *configurable* flag of the *ActivationConstraints* element defines whether *Sporadic* or *PeriodicTimer* and *DataTriggered* can be selected according to Table 5.2. The invariant for this constraint can be expressed as follows (some element names have been shortened for readability reasons):

$$\begin{aligned}
&\forall tnode \in TaskNode : \\
&\qquad (tnode.activation = null \lor tnode.taskRealiz = null) \lor \\
&(\neg tnode.taskRealiz.activConstr.configurable \land tnode.activation = Spradic) \lor \\
&\qquad (tnode.taskRealiz.activConstr.configurable \land \\
&(tnode.activation = PeriodicTimer \lor tnode.activation = DataTriggered)) \qquad (5.12)
\end{aligned}$$

Again, using the De Morgan's laws, the negated version of this constraint can be implemented as a model validation check shown in Listing 5.8 (please note that the access of activation-constraints required an additional, implementation-specific step in between).

```
@Check
def checkActivationSourceFitsToActivationConstraints(TaskNode tnode) {
  if(tnode.activation != null) {
    if(tnode.taskRealization != null
        && tnode.taskRealization.task.taskExtensions.size > 0) {
      val activConstraints = tnode.taskRealization.task.taskExtensions
        .filter(typeof(ActivationConstraints)).head;
      if(activConstraints != null) {
        if(activConstraints.configurable == false) {
          // only Sporadic activation source is reasonable
          if(tnode.activation instanceof DataTriggered
              || tnode.activation instanceof PeriodicTimer) {
            warning("Task realization "+tnode.taskRealization.task.name
              + " defines non-configurable ActivationConstraints,"
              + " therefore the ActivationSource should be Sporadic",
              PerformancePackage.Literals.TASK_NODE__ACTIVATION);
          } // end if(tnode.activation instanceof DataTriggered
        } else {
          // configurable == true => ActivationSource cannot be Sporadic
          if(tnode.activation instanceof Sporadic) {
            warning("Task realization "+tnode.taskRealization.task.name
              + " defines configurable==true ActivationConstraints,"
              + " therefore the ActivationSource should be"
              + " a PeriodicTimer or DataTriggered",
              PerformancePackage.Literals.TASK_NODE__ACTIVATION);
          } // end if(tnode.activation instanceof Sporadic)
        } // end else if(activConstraints.configurable == true)
      } // end if(activConstraints != null)
    } // end if(tnode.taskRealization != null && ...)
  } // end if(tnode.activation != null)
}
```

Listing 5.8: Checking that ActivationSource fits to ActivationConstraints

The case where the flag *configurable* is set to **false** means that the related task implements an internal trigger, whose activation characteristics are unchangeable (see Section 4.2.2 for more details). In this case, the only reasonable choice is the *Sporadic ActivationSource*, which describes hard facts with respect to minimum and maximum activation frequencies (defined in Hertz unit) of that task. The selection of the *Sporadic ActivationSource* has two effects on the parent *TaskNode*. First, the selected minimal and maximal *ExecutionTime* values must be smaller than the inverse of the relevant maximal and minimal activation frequencies (see Equation (5.13)).

$$MinExecTime \leq \frac{1}{MaxActFreq} \wedge MaxExecTime \leq \frac{1}{MinActFreq} \tag{5.13}$$

In other words, the defined *ExecutionTime* must fit within the boundaries defined by the activation frequencies of the *PeriodicTimer* or *Sporadic ActivationSources*. The somewhat lengthy invariant in Equation (5.14) shows the relevant model constraint (again, some names have been shortened for readability reasons).

$$\begin{aligned}
&\forall\, task \in TaskNode : \\
&\quad (task.activ = PeriodicTimer \wedge task.execTime.min \leq \frac{1}{task.activ.perActFreq} \wedge \\
&\qquad\qquad task.execTime.max \leq \frac{1}{task.activ.perActFreq}) \vee \\
&\quad (task.activ = Sporadic \wedge task.execTime.min \leq \frac{1}{task.activ.maxActFreq} \wedge \\
&\qquad\qquad task.execTime.max \leq \frac{1}{task.activ.minActFreq})
\end{aligned} \tag{5.14}$$

The negated version of the invariant from Equation (5.14) is implemented as a model validation check shown in the following (see Listing 5.9).

```
@Check
def checkExecutionTimeFitsInUpdateRate(ExecutionTime et) {
  val task = (et.eContainer as TaskNode)
  if(task.activation != null) {
    if(task.activation instanceof PeriodicTimer) {
      val pt = (task.activation as PeriodicTimer)
      if(getTime(et.minTime) > 1.0/pt.periodicActFreq) {
        error("Minimal ExecutionTime is too high for the selected periodic"
          + " activation frequency of " + pt.periodicActFreq.toString,
          PerformancePackage.Literals.EXECUTION_TIME__MIN_TIME);
      } else if(getTime(et.maxTime) > 1.0/pt.periodicActFreq) {
        warning("Maximal ExecutionTime violates the selected periodic"
          + " activation frequency of " + pt.periodicActFreq.toString,
          PerformancePackage.Literals.EXECUTION_TIME__MAX_TIME);
      }
    } else if(task.activation instanceof Sporadic) {
      val sp = (task.activation as Sporadic)
      if(sp.minActFreq > 0.0 && getTime(et.maxTime) > 1.0/sp.minActFreq) {
        error("Maximal ExecutionTime is too high for the minimal sporadic"
          + " activation frequency of " + sp.minActFreq.toString,
          PerformancePackage.Literals.EXECUTION_TIME__MAX_TIME);
      } else if(sp.maxActFreq > 0.0
          && getTime(et.minTime) > 1.0/sp.maxActFreq ) {
        error("Minimal ExecutionTime is too high for the maximal sporadic"
          + " activation frequency of " + sp.maxActFreq.toString,
          PerformancePackage.Literals.EXECUTION_TIME__MIN_TIME)
      }
    }
  }
}
// this local method converts the TimeValue into seconds of type double
def private double getTime(TimeValue tv) {
  var double result = 0.0
  switch(tv.unit) {
    case TimeUnit::SEC: result=tv.value
    case TimeUnit::MSEC: result=tv.value/1000.0
    case TimeUnit::USEC: result=tv.value/1000.0/1000.0
  }
  return result
}
```

Listing 5.9: Checking that ExecutionTime fits to ActivationSource

The second effect of selecting the *Sporadic* activation source is that all the *InputNodes* of the parent task are used as registers. This means that, at any activation of that task, only the latest data value currently available at each *InputNode* is taken. This again implies that depending on the update frequencies of the relevant *InputNodes* and the actual activation frequencies of the task, either undersampling or oversampling of data values can occur. Whether undersampling and/or oversampling are harmful in the internal implementation of a task is specified by the Boolean attributes *undersamplingOk* and *oversampligOk* of the *ActivationConstraints* element. These two constraints cannot be directly checked at the model creation time, because the update frequencies of the *InputNodes* might not only depend on the preceding *TaskNode* but can be even a propagation of triggering events in a chain of several preceding *TaskNodes*. Therefore, the two constraints can be first checked in the later performance analysis and then additionally be monitored at runtime.

In case the *configurable* flag is set to **true**, one of the other two *ActivationSources* can be selected (see Table 5.2). A *PeriodicTimer* thereby defines, as the name suggests, an internal timer that periodically triggers the next task cycle with the activation-frequency (in Hertz units) defined by the attribute *periodicActFreq*. Similar as with the *Sporadic* element (above), the *ExecutionTime* values must be lower than the inverse of the *periodicActFreq*, and the *InputNodes* of the task (which are used as registers) must again satisfy the over-/undersampling constraints of the relevant *ActivationConstraints*.

Finally, the *DataTriggered ActivationSource* links the update frequency of the parent *TaskNode* to one of its *InputNodes*. Only the *TriggerInputNodes* are allowed as the activation trigger. Because a *TaskNode* can consist of exactly one *ActivationSource*, this implies that exactly one *InputNode* only can be used as a trigger for that task. The definition of more than one *ActivationSource* for a task would be technically possible but might also lead to a hardly predictable execution behavior (as also discussed in Section 4.1.2). Semantically, a *TaskNode* with the *DataTriggered ActivationSource* follows the activation frequency of the preceding *TaskNode* connected through the referenced *TriggerInputNode* and a *DataFlow* link. The activation frequency of the preceding *TaskNode* can be further subdivided by a *prescale* factor defined in the *DataTriggered* element. The preceding *TaskNode* itself might depend on yet another *InputNode's* activation frequency, which again might be subdivided, and so forth. At a certain point, there is one initial *TaskNode* that acts as the first trigger with a defined activation frequency. The selection of the different *ActivationSource* elements for the individual *TaskNodes* in a task net has a direct influence on the overall end-to-end latency and jitter values for individual data messages flowing through the task net (typically starting from certain sensor-tasks, traversing some intermediate filter-tasks, and arriving at certain actuator-tasks; see the next section for further details).

5.3.2. The definition of cause–effect task chains

This subsection describes the elements related to specifying task chains—defined by the three elements *TaskChain*, *NodeRef* and *End2EndSpecs* in the upper left corner of the Ecore diagram in Figure 5.8 (additionally marked with the light-blue background color in the diagram figure). The main concern of task chains is to identify relevant acyclic paths in a task net and then to annotate application-specific constraints related to the overall responsiveness of the system.

The preceding subsection introduced the definition of task nets where individual *TaskNodes* interact with each other through *InputNodes* and *DataFlow* links. In such a task net, many potential paths can be found, arbitrarily starting from one of the *TaskNodes*, traversing a chain of intermediate *TaskNodes*, potentially with infinite loops in-between and finally reaching any arbitrary *TaskNode* acting as an end-node. As mentioned in the Section 5.1.2 and Section 5.1.3, responsiveness is an important aspect to be designed during the system integration phase that needs a deliberate design of response times related to such chains of *TaskNodes*. One compelling approach might be to try to automatically derive unique *TaskNode* paths from the task net, to calculate end-to-end latencies and jitter values for each of these paths and to present the results to the system integrator as an additional source of information about the system. While this approach might appear tempting at first, it also brings with it some unrealistic assumptions. For instance, loops (i.e., cycles) in a task net are natural, unavoidable, and unresolvable in an automatic and generic way. Moreover, as described in Section 3.2.5, responsiveness is neither realized by a single component nor does it impose the same requirements on all the involved components of a system. Nevertheless, responsiveness directly relates to a few clearly identifiable paths in a task net. Such paths start from one particular *TaskNode* (typically implementing a sensor functionality), traversing an acyclic path of intermediate *TaskNodes*, and ending in a specific *TaskNode* (typically implementing an actuator functionality). For a system integrator, it is often very easy and intuitive to identify the relevant paths in a task net. Therefore, a pragmatic solution is just to provide the tools for system integrators to select the paths of interest and to annotate end-to-end timing constraints individually for each of these selected paths.

The meta-model for the selection of paths in a task net is fairly simple, only consisting of three main elements, namely the *TaskChain*, *NodeRef* and *End2EndSpecs* (see upper left corner in Figure 5.8). The *TaskChain* is a named element that consist of at least two *NodeRef* elements. A *NodeRef* element directly selects and derives its name from one of the *TaskNodes* in a task net. Each *TaskNode* can be referenced from several *TaskChains* at the same time, because some paths in a task net might naturally cross the same *TaskNodes*. Now, a task chain is defined by a list of concatenated *NodeRef* elements. It is imperative that successive *NodeRef* elements reference *TaskNodes* that are directly connected through a *DataFlow* link. This dependency can be mathematically expressed as follows:

$$\begin{aligned}\forall\, chain \in TaskChain : \forall\, node \in chain.nodes :& \\ \exists\, flow \in DataFlow : flow.source = node \,\wedge& \\ flow.destination = InputNodeOf(node+1)& \qquad (5.15)\end{aligned}$$

In this invariant, the $flow.destination$ actually references an *InputNode* instead of a *TaskNode*. Therefore, we assume (as a small simplification due to the limited expressiveness of predicate logic) that $InputNodeOf(node + 1)$ returns the matching *InputNode* of the next *NodeRef* in the *TaskChain* list. Just like with the invariants before, it is negated and implemented as an Xtend-based model validation check shown in Listing 5.10. As can be observed, the realization of the $InputNodeOf(\ldots)$ method is implemented as a combination of a "for" loop and the usage of the "findFirst" method in lines 10–11 of Listing 5.10.

```
@Check
def checkReachableNodeRef(NodeRef curr) {
  val chain = (curr.eContainer as TaskChain);
  val perfModel = (chain.eContainer as PerformanceModel);
  val ind = chain.nodes.indexOf(curr);
  if(ind > 0) {
    val prev = chain.nodes.get(ind-1);
    // search for a Flow between PREV.REF and one of the CURR.REF.INPUTS
    var found = false;
    for(input: curr.ref.inputs) {
      val flow = perfModel.dataFlows.findFirst[it.destination==input];
      if(flow != null && flow.source == prev.ref) {
        found = true;
      }
    }
    if(!found) {
      warning("Node " + curr.name +
        " seems not to be connected with its preceding node " + prev.name,
        PerformancePackage.Literals.NODE_REF__NAME);
    }
  }
}
```

Listing 5.10: Checking that NodeRefs are concatenated

In addition to the model validation check in Listing 5.10, a simple Xtext completion proposal provider (shown in Listing A.7 in the Appendix chapter) supports in selecting successive *NodeRef* elements in a *TaskChain*. Because the *NodeRef* element derives its name from the referenced *TaskNode*, this also ensures that no duplicates can be modeled, which in turn ensures that the task chain is free of cycles (which otherwise would result in a name conflict).

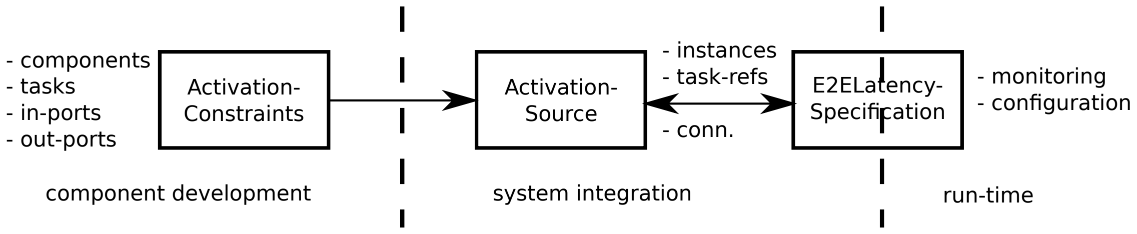


Figure 5.10.: Relation between Activation-Constraints, Activation-Source and End2End latencies

Finally, each *TaskChain* can optionally consist of one *End2EndSpecs* element that specifies—as the name suggests—end-to-end timing requirements for that task chain. There is an interesting relationship between the three elements *ActivationConstraints*, *ActivationSource* and *End2End-Specs*, which are distributed over several views (see illustration in Figure 5.10). In the *component* view, *ActivationConstraints* define boundary conditions for each *Task*, which restrict the selection of relevant *ActivationSource* options in the *performance* view. The combination of selected *ActivationSource* elements for all *TaskNodes* in a task chain directly affects the overall end-to-end latency and jitter for data messages that are propagated in that task chain. The role of

End2EndSpecs thereby is to provide application-specific requirements for that end-to-end latency and jitter values, which can first be checked after the performance analysis is performed (see next Chapter 6). However, any offline analysis only is a more or less accurate (preferably pessimistic) estimation of the real robot's behavior. Therefore, it makes sense to not only use the *End2End-Specs* requirements for the design process alone but in particular to directly influence the runtime execution behavior of the robot by generating and configuring a monitoring infrastructure for individual components (see Section 6.3), which detects and even allows to react to potential time violations without breaking the overall system. Hence, even a system only providing best-effort capabilities can robustly execute its tasks while maintaining (as good as possible) the desired quality in execution.

5.3.3. Further performance-analysis related elements

The remaining two elements in the *performance* view (in Figure 5.8) are the *CPUCore* and the *Scheduler*. A *CPUCore* allows binding of each *TaskNode* to one particular CPU of a *Device*. The *performance* view additionally allows defining several cores for each CPU and assigning each *TaskNode* to a specific core *number*. The modeled dependency is from a *TaskNode* toward a *CPUCore* (not the other way around, which also would be possible). The advantage of this design is that each *TaskNode* can be assigned to at most one particular *CPUCore* and there are no further model checks required. It is then just a matter of the right model-editor tooling to display the *TaskNode* elements attached to the relevant *CPUCore* (see example models in Chapter 7). Interestingly, one could define additional CPU-specific properties such as the CPU architecture. However, it always is a trade-off between providing as much structure and specifications as necessary but not more. For the initial meta-model design, this additional complexity has been excluded. Nevertheless, a relevant extension is conceptually straightforward.

The *Scheduler* element defines—as the name suggests—the scheduling strategy to be used for each individual *TaskNode*. Presently, *FIFO*, *RR* (RoundRobin), and *DEFAULT* scheduler *types* are supported. The latter is a placeholder for any standard scheduler available on the target Operating System (OS) (such as the default scheduler of Linux or Windows). Additionally, the scheduler *priority* can be specified for each *TaskNode* individually. If not specified, the same neutral *priority* is used by default (which again depends on the currently used OS).

5.4. Related Works and Conclusion

This section selectively lists and shortly discusses some approaches related to the overall system-integration phase. Therefore, the following Section 5.4.1 addresses related approaches from the domain of service robotics while the follow-up Section 5.4.2 reaches out for other approaches outside of robotics.

5.4.1. System integration approaches within the domain of service robotics

Within the domain of service robotics, there are a bunch of approaches and projects that address (at least parts of) the overall system-integration challenge. First, the BRICS project[3] developed the BRICS Component Model (BCM) [Bru+13] with the BRIDE and BROCRE tools that aim to ease the search for potential ROS packages that might offer the required functions, which is a difficult problem due to the rather wide diversity of ROS repositories from all over the world. The rather recent ReApp project[4] [Awa+16] has a similar focus but goes even further by providing a so-called integration platform. This integration platform allows annotation of semantic models for individual ROS packages, which eases the search for related and interface-compatible ROS nodes. While these two projects address a relevant problem when developing ROS-based systems, they are too closely related to the ROS environment and tools. In particular, unlike the SmartMDSD component model [Sta+16] (which served as initial work for this dissertation), these approaches lack a sophistic component model that is detailed enough in such a way that the required compositional aspects can be effectively managed independent of the ROS-based semantics. Moreover, they both lack anything related to handling non-functional aspects such as those identified as relevant in this dissertation.

Maybe the only currently available robotic approach that can be considered close to the scope of this dissertation is the Robot Construction Kit (in short Rock) [JA11] with its “oroGen” tool. Rock allows specification of time- and data-triggered activation of components. These semantics are comparable to the *PeriodicTimer* and *DataTriggered* activation sources in this dissertation, but the specification and analysis of effect-chains as such is not provided in Rock.

5.4.2. System integration approaches outside and beyond service robotics

Outside of robotics, two prominent approaches to deal with similar system-level aspects are the Architecture Analysis & Design Language (AADL) [AAD04] and the OMG Modeling And Analysis Of Real-Time Embedded Systems (MARTE) standard [MAR11]. The former originating in the avionics domain, allows specification of safety critical systems and several safety standards use AADL notations. A particularly interesting part of AADL is its flow-latency analysis [Han07] that allows specification and analysis of end-to-end data-flows. Biggs et al. [BFA14] demonstrate in an ad-hock example the usage of AADL flows for an autonomous wheelchair example. While AADL already defines several elements that can also be found in the component meta-model in this dissertation in a similar form, AADL generally lacks robotic-specific, focused modeling views that reflect and cluster the needs of relevant developer roles in respective development phases as shown in this dissertation. In other words, based on the scope of this dissertation, AADL can be considered as a General-Purpose (Modeling) Language (GPML), while the meta-models in this dissertation provide dedicated views that can be used on top of AADL. In this sense, AADL and the meta-models in this dissertation can complement each other in a

[3]`http://www.best-of-robotics.org/`
[4]`http://www.reapp-projekt.de/`

non-conflicting way. The mappings and relations to AADL are therefore conceptually discussed in Chapter 8.

The OMG MARTE standard [MAR11] originated in the embedded domain. It provides a *General Component Model (GCM)* that includes specifications for a flow-port and a client-server port. The flow-port distinguishes between "push" and "pull" semantics. The former is comparable with the *DataTriggered* activation source while the latter defines a passive receiver, which could be used to implement the *PeriodicTimer* semantics (if an additional timer is used to trigger the task). Overall, MARTE provides a very detailed modeling level with various hardware related low-level details. However, it is exactly this huge amount of low-level details that makes MARTE (too) complex and difficult to apply for robotic applications. By contrast, this dissertation provides modeling views on a higher level of abstraction that minimizes the number of choices to an essential set and thus reduces the modeling complexity. Consequently, this allows a simple integration of model analysis tools such as those demonstrated in the follow-up chapter (which is not that straightforward with MARTE).

Another interesting approach from the embedded domain is Sentilles's [Sen12] which coined the term *extra-functional properties*. The general motivation presented by Sentilles is entirely in line with the discussions presented in this dissertation. However, Sentilles came to different conclusions in his work because he addressed a much more broader and more generic context for non-functional system properties. By contrast, this dissertation is much more focused on one specific domain, namely service robotics, which allows to radically narrow the modeling choices and thus to effectively reduce the overall model complexity (which of course comes with the expense of not being as generic and flexible as the approach by Sentilles). This narrow modeling focus is entirely on purpose and allows specification of real-world robotic applications with realistic complexity (as e.g. demonstrated in Chapter 7).

5.4.3. Conclusion

This chapter discussed several aspects of the overall system-integration phase. Thus, three distinct views have been presented, namely system configuration, deployment, and performance specification. These three views are interlinked at the meta-model level and allow specifying different system-level aspects (as discussed in Section 5.1) that altogether define the overall system. While the former two views (presented in detail in Section 5.2) have been inspired by previous works, the latter performance view (presented in Section 5.3) addresses a novel concern during development of service-robotic applications and is the foundation for integrating further analysis tools such as those demonstrated in the follow-up chapter.

“You don’t have to reinvent the wheel, just attach it to a new wagon.”

—Mark McCormack

6 SymTA/S Performance Analysis and Other Runtime Aspects

While the two preceding chapters focused on design-time aspects, this chapter covers three complementary topics related to dealing with dynamic runtime conditions on the robot. In other words, the structural design-time modeling views from the preceding chapters are bridged with dynamic runtime aspects in this chapter. There are basically two options how runtime conditions can be designed and managed. First, they can be simulated and estimated beforehand by e.g. integrating a Compositional Performance Analysis (CPA) into the overall development process (see Section 6.1). Second, the runtime conditions can be observed and examined in an executed system using logging (see Section 6.2) and monitoring (see Section 6.3) mechanisms.

In brief, this chapter is structured as follows. First, Section 6.1 provides some technical details with respect to integrating a Compositional Performance Analysis (CPA) based on the SymTA/S & Trace Analyser tool. A CPA allows simulation and analysis of dynamic runtime condition of a system. This includes a scheduling analysis that considers inter-task dependencies according to system-models that conform to the meta-models presented in the preceding chapter. This again allows designing the dynamic system behavior early as part of the overall robotic development workflow even before the actual target hardware comes to exist (or is built up). Next, Section 6.2 offers some details related to a generic logging infrastructure for capturing ground truth values that are later used (in Chapter 7) to evaluate the presented abstractions and models using a real-world robotic scenario example. Finally, Section 6.3 provides conceptual ideas for designing a monitoring infrastructure that allows capturing all relevant runtime conditions of a system on the fly and that envisions the usage of the captured information by the system for improving the overall execution performance at runtime. A concluding Section 6.4 lists some related works with respect to the three topics in this chapter.

6.1. Integrating the SymTA/S-based Computational Performance Analysis (CPA) into the Overall Robotic Development Workflow

In this dissertation, one of the main objectives is to gain control over specific performance-related aspects of a system (such as responsiveness) by making them an integral part of the overall system design. Such performance aspects can be considered to be highly runtime-specific. This means that at runtime the scheduler decides on the actual order of execution of individual threads and the communication between threads consumes a variable amount of time (depending on the variable amount of data transmitted over a communication channel with limited bandwidth). All this makes it difficult to design performance aspects beforehand. There is an enormous Distributed, Real-time and Embedded (DRE) community that deals with analysing the end-to-end response times and delays. AADL [AAD04] with the *OSATE 2* tool [OSA] and the Symbolic Timing Analysis for Systems (SymTA/S) approach with the Symtavision SymTA/S & Trace Analyser tool[1] are just two prominent examples in this community. Unfortunately, such approaches and tools are not yet common within the domain of robotics. In line with one of the McCormack's credos[2], i.e., not reinventing the wheel for already available and established solutions, it is much more expedient to integrate such tools into the general robotic development workflow and to ease their usage by automating the transition from structural models toward the input models for such performance analysis tools rather than inventing yet another custom solution. This is also stated as an overall Objective 1.4 in Chapter 1. The structures presented in this dissertation allow a flexible usage of different analysis tools such as the SymTA/S & Trace Analyser (as demonstrated hereinafter) and *OSATE 2* [OSA] (as conceptually shown in Chapter 8).

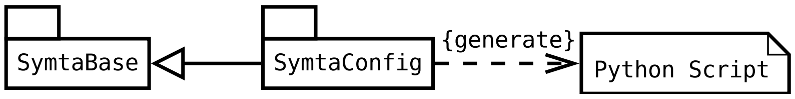


Figure 6.1.: SymTA/S artefacts

The focus in this section is to discuss the mapping between the models from the *performance* view presented in Section 5.3 and the performance analysis based on SymTA/S [Hen+05]. This mapping is realized by two Ecore meta-models *SymtaBase* and *SymtaConfig*, and a generated Python script (see Figure 6.1). Section 6.1.1 presents an automated model-to-model transformation step from the models in the *performance* view into *SymtaBase*. Then, Section 6.1.2 presents additional elements in the *SymtaConfig* meta-model, which are specific to the SymTA/S performance analysis. Finally, Section 6.1.3 gives some insights into the resulting project configuration within the SymTA/S & Trace Analyser tool.

[1]SymTA/S & Trace Analyser: www.symtavision.com/products/symtas-traceanalyzer/
[2]See McCormack quote at the beginning of this chapter.

6.1.1. SymtaBase meta-model and the model-to-model transformation

This section presents the *SymtaBase* meta-model (see Ecore diagram in Figure 6.2). *SymtaBase* serves as the interfacing part between the *performance* view (presented in Section 5.3) and the SymTA/S performance analysis based on the SymTA/S & Trace Analyser tool. While the SymTA/S & Trace Analyser tool is a fully-fledged, Eclipse-based workbench that can be used independently, the tool can also be operated over a Python API. The integration of the SymTA/S & Trace Analyser tool into the overall robotic workflow as well as into the Eclipse-based toolchain is realized using the Python API. In general, there are two main alternatives for using this Python API. First, one could implement a direct model-to-text transformation by generating a Python script, which creates a project within the SymTA/S & Trace Analyser tool and initializes all the project elements according to an input model based on the *performance* view (from Section 5.3). However, this approach easily leads to an inconsistent use of the Python API because any API changes are not detected in the model-to-text transformation step. Ultimately, the result from the model-to-text transformation just is plain ASCII text without any particular syntax. Besides, the Python API is rather extensive with lots of extra features that are not required within the scope of this dissertation. For these reasons, in this section some of the API's core elements are abstracted away and formalized in a separate meta-model. This meta-model is divided into two parts *SymtaBase* (described in the following) and *SymtaConfig* (described next in Section 6.1.2). The first part *SymtaBase* provides only those elements and attributes which can be directly and automatically derived from the preceding model from the *performance* view. The advantage of this approach is that, on the one hand, the *SymtaBase* meta-model can be kept consistent with the Python API independent of the other meta-models. On the other hand, any potential inconsistencies between the *SymtaBase* and the *performance* meta-models are automatically detected within the model-to-model transformation as a compiler error that allows fixing the inconsistencies early, before deploying and using the developed Eclipse plugins.

Figure 6.2 shows the *SymtaBase* Ecore meta-model diagram. The root element called *SVWorkbench* represents a new project (with the provided name) within the SymTA/S & Trace Analyser tool. The root element within a project is a SymTA/S system, represented by the element *SymtaSystemDef*. In theory, several systems per project could be created; however, for reasons of simplicity and clarity, no more than one system is created for each separate project. Within a SymTA/S system different elements can be created. From the perspective of this dissertation, the following four elements are relevant: (i) a *CoreDef* representing a certain CPU core, (ii) a *TaskDef* representing a physical task in a system, (iii) a *TriggerDef* referring to a data-trigger source (see below), and finally (iv) a *PathDef* referring to a task-chain sequence.

The actual model-to-model transformation is implemented with the Xtend language (see Listing 6.1 below) using the factory methods from the *SymtaBase* Ecore meta-model definition for creating new *SymtaBase* meta-model element-instances. Table 6.1 additionally summarizes the core mappings of the model-to-model transformation.

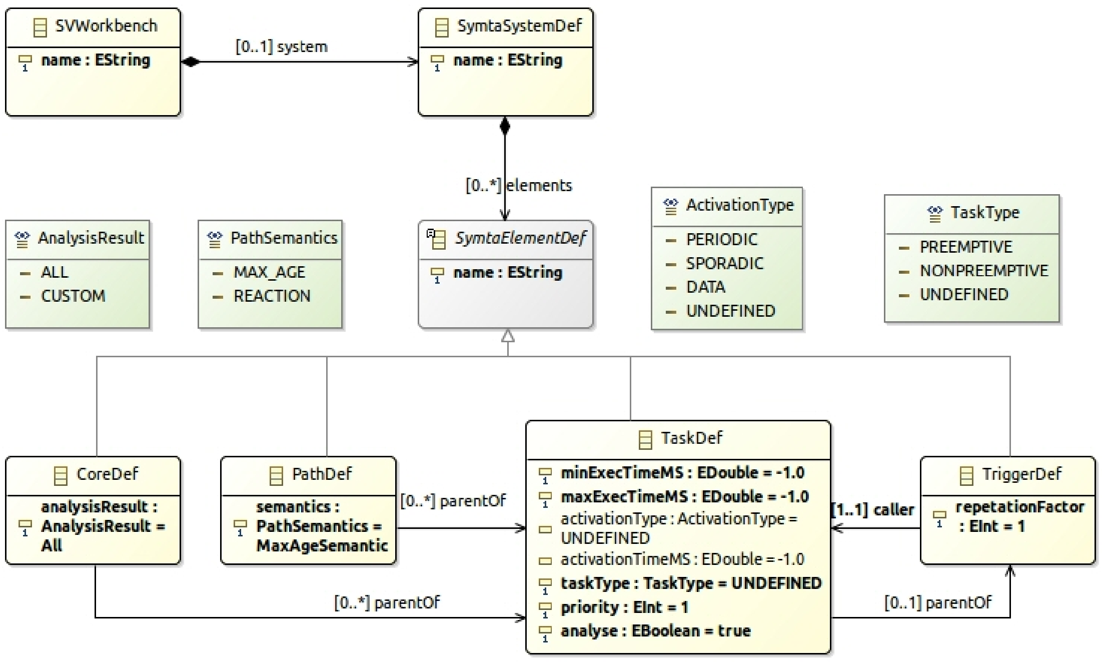


Figure 6.2.: SymTA/S Base Ecore meta-model

performance model element	**performance model attribute**	**SymtaBase model element**	**SymtaBase model attribute**
PerformanceModel	name	SVWorkbench	name
PerformanceModel	name	SymtaSystemDef	name
TaskNode	name	TaskDef	name
ExecutionTime	minTime (in ms)	TaskDef	minExecTimeMS
ExecutionTime	maxTime (in ms)	TaskDef	maxExecTimeMS
Scheduler	priority	TaskDef	priority
Scheduler	type != DEFAULT	TaskDef	analyse=true
PeriodicTimer	1000 / periodicActFreq	TaskDef ActivationType	activationTimeMS PERIODIC
Sporadic	1000 / maxActFreq	TaskDef ActivationType	activationTimeMS SPORADIC
DataTriggered	–	ActivationType	DATA
DataTriggered	eContainer.name+ ”Trigger”	TriggerDef	name
DataTriggered	prescale	TriggerDef	repetitionFactor
DataFlow	source	TriggerDef	caller
CPUCore	name	CoreDef	name
TaskNode*	affinity	CoreDef	parentOf*
TaskChain	name	PathDef	name
TaskChain	nodes*	PathDef	parentOf*
PreemptiveTask	–	TaskType	PREEMPTIVE
CooperativeTask	–	TaskType	NONPREEMPTIVE

Table 6.1.: Mapping from the *performance* meta-model to the *SymtaBase* meta-model

```
/**
 * This method does the actual Model2Model transformation: PerformanceModel =>
     SymtaBase
 * Inconsistencies between the two Ecore meta-models will be detected through
   compiler errors
 */
def transformIntoSymtaBaseModel(PerformanceModel perfModel) {
  // create SVWorkbench model element
  val svWorkbench = SymtaBaseFactory.eINSTANCE.createSVWorkbench;
  svWorkbench.name = perfModel.name;
  // create SymtaSystemDef model element
  val symtaSystem = SymtaBaseFactory.eINSTANCE.createSymtaSystemDef;
  symtaSystem.name = perfModel.name;

  var taskDefList = new ArrayList<TaskDef>();
  var triggerDefList = new ArrayList<TriggerDef>();

  // create PathDefs and TriggerDefs out of TaskNodes with
  this.createTaskDefsAndTriggerDefs(perfModel, symtaSystem, taskDefList,
    triggerDefList);

  // add all Task definitions to the SymtaSystemDef model element
  symtaSystem.elements.addAll(taskDefList);

  // set-up the caller dependencies from TriggerDefs to according TaskDefs
  this.setUpCaller(perfModel, taskDefList, triggerDefList);

  // add all Trigger definitions to the SymtaSystemDef model element
  symtaSystem.elements.addAll(triggerDefList);

  // create PathDefs from TaskChains and add them to SymtaSystemDef
  this.createAndAddPathDefs(perfModel, symtaSystem, taskDefList);

  // create CoreDefs out of CpuCores and add them to SymtaSystemDef
  this.createAndAddCoreDefs(perfModel, symtaSystem, taskDefList);

  // assign SymtaSystemDef to SVWorkbench
  svWorkbench.system = symtaSystem;

  // return the completed/transformed SVWorkbench model element
  return svWorkbench;
}
```

Listing 6.1: Performance => SymtaBase Model2Model Transformation

The root element *PerformanceModel* (from the *performance* view) results in a new *SVWorkbench* and a new *SymtaSystemDef* elements, both with the name of the *PerformanceModel* (see lines 7–11 in Listing 6.1). Thereafter, the *SymtaSystemDef's* children elements of type *TaskDef*, *TriggerDef*, *CoreDef* and *PathDef* with their relevant attributes are created and configured in separate methods (indicated by the keyword `this`), which are individually described in the follow-up listings below.

```
def private void createTaskDefsAndTriggerDefs(
  PerformanceModel perfModel, SymtaSystemDef symtaSystem,
  List<TaskDef> taskDefList, List<TriggerDef> triggerDefList)
{
  for(task: perfModel.tasks) {
    val taskDef = SymtaBaseFactory.eINSTANCE.createTaskDef;
    // set-up task-name
    taskDef.name = task.name;
    // set-up task priority
    if(task.scheduler != null) {
      taskDef.priority = task.scheduler.priority;
      if(task.scheduler.type!=SchedulerType.DEFAULT) {
        taskDef.analyse = true;
      } else {
        taskDef.analyse = false;
      }
    } else {
      // default
      taskDef.priority = 0;
      taskDef.analyse = false;
    }
    // get min/max values from the executionTime element
    val execTime = task.executionTime;
    if(execTime != null) {
      taskDef.minExecTimeMS = execTime.minTime.timeInMS;
      taskDef.maxExecTimeMS = execTime.maxTime.timeInMS;
    } else {
      // default values
      taskDef.minExecTimeMS = 0.0;
      taskDef.maxExecTimeMS = 0.0;
    }

    // set-up activation type of TaskDef and create a TriggerDef if needed
    this.setUpActivation(task, taskDef, triggerDefList);

    // set-up TaskType
    val taskRealization = task.taskRealization;
    if(taskRealization != null) {
      if(taskRealization.task instanceof CooperativeTask) {
        taskDef.taskType = TaskType.NONPREEMPTIVE;
      } else {
        taskDef.taskType = TaskType.PREEMPTIVE;
43      }
      } else {
45        // this is the default
        taskDef.taskType = TaskType.PREEMPTIVE;
47    }

49    // add the current TaskDef to the local list
    taskDefList.add(taskDef);
51  }
}
```

Listing 6.2: M2M Transformation: Create TaskDefs and TriggerDefs

For each *TaskNode* element from the *performance* model, a *TaskDef* element as a child of the parent *SymtaSystemDef* is created (lines 5–8 in Listing 6.2). The definition of *TaskDefs* is the basis for anything else in a SymTA/S system.

In case a *Scheduler* element is defined for a *TaskNode*, the *Scheduler's priority* value is assigned to the *TaskDef's priority* attribute (lines 10–21 in Listing 6.2) where higher values refer to higher priorities. Moreover, in case the *Scheduler* is of any *type* different to DEFAULT, the *TaskDef* is configured to be considered in a later performance analysis. The reason for this flag is that some *TaskNodes* might be defined for the sole purpose to specify their *ActivationSource* but not for analysing (nor specifying) their exact execution behavior. This is particularly reasonable for those *TaskNodes* that are not time-critical and thus executed with best-effort only in the spare time not used by the real-time scheduler.

The *TaskDef's* minimal and maximal execution times are defined by the respective attributes *minExecTimeMS* and *maxExecTimeMS* (lines 23–31 in Listing 6.2), which are derived from the *TaskNode's executionTime* attribute with its *minTime* and *maxTime* values. Both original time-values need to be converted into the milliseconds unit before being used. The *taskType* attribute of a *TaskDef* is set according to the original type where a *PreemptiveTask* results in PREEMPTIVE and a *CooperativeTask* results in NONPREEMPTIVE types (lines 37–47 in Listing 6.2).

```
def private void setUpActivation(TaskNode task, TaskDef taskDef,
  List<TriggerDef> triggerDefList)
{
  val activation = task.activation;
  if(activation != null) {
    if(activation instanceof DataTriggered) {
      val dataTriggered = (activation as DataTriggered);
      val triggerDef = SymtaBaseFactory.eINSTANCE.createTriggerDef;
      // set the name of the trigger to the InputNode.name
      triggerDef.name = taskDef.name+"Trigger";
      // repetationFactor = DataTriggered.prescale
      triggerDef.repetationFactor = dataTriggered.prescale;
      // define task to be parentOf this trigger
      taskDef.parentOf = triggerDef;
      // store trigger in local trigegrList
      triggerDefList.add(triggerDef);
18     // default values
       taskDef.activationType = ActivationType.DATA;
20    } else if(activation instanceof PeriodicTimer) {
       val periodicTimer = (activation as PeriodicTimer);
22     // activationTimeMS = (1/activationFreqHZ)*1000
       taskDef.activationTimeMS = 1000.0/periodicTimer.periodicActFreq;
24     taskDef.activationType = ActivationType.PERIODIC;
      } else if(activation instanceof Sporadic) {
26     val sporadicTimer = (activation as Sporadic);
       // activationTimeMS = (1/activationFreqHZ)*1000
28     taskDef.activationTimeMS = 1000.0/sporadicTimer.maxActFreq;
       //minActFreq is not relevant for the SymTA analysis and can be skipped
30     taskDef.activationType = ActivationType.SPORADIC;
      }
32   } else {
      // default values
34    taskDef.activationType = ActivationType.PERIODIC;
      taskDef.activationTimeMS = 0.0;
36   }
    }
```

Listing 6.3: M2M Transformation: Set-up ActivationType for a TaskDef and (if needed) create a TriggerDef

Each *TaskDef* can be triggered either by an external *TriggerDef* (which is *called* by another *TaskDef*), or by an internal *ActivationType* (which can be either PERIODIC or SPORADIC). Both trigger kinds are mutually exclusive. At this point, it is worth mentioning that this exclusiveness is not well represented in the *SymtaBase* meta-model. On the one hand, it would be technically possible to modify the meta-model in such a way that an abstract base element over the *TriggerDef* and over a new element for the internal trigger-types ensures that only one of the different trigger-types can be defined for each *TaskDef*. On the other hand, this modified meta-model would not well represent the SymTA/S Python API any more. Ultimately, it is a trade-off between directly formalizing the API and aligning the two meta-model semantics. Interestingly, as long as the original *performance* meta-model and the model-to-model transformation are both consistent (which they are), the generated *SymtaBase* model also becomes consistent (even if it theoretically allows inconsistent definitions). For this reason, the *SymtaBase* meta-model is kept closer to the API rather than closer to the *performance* meta-model semantics.

The *TriggerDef* element is derived from the *DataTriggered ActivationSource* (lines 6–19 in Listing 6.3). Like any other element derived from *SymtaElementDef*, the *TriggerDef* as well requires a unique name. Unfortunately, the *DataTriggered* element does not have (and does not need) a name. Because the *TriggerDef* is assigned to the respective *TaskDef* through the *parentOf* reference anyway (in line 14), it makes sense to name the *TriggerDef* similar (but not equal) to its parent *TaskDef*. Therefore, the name for the *TriggerDef* element is constructed from the parent *TaskDef* name followed by the postfix “Trigger” (see line 10 in Listing 6.3).

The *repetitionFactor* of a *TaskDef* is directly derived from the *prescale* attribute of the related *DataTriggered* element (in line 12). Semantically, a *TriggerDef* concatenates two *TaskDefs* and propagates the activation frequency (subdivided by the *repetitionFactor*) from the preceding *TaskDef* (specified as the *caller* of the *TriggerDef*) toward the follower *TaskDef* (specified by its *parentOf* relation). The *DataTriggered* activation source further results in the *TaskDef's activationType* set to DATA (line 19).

The mapping from a *PeriodicTimer* or a *Sporadic ActiviationSources* toward the relevant trigger kind for a *TaskDef* is rather straightforward (lines 20–31 in Listing 6.3). For a *PeriodicTimer*, the *activationType* of the *TaskDef* is set to PERIODIC and the *activationTimeMS* is calculated by the following formula:

$$activationTimeMS = \frac{1000}{periodicActFreq} \tag{6.1}$$

Similarly, the *Sporadic ActivationSource* results in the *activationType* set to SPORADIC and the *activationTimeMS* is calculated by the formula:

$$activationTimeMS = \frac{1000}{maxActFreq} \tag{6.2}$$

The *minActFreq* of the *Sporadic* element is ignored at the moment due to its irrelevance for the worst-case analysis. However, this *minActFreq* could be added as jitter (not yet modeled) to the *TaskDef*'s *activationTime*, thus improving the accuracy of the best-case analysis. This improvement is postponed for future works.

```
def private void setUpCaller(PerformanceModel perfModel,
  List<TaskDef> taskDefList, List<TriggerDef> triggerDefList)
{
  // set-up the caller dependencies: TriggerDef.caller = TaskDef
  for(dataFlow: perfModel.dataFlows) {
    if(dataFlow.destination instanceof TriggerInputNode) {
      val taskDef = taskDefList.findFirst[it.name==dataFlow.source.name];
      if(taskDef != null) {
        val taskNode = (dataFlow.destination.eContainer as TaskNode);
        val triggerName = taskNode.name+"Trigger";
        val triggerDef = triggerDefList.findFirst[it.name==triggerName];
        // there might be no trigger in case that a task has T
        // riggerInputNodes but no DataTriggered activation source
        if(triggerDef != null) {
          triggerDef.caller = taskDef;
        }
      }
    }
  }
}
```

Listing 6.4: M2M Transformation: Set-up the caller attribute for all TriggerDefs

To derive the *caller* attribute of a *TriggerDef* element, an appropriate *DataFlow* element needs to be found that connects the related *TaskNode* over a *TriggerInputNode* to a preceding *TaskNode* (see Listing 6.4). The *source* value of the *DataFlow* then is the *caller* of the *TriggerDef* element. This completes the creation of all *TaskDefs* and *TriggerDefs* including their mutual references (lines 20 and 26 in Listing 6.1 show their assignment to the parent *SymtaSystemDef*).

```
   def private void createAndAddPathDefs(PerformanceModel perfModel,
 2   SymtaSystemDef symtaSystem, List<TaskDef> taskDefList)
   {
 4   // for each TaskChain element create a PathDef element
     for(chain: perfModel.chains) {
 6     val pathDef = SymtaBaseFactory.eINSTANCE.createPathDef;
       // set-up pathDef-name
 8     pathDef.name = chain.name;
       // set-up PathSemantics
10     if(chain.specs != null) {
         if(chain.specs.maxAge != null) {
12         pathDef.semantics = PathSemantics.MAX_AGE;
         } else if(chain.specs.reaction != null) {
14         pathDef.semantics = PathSemantics.REACTION;
         } else {
16         // default
           pathDef.semantics = PathSemantics.MAX_AGE;
18       }
       } else {
20       // default
         pathDef.semantics = PathSemantics.MAX_AGE;
22     }
       for(taskRef: chain.nodes) {
24       val taskDef = taskDefList.findFirst[it.name==taskRef.name];
         if(taskDef != null) {
26         pathDef.parentOf.add(taskDef);
         }
28     }
       // add the Path definition to the SymtaSystemDef model element
30     symtaSystem.elements.add(pathDef);
     }
32 }
```

Listing 6.5: M2M Transformation: Create PathDefs

SymTA/S specifies dependencies between tasks by event-streams. This means that a task can serve as trigger for another task along a sequence of tasks. Such a sequence is defined by the *PathDef* element. Therefore, each *TaskChain* element from a *performance* model leads to a new *PathDef* element with the same name (lines 5–8 in Listing 6.5). The *semantics* attribute of *PathDef* (lines 10–22) allows choosing between one of the two *PathSemantics*: *MaxAge* (default) and *Reaction*. *MaxAge* refers to the maximal overall signal delay in a chain of tasks on a specific CPU core taking all intermediate delays, jitters, and periods into account. *Reaction* is more restrictive with the focus on sporadic (i.e., non-periodic) events where the first reaction time is more important rather than the cyclic delay. The *parentOf* list of *PathDef* specifies which *TaskDefs* (from the original *TaskChain*) make up the actual event-streams (see lines 23–28).

```
   def private void createAndAddCoreDefs(PerformanceModel perfModel,
 2   SymtaSystemDef symtaSystem, List<TaskDef> taskDefList)
   {
 4   // add a Core definition for each CPUCore in the Performance Model
     for(core: perfModel.cpuCores) {
 6     val coreDef = SymtaBaseFactory.eINSTANCE.createCoreDef;
       // the name is set to the "kind" value of the CPU element
 8     coreDef.name = core.name;

10     // set the performance analysis results depending on
       // the selected Scheduler types
12     if(perfModel.tasks.exists[it.scheduler==null
         || it.scheduler.type==SchedulerType.DEFAULT])
14     {
         // if there is at least one defined TaskNode that
16       // either did not specify a Scheduler or has a
         // Scheduler of type default, then the custom
18       // analysis needs to be performed
         coreDef.analysisResult = AnalysisResult.CUSTOM;
20     } else {
         coreDef.analysisResult = AnalysisResult.ALL;
22     }

24     // set the parent-of dependency for all relevant tasks
       for(task: perfModel.tasks.filter[it.affinity == core]) {
26       val taskDef = taskDefList.findFirst[it.name==task.name];
         if(taskDef != null) {
28         coreDef.parentOf.add(taskDef);
         }
30     }
       symtaSystem.elements.add(coreDef);
32   }
   }
```

Listing 6.6: M2M Transformation: Create CoreDefs

The last core element type of a *SymtaSystemDef* is *CoreDef*. For each original *CPUCore* element from the *performance* model, a new *CoreDef* element with the same name is created (lines 5–8 in Listing 6.6). The *CoreDef*'s attribute *analysisResult* (lines 12–22) specifies whether all modeled *TaskDefs* need to be considered in the SymTA/S analysis (indicated by the *ALL* value), or if some of the specified *TaskDefs* need to be skipped (indicated by the *CUSTOM* value). Which *TaskDefs* will be skipped in the *CUSTOM* case depends on the Boolean value of their individual *analysis* attribute. The *parentOf* relation (lines 25–30 in Listing 6.6) between a *CoreDef* and several *TaskDefs* is contrary to the original *affinity* relation between a *TaskNode* and a *CPUCore*.

Therefore, the list of *TaskNodes* needs to be filtered for each *CPUCore* separately. Thereafter, the relevant *TaskDefs* (found by the name) are assigned to the *parentOf* reference list of the related *CoreDef* element.

Finally, the *SymtaSystemDef* element is assigned to the root element *SVWorkbench* (in line 35 in Listing 6.1), which completes the model-to-model transformation. Technically, the *SVWorkbench* element provides an in-memory representation of a *SymtaBase* model-instance. To make this model persistent the default `SerializerFragment`[3] form the *SymtaBase* Xtext grammar (see Listing A.8) is used.

6.1.2. SymtaConfig meta-model

The previous subsection presented the mapping between the *performance* and the *SymtaBase* meta-models in an automated model-to-model transformation step. Apart from those elements that can be automatically derived in a model-to-model transformation step, the SymTA/S analysis requires additional configuration attributes that are provided by the *SymtaConfig* meta-model presented in the following. Therefore, *SymtaConfig* extends the *SymtaBase* meta-model by derivation (see Figure 6.3).

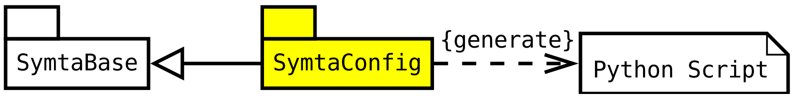


Figure 6.3.: SymtaConfig package

Figure 6.4 shows the *SymtaConfig* Ecore meta-model diagram. The grayed-out elements represent the original (i.e., imported) *SymtaBase* meta-model elements. *SymtaConfig* extends two original elements: *SymtaSystemDef* and *CoreDef* by the two refined elements: *SymtaSystemConfig* and *CoreConfig*.

SymtaSystemConfig extends *SymtaSystemDef* by the *numberOfTraces* and the *traceTimeMS* attributes. *numberOfTraces* refers to the number of simulation rounds that a SymTA/S analysis will execute while simulating different execution times for tasks within their min/max boundaries and simulating the runtime scheduler. *traceTimeMS* refers to the desired maximal length of one trace-run (which limits computational resources for performing the analysis). The default values for these two attributes have been determined empirically.

Next, the CPU core is extended by a few SymTA/S-specific analysis attributes within the *CoreConfig* element. The *scheduler* attribute probably has the highest impact on the performance analysis accuracy, and on whether the performance analysis will reasonably reflect the real system's execution behavior. For the default case, the selection "GenericOSEK" offers the best approximation for many common Linux-based robotic systems that can use schedulers such as FIFO and RoundRobin.

[3]Serialization: https://eclipse.org/Xtext/documentation/303_runtime_concepts.html

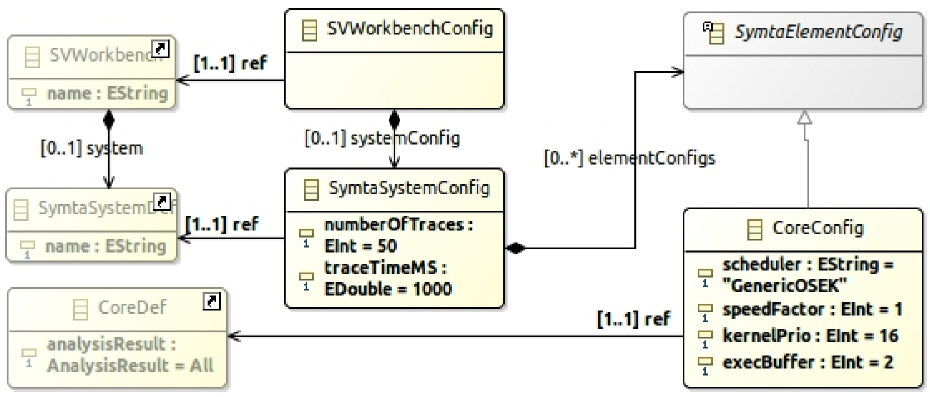
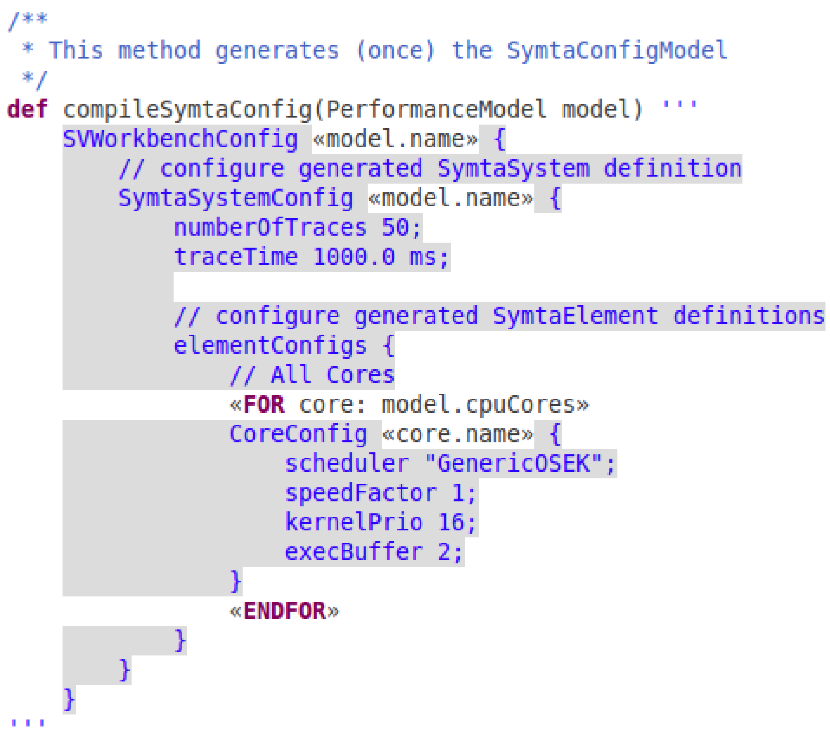


Figure 6.4.: SymTA/S Configuration Ecore meta-model

Using a different *speedFactor* than 1 allows a temporary change in the CPU speed to experiment with different speed options without the need to modify the actual execution times of the relevant tasks. The default value of 1 means that the execution times are taken exactly as modeled. The *execBuffer* limits the number of waiting activations per task. The general policy followed in this dissertation is that the activation of tasks should not be queued, which means that if a task misses an activation, it just waits for the next activation. In practice, however, as an optimization, it makes sense to let the thread pass into its next cycle without waiting in case of a missed intermediate activation. Therefore, the default value of 2 simulates this optimized case. The *kernelPrio* attribute is a mandatory element that needs to be specified for a SymTA/S analysis and allows handling the OS overhead priority levels. The default value of 16 has been empirically determined (however, this value does not have much influence on the overall analysis and might be statically generated by default in future implementations).

Overall, the *SymtaConfig* meta-model allows the definition of SymTA/S-specific analysis options that are not directly related to the actual system but are required to execute the analysis. Deriving from the *SymtaBase* has two advantages. First, because the *SymtaBase* is automatically derived from the *performance* view, all original model changes are directly reflected in the *SymtaBase* model, thus providing a synchronized base for the derived *SymtaConfig* model. This easily allows finding inconsistencies in the *SymtaConfig* model (in cases where base elements are deleted or new base elements are added). The second advantage is that a code generator can be implemented based on the *SymtaConfig* meta-model that also considers the elements from the *SymtaBase* meta-model. In fact, that is exactly the approach to generate the Python script out of the *SymtaConfig* model (see next section). The generated Python script uses the SymTA/S scripting API to initialize SymTA/S projects with configured elements according to the original *SymtaConfig* model.



```
/**
 * This method generates (once) the SymtaConfigModel
 */
def compileSymtaConfig(PerformanceModel model) '''
    SVWorkbenchConfig «model.name» {
        // configure generated SymtaSystem definition
        SymtaSystemConfig «model.name» {
            numberOfTraces 50;
            traceTime 1000.0 ms;

            // configure generated SymtaElement definitions
            elementConfigs {
                // All Cores
                «FOR core: model.cpuCores»
                CoreConfig «core.name» {
                    scheduler "GenericOSEK";
                    speedFactor 1;
                    kernelPrio 16;
                    execBuffer 2;
                }
                «ENDFOR»
            }
        }
    }
'''
```

Figure 6.5.: Performance => SymtaConfig model-to-text transformation

To reduce the effort of manually creating *SymtaConfig* models with standard values, an initial version of a *SymtaConfig* model is automatically generated by default. Figure 6.5 presents the *Performance* => *SymtaConfig* model-to-text transformation (using Xtend's template mechanism). The generated textual model conforms to the Xtext grammar provided in Appendix A.7. Thus, the average developer is not immediately confronted with the detailed SymTA/S configurations and can rely on the default values, while at the same time, system-specific adjustments can be specified (if needed). This model is generated only once at the beginning, and not overwritten anymore, to keep developer's potential changes.

6.1.3. SymTA/S Python script generation and SymTA/S project configuration

Figure 6.6 illustrates the last step in the SymTA/S mapping, namely the model-to-text transformation from a *SymtaConfig* model (see preceding Section 6.1.2) toward a Python script that uses the scripting API of the SymTA/S & Trace Analyser tool.

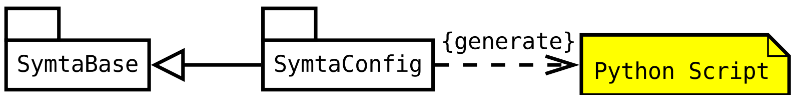


Figure 6.6.: PythonScript generation step

The Python API from the SymTA/S & Trace Analyser allows initializing a new project as well as providing all the project details required to analyze the current system. Therefore, the SymTA/S & Trace Analyser tool is started from a terminal and the generated Python script is passed to the tool as a console parameter. The following Listing 6.7 shows the relevant command line call for a Linux OS.

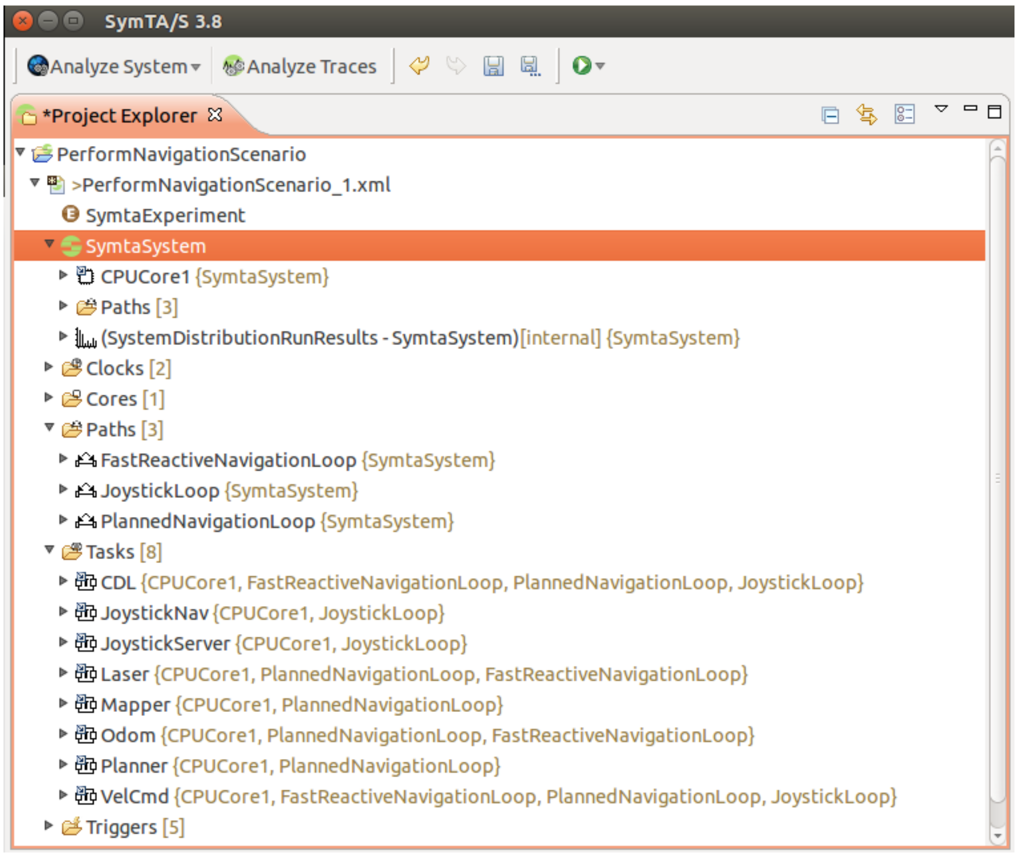
```
./SymTA-S -noSplash -data <output-folder> -script <generated-python-script>
```

Listing 6.7: SymTA/S command line

The SymTA/S & Trace Analyser tool starts, reads-in the Python script, creates the project structure and executes the analysis, thereby simulating the runtime behavior for the given system. Figure 6.7 shows the resulting SymTA/S Project Explorer view of an example system. The project structure and the generated simulation and analysis results are stored within the generated Extensible Markup Language (XML) file (such as the *PerformNavigationScenario_1.xml* file in the example in Figure 6.7).

Switching into the *Results* perspective within SymTA/S & Trace Analyser allows closely investigating all the determined results. Chapter 7 provides a detailed example including a summary of the relevant SymTA/S analysis results.

Overall, the two transformation steps—(i) M2M from *performance* toward *SymtaBase* and *SymtaConfig* and (ii) M2T from *SymtaConfig* toward the Python script—lead to the following



```
PerformNavigationScenario
  >PerformNavigationScenario_1.xml
    SymtaExperiment
    SymtaSystem
      CPUCore1 {SymtaSystem}
      Paths [3]
      (SystemDistributionRunResults - SymtaSystem)[internal] {SymtaSystem}
    Clocks [2]
    Cores [1]
    Paths [3]
      FastReactiveNavigationLoop {SymtaSystem}
      JoystickLoop {SymtaSystem}
      PlannedNavigationLoop {SymtaSystem}
    Tasks [8]
      CDL {CPUCore1, FastReactiveNavigationLoop, PlannedNavigationLoop, JoystickLoop}
      JoystickNav {CPUCore1, JoystickLoop}
      JoystickServer {CPUCore1, JoystickLoop}
      Laser {CPUCore1, PlannedNavigationLoop, FastReactiveNavigationLoop}
      Mapper {CPUCore1, PlannedNavigationLoop}
      Odom {CPUCore1, PlannedNavigationLoop, FastReactiveNavigationLoop}
      Planner {CPUCore1, PlannedNavigationLoop}
      VelCmd {CPUCore1, FastReactiveNavigationLoop, PlannedNavigationLoop, JoystickLoop}
    Triggers [5]
```

Figure 6.7.: SymTA/S Project Explorer

advantages. First, both transformations can be automatically executed in the background without the toolchain user (i.e., system integrator) even taking notice of their existence. Nevertheless, the toolchain developer can easily maintain the meta-models related to the SymTA/S Python API independent of the core robotic meta-models. Inconsistencies between these meta-models are automatically detected as compiler errors in the model-to-model transformation. Second, a SymTA/S performance analysis—which is based on a current *performance* model—can be directly triggered by pushing a button. Thus, the initial learning curve for using the SymTA/S & Trace Analyser is considerably reduced. In the same way, other analysis tools, such as e.g. MAST [MAS] or OSATE 2 [OSA] can be integrated and used alternatively or even in parallel to the SymTA/S & Trace Analyser.

6.2. Logging End-to-End Delays in a Real System

Logging generally refers to capturing data from an executed system. The data is thus stored in log-files for a later offline analysis. Which data exactly is captured highly depends on the desired information to be derived in the offline analysis. The primary goal of this dissertation is to integrate performance-related aspects, such as end-to-end delays, into an overall robotic development process. Therefore, a performance analysis based on SymTA/S is integrated and interlinked with the robotic models. The next Chapter 7 evaluates whether the analysis yields meaningful results for a real robot. To be able to asses this, first, some ground truth values need to be captured from the system executed in a realistic setting. Therefore, the following three types of information are of particular interest in this dissertation:

1. The activation frequency (periodic/sporadic) of individual tasks in a system
2. The real execution times, i.e., cycle times (min/max, average/mean, distribution, etc.) of individual tasks in a system
3. Overall end-to-end delay times (with jitter) of signals traversing several interlinked tasks (of a task-chain)

6.2.1. Designing the logging infrastructure

Before the logging functionality can be designed, some initial assumptions need to be made. To begin with, robotic systems consist of several distributed components, each potentially consisting of several tasks. Therefore, one could implement a central Logging Server or even to use the System Logger such as e.g. described in [HJS03b] that interacts with the components over a generic and networked logging interface. While this is a flexible and powerful solution, it imposes a severe modification and a heavyweight extension of the underlying framework infrastructure, which might become a reasonable course of action in future implementations. However, because this dissertation only requires the logging of rather few local data values, a much simpler and lightweight solution is more appropriate. The approach is basically to stream all required log entries into a separate log-file for each component. This can be easily implemented using standard file-streams.

To derive the three types of information mentioned above, it is necessary to identify the exact types of data values to be logged and the exact places in the code where relevant events to be captured occur. Since we are always interested in timings in all three information types (above), the main value to be logged is the current time-stamp of any relevant event. The exact places in code depend on the type of information to be derived. First, for deriving the activation frequencies of tasks, a log entry needs to be generated each time a task starts its current cycle. In a later offline analysis, the cycle time can be determined by calculating the time differences between successive task activations.

Second, for deriving the execution times of tasks, an additional log entry needs to be generated each time the task finishes its current cycle. The execution times can be calculated by simply subtracting the start times from the end times of each task cycle. Thus, two additional aspects need to be considered. First, the system scheduler might preempt a task during its current task cycle by another task with a higher priority. As argued in Chapter 7, this problem can be mitigated by using a FIFO scheduler and assigning the same priority to all tasks. Hence, a task always completes its current cycle without being interrupted by other tasks, thus leading to realistic execution time measures. Second, it turns out that the end of a task cycle is not so clearly defined as it might appear at first. For instance, because the general function of any task is to produce some kind of data, the end of a task cycle could be defined as the time just before the task's calculated results are published. Alternatively, the end of a task cycle might also include the publication (i.e., transmission time) of the calculated results to the next component(s). In the current implementation of the *PushServer* in ACE/SmartSoft, the actual transmission of published results is effectively performed by the calling task, which means that a task cycle should include the transmission time in order to determine a realistic worst-case scheduling analysis (as part of the SymTA/S performance analysis).

The third type of information—related to end-to-end delay measuring—requires additional efforts with respect to code instrumentation. First, tasks in a system are individual entities and can be executed either synchronously (i.e., blocking until results from preceding tasks become available), or asynchronously (i.e., it is not defined which exact result-message from the preceding task will be used in the current task's cycle). In both cases, it is necessary to exactly know which produced message in one task is used in the next cycle of the follower task, which in turn produces its own message and propagates it to yet another follower task in the chain and so forth. The basic idea to address this problem is based on the following two extensions. First, each task gets its own update-counter, which is incremented for each new task cycle. Second, this update-counter is stored in the produced message that is communicated to the follower component (or respectively a certain task of that component). At this point, a log entry for the start and the end of a task cycle needs to store not only the current time-stamp and the current update-counter value but also the update-counter value of the preceding task in a task-chain. Thus, in a later offline analysis, log entries of individual task cycles can be linked together along the chain of tasks by correlating the relevant update-counter values.

6.2.2. Logging core instrumentation

The instrumentation of code for logging can be separated into two distinct parts. One part is related to a generic logging infrastructure which every component automatically gets by default. This part can be implemented within the underlying robotic framework. The other part is the individual instrumentation of the component's internal business logic. This latter part can be partially automated by code generators using the generation gap pattern. Thus, the component-specific logging infrastructure is generated for each relevant element in accordingly generated base classes, while the actual business logic is implemented in derived classes. The advantage is that the logging infrastructure is automatically included for all relevant elements of a component without bothering the component implementer with irrelevant details.

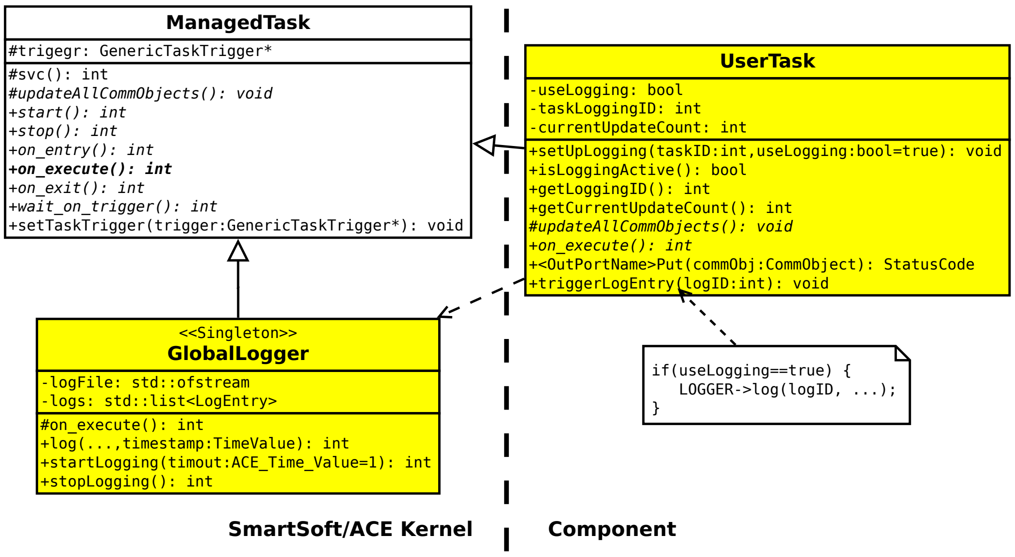


Figure 6.8.: Class-diagram for the two generic logging classes and a UserTask class

Figure 6.8 shows a class diagram with the main logger class `GlobalLogger` as part of the ACE/SMARTSOFT framework implementation and an exemplary representation of a custom `UserTask` implementation as part of a hypothetical component (see classes with yellow background color in Figure 6.8). The `GlobalLogger` implements the singleton design pattern and thus can be used from everywhere within a component (using the macro `LOGGER` as shown in the comment in Figure 6.8). Each call of the `log(...)` method creates a new log entry, including the passed arguments and a time-stamp from the system clock. The log entry is then pushed back onto an internal FIFO queue. A thread drains this queue and writes the entries into a file-stream.

```
Sec|Usec|ID|CurrCO|PrevCO
1471509610|446090|0|0|146
1471509610|448075|1|0|146
1471509610|448082|2|0|146
1471509610|685323|10|0|0
1471509610|712814|0|1|156
1471509610|716006|1|1|156
1471509610|716018|2|1|156
...
```

Listing 6.8: Log-file content example (the '|' separates the columns)

Listing 6.8 shows an example of a typical log-file content. The header line defines a name for each column. In this example, the first two columns provide the time-stamp (in seconds since 1970 and microseconds of the current second). The third line is an ID that defines the source that triggered that log entry. This source ID is calculated by combining the `taskLoggingID` (for each new task in a component the ID is incremented by 10) and a numerical offset, identifying the actual place within a task where the log entry has been triggered. For example, ID value 0 refers to the start time of a cycle from task with the `taskLoggingID` 0. The ID value of 3 refers to the end time of that same task. The last two columns store the task's `currentUpdateCount` and the update-count from the preceding task that has in turn generated the currently used input message.

Figure 6.9 shows a sequence-diagram that illustrates the common procedure to log the above-mentioned start/end times of each task cycle. The main procedure of each task is executed by three methods: `on_entry()`, `on_execute()`, and `on_exit()`. The first and the third methods are executed only once during respectively the initialization and destruction of a task. The method `on_execute()` is cyclically executed in an infinite loop. In the first step of this loop, the task awaits a release (i.e., activation) from its own trigger. As described in Section 4.2.3, the trigger can be configured as one of three options: a *PeriodicTimerTrigger*, a *PushClientTrigger*, or a (custom) trigger by overloading the method `wait_on_trigger()` in derived task classes. In any case, the second step in the loop is to update a local copy for all required communication objects that are currently available in the referenced input ports. Hence, the task can use a consistent copy of all required communication objects throughout its entire cycle without the objects being overwritten in between.

The input communication objects store the update-counters from the preceding tasks of a task-chain. Again, this counter is required to correlate the current task cycle with the relevant task cycle of the preceding task. Hence, the next step is to log the current time as a current task cycle's **start time** along with the task's ID, the current (local) update counter value and the update-value from the preceding task. It is worth mentioning that the time to copy the communication object(s) is neglected here. Particularly for big communication objects, this time might not be insignificant. However, in most cases, this time is considerably smaller than the actual execution time of a task and thus can be neglected.

Subsequently, the task executes its business logic. The results from that part are typically either propagated through the relevant output port or sent to an internal actuator driver. In any case, the next step is the time where the current task cycle's **end time** has to be logged (which is done the same way as the start time). The last step is to increment the task's own update-counter and to repeat the whole loop.

In summary, this subsection presented a logging mechanism to store the start- and end times of each task cycle for each individual task in a component. The resulting logs allow reproducing the sequences of successive task cycles for tasks along a modeled task-chain. Overall, these log-files are used as input for a manual analysis of the real system's execution times and end-to-end times, which are used in the next chapter as ground truth values to evaluate and asses the suitability of a SymTA/S-based performance analysis for a real system.

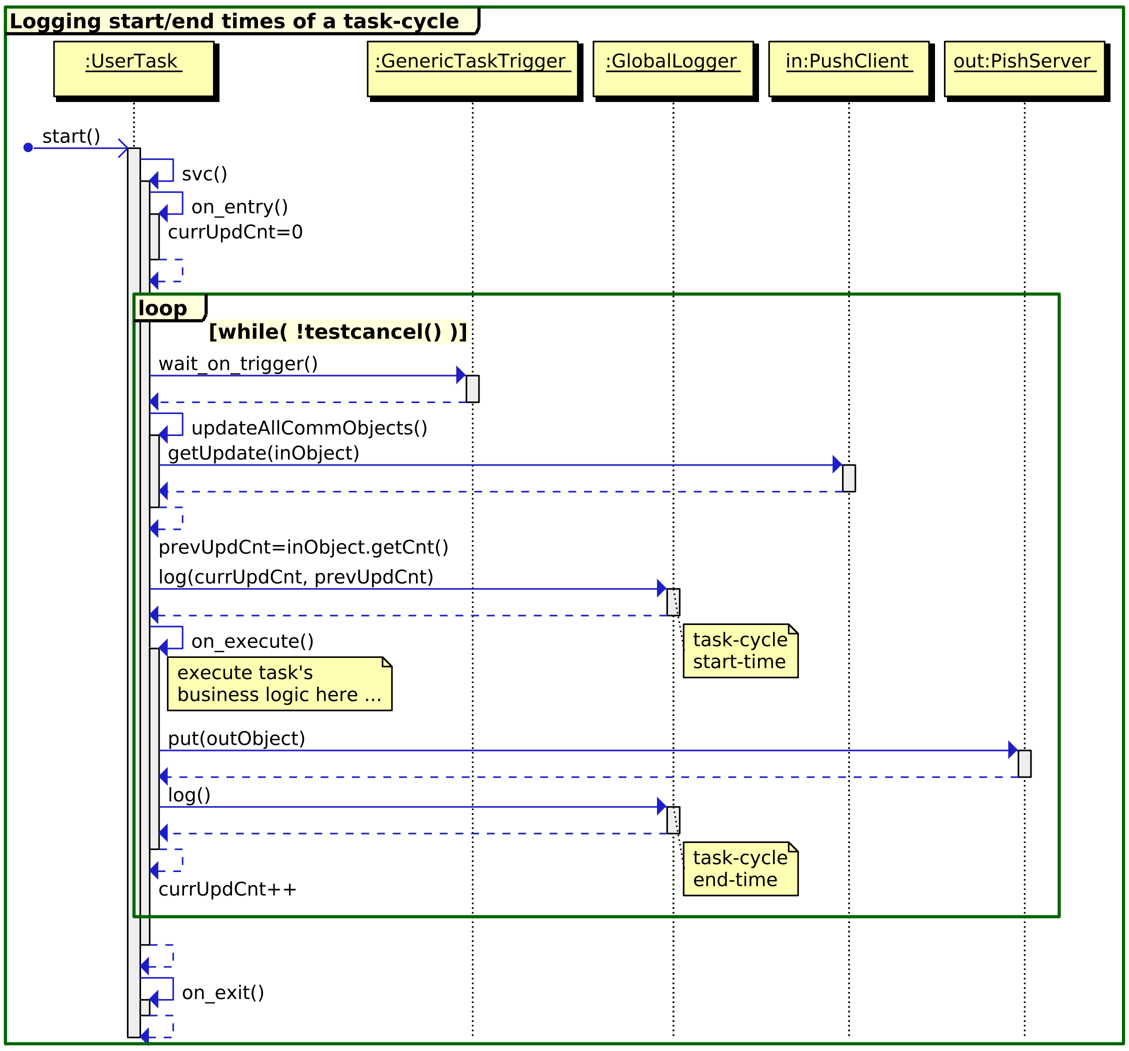


Figure 6.9.: Sequence-diagram showing the logging of task cycle's start/end times

6.3. Monitoring System Properties at Runtime

One particularity of the domain of service robotics is that, after a software system has been deployed to the actual robot platform, the system does not necessarily become static and fixed. Instead, the system needs to flexibly and continuously adapt to changing situations and conditions that the robot will face during execution. This means that the robot will reconfigure its on-board software components according to a predefined set of rules and a currently executed task so as to maintain a certain service quality in execution. To make informed decisions, a robot requires a certain level of information, not only about its surrounding but also about its own internal system state. The extraction of the current system state is often referred to as runtime monitoring.

Runtime monitoring can be considered as an extension to logging (compare Section 6.2). The general difference between logging and monitoring is that logging stores all the collected information in log-files for later offline analysis. Monitoring, on the other hand, additionally analyzes the recorded information on the fly and directly provides an instant feedback from the system, typically visualized in a Graphical User Interface (GUI). Both approaches have pros and cons. For instance, logging typically is more efficient and affects less the runtime behavior of the system but requires offline analysis tools leading to less efficient overall analysis loops. Monitoring allows performing in-system debugging, directly triggering testing events according to the monitored information, as well as feeding the monitored information back into the system as an additional source of information for improving the overall runtime behavior. The drawback of monitoring is its higher resource demands. Analyzing information on the fly inevitably consumes computation resources, thereby affecting the actual runtime behavior.

This section henceforth presents conceptual ideas for designing a monitoring solution and is structured as follows. First, Section 6.3.1 provides further details about service quality aspects from the perspective of this dissertation. Then, Section 6.3.2 discusses common runtime properties that might be interesting for runtime monitoring.

6.3.1. Runtime variability exploitation for maintaining a service quality

Autonomous mobile service robots always need to deal with limited on-board resources, incomplete and faulty information, as well as unpredictable outcomes of the robot's own actions and the generally unpredictable situations of the real world. Therefore, a service robot needs to do a trade-off between maximizing the success probability to accomplish the current goal and minimizing computation efforts. A service robot thus continuously looks for promising sequences of actions to accomplish the current mission at hand. As illustrated in [Lot+14], a robot often finds several alternative sequences with comparable success expectations. In order to choose one alternative over another, a robot further needs to take additional *non-functional aspects* [Lot+14] into consideration such as e.g. maximizing overall execution performance, reducing energy consumption, maximizing safety, etc.

In this context, the overall responsiveness of the robotic system at runtime can be considered as one of such *non-functional system properties* that has a direct influence onto the robot's performance and the perceived execution quality while interacting with humans. As mentioned at the start of this dissertation, such properties need to be managed as early as possible in the overall development process. An appropriate solution based on the SymTA/S & Trace Analyser is presented in Section 6.1. However, an analysis is always just an approximation of the real runtime behavior. While the analysis already is highly beneficial during the system design, it must **not** stop there. Instead, the resulting system can be additionally instrumented in such a way that it can monitor its own *non-functional system properties* as an additional source of information to adjust and adapt its own execution.

For example, a robot that performs a typical pick-and-place task might determine that the responsiveness of its own obstacle-avoidance control loop is better than originally presumed during the system design phase, thus potentially allowing higher velocities without violating any safety constraints. To increase the overall performance, a robot thus might try to gradually increase its velocity (bounded by a maximum value) while monitoring the responsiveness. Overall, using a direct system feedback at runtime for improving the overall execution quality seems to be the next logical step; however, it is not yet fully explored within this dissertation and can be investigated in a future work.

6.3.2. Best-effort and hard real-time execution

Another interesting runtime aspect is the required accuracy of the expected and presumed runtime behavior. Analysis approaches such as e.g. SymTA/S allow predicting the runtime behavior of a system with a certain accuracy. As argued in Section 3.2.5, for many basic functions of a robot, best-effort behavior is entirely sufficient. For these best-effort cases, approaches such as SymTA/S provide an approximation only (i.e., average case values) of the overall timings. It is important to notice that these average values alone already are a huge support in the overall system design. However, it might still be of additional interest to know (at least in retrospect) how the system has behaved in a certain environment with all the specific cases. Therefore, a dedicated monitoring infrastructure can be injected into each component using structural information from the *performance* models. This infrastructure allows detection of all the cases that deviate from the simulated limits of the analysis tool. Moreover, as argued above, if a system behaves in a "more efficient" fashion in a certain environment than what has been predicted in a performance analysis, then a monitoring solution can be used at runtime by the system to adapt its own behavior so that it can reduce resource consumption or increase performance without violating the designed timing constraints. In this sense, a monitoring solution adds flexibility to the overall system design and increases development efficiency because the system designer can reduce the efforts of designing an overall system in the traditional (i.e., hard real-time) manner by using regular hardware components and regular operating systems while still being able to get in-system feedback for further refinements if needed.

There are different levels at which runtime execution parameters with respect to responsiveness can be monitored and managed at runtime. It starts within individual components and tasks that can react to timeouts while receiving incoming data. Moreover, individual time-outs can be sometimes cured in a whole chain of involved components or accumulate to time violations of the overall chain that need to be handled on a higher level. This higher level is the task coordination layer called the *sequencing layer* [Lut+14]. How time violations are handled on the *sequencing layer* is beyond the scope of this dissertation. However, a prerequisite is again the ability to detect the time violations in the first place.

This leads to the conclusion that a mechanism is needed that can detect timeouts locally within a component, allowing implementing local timeout handling strategies. Moreover, system-level timeouts are of particular interest to be detected by monitoring several chained components in combination. While the conceptual foundations for this part are clear and a previous work in [LSS11] introduced a generic monitoring solution, the implementation of monitoring as described above has not yet been fully realized and thus remains an interesting case for future works. However, it is important to notice that a monitoring solution is costlier to develop than a comparable logging solution (regardless of its later usefulness). Therefore, it often makes sense to start with a simple logging approach (such as e.g. presented in Section 6.2) and then to extend and refine it toward a fully-fledged monitoring solution (which is often conceptually straightforward).

6.4. Related Works and Conclusion

This section lists some selected approaches related to simulating and analysing the overall execution performance of a system and an approach related to runtime monitoring.

6.4.1. Performance analysis approaches

The analysis methods for the overall execution performance of a system can be separated into two main groups. The first group is related to determining the Worst-Case Execution Time (WCET) and the second group is related to analysing scheduling and network arbitration effects that additionally consider possible inter-task-dependencies.

There are several approaches for determining a WCET. An overview is provided in [Wil+08]. However, as argued in [Lot+16] such approaches are not well suited for robotic cases due to the typically rather unpredictable runtime behavior of common robotic algorithms. Instead, it is more appropriate to use profiling, i.e., to instrument the source code in such a way that the min/max execution times can be directly measured and logged in the target platform. Therefore, it is often sufficient to execute isolated parts of a system (which is typically done for testing purposes anyway).

Having the execution times the next problem to solve is the scheduling analysis of an entire system. Therefore, there are several approaches such as the work by Liu and Layland [LL73], which uses *Rate-Monotonic Scheduling (RMS)*, and the work by Dertouzos [Der74], which uses *Earliest Deadline First (EDF)* scheduling. While these approaches are precise and useful in small systems with a global view on the entire system at the lowest possible level of abstraction (i.e., all threads with priorities are known), these approaches scale badly for more complex systems with highly distributed parts and varying guarantee demands, which is very common in robotics as is also argued in [BS05]. An approach that performs better under such conditions is found in the work of Lehoczky [Leh90], which uses the *busy window approach* and other approaches based on Reservation-Based Scheduling (RBS) such as in [AB01].

As a result, a scheduling analysis alone is not enough to fully analyze specific runtime performance aspects such as end-to-end delays in a system. Instead, additional approaches are required such as the Pletzer's *Timing Definition Language (TDL)* [Ple12], the *Flow-Latency Analysis* [Han07] of AADL, and the SymTA/S [Hen+05] approach. As *TDL* is generally based on the *Logical Execution Time (LET)* [Gho+04], it is generally too inflexible for robotic cases as it requires pressing the execution cycles of all threads into globally synchronized time frames, which scales badly for more complex robotic systems of realistic size and complexity. A very appealing approach is the AADL's *Flow-Latency Analysis* [Han07], which is implemented as a plugin within the OSATE2 tool. Unfortunately, at the time of writing, the *Flow-Latency Analysis* within the OSATE2 tool had some instability issues. That called for other, more matured tools. Alternatives include the MAST tool [MAS] and the Symtavision SymTA/S & Trace Analyser tool. The latter tool has been selected due to its good accessibility and high maturity. As will be shown in the next chapter, SymTA/S & Trace Analyser tool performs reasonably well with appropriate precision also for robotic systems.

6.4.2. Monitoring and runtime adaptation

In a previous work published in [LSS11], a monitoring approach has been developed that allows monitoring different aspects of robotic software components without breaking the components' encapsulation while minimizing the resource overhead of monitoring. The basic idea is to divide the monitoring solution into two parts: one efficient part embedded into the infrastructure of each component, and the other more elaborate part consisting of the analysis logic with a GUI which can be executed on a remote development computer (which is typically less resource constrained). The basic ideas of this approach are discussed in Section 6.3. However, further investigation is needed with respect to the use of monitored information in a system for flexible runtime adaptation.

Beyond the domain of robotics, there is a research community around the topics of *self-adaptive systems* [Wey+13; WIS13; IW15], and *architecture-based self-adaptation* [Ore+99]. The overall idea is that a system is separated into at least two layers, where the lower layer(s) implement the managed system that directly interacts with the environment and a higher layer(s) implementing the managing system that adapts the underlying managed system if needed. The managing system typically comprises the MAPE-K [IBM06; IW15] components, which altogether implement the adaptation control loop for the underlying managed system. In this sense, monitoring as discussed above is the first part of such a MAKE-K control loop. Further details related to runtime adaptation for robotic scenarios can be found in [Lot+14].

6.4.3. Conclusion

This overall chapter has focused on runtime-specific aspects such as the analysis of dynamic runtime conditions using a Compositional Performance Analysis (CPA) based on SymTA/S. Therefore, the SymTA/S & Trace Analyser tool has been interlinked with a performance view (see Section 6.1). Moreover, Section 6.2 presented a flexible logging solution that allows recording in-system timings. This logging mechanism is used in the next chapter for determining end-to-end times, which will be helpful for evaluating whether the proposed abstraction of the *performance view* along with the presented performance analysis is adequate for designing real-world robotic applications. In addition, Section 6.3 provided some initial ideas about coming up with a monitoring solution as a means to dynamically improve the overall runtime execution performance.

Part III.

Results, Future Works and Conclusions

"In theory there is no difference between theory and practice.
In practice there is."

—Yogi Berra

7 Demonstration Examples and Performance Analysis Results

So far, the overall focus in the preceding four method chapters has been on concepts and meta-models with their structures and semantics. By contrast, this chapter henceforth shifts the focus toward a real-world system example that is developed using the novel modeling tools. This allows empirically evaluating the modeling tools. Additionally, the system example is not only designed at the model level but is particularly grounded in real component implementations that are executed on a Pioneer P3DX robot in a realistic setting. Consequently, the actual end-to-end timings are logged on the robot and are used to evaluate the precision of the SymTA/S-based Compositional Performance Analysis (CPA) that is directly initiated from the example system models. The idea is to evaluate the hypothesis that a CPA allows estimating and designing vital performance-related system aspects early in a robotic development process even before the actual hardware comes to exist.

This chapter is structured as follows. The next Section 7.1 presents and discusses the navigation scenario, which has been chosen to represent common robotic use-cases. Therefore, the graphical model notation is presented and the involved architectural design choices for individual component and system models are discussed. Thereafter, Section 7.2 conducts a SymTA/S-based performance analysis and compares it with ground truth measures from a real robot executed in a realistic environment. The goal is to assess whether SymTA/S yields realistic results for a real robot. Section 7.3 concludes this chapter with a discussion of the presented results and reflects them within the overall scope of this dissertation.

7.1. Navigation Component- and System-Models

In order to evaluate the modeling capabilities of the novel tools from this dissertation, the navigation scenario (shortly introduced in Section 3.2.1) has been selected as one of the recurring scenarios in the domain of service robotics. The navigation scenario has been remodeled with the new modeling tools whose implementation is publicly available as open-source on source-forge: https://sourceforge.net/p/smart-robotics/smartmdsd-v3

The navigation scenario consist of several components (see Figure 3.5 in Section 3.2.1) whose internal implementations have been ported to support the flexible task configuration as described in Section 4.2.3. The modified component implementations together with the accordingly extended ACE/SMARTSOFT framework are all available as open-source on source-forge (same URL as above).

This section presents and discusses the models of the involved navigation components and the designed navigation system. The model-diagrams have been created with a graphical model editor based on the *Eclipse Sirius*[1] plugin. The model editor provides a graphical representation for all the meta-model elements and most of their attributes. In some cases, where it is cumbersome to model element-attributes graphically, another representation has been used. For instance, the attributes from *ActivationConstraints* of a component are modeled in tabular form. It is worth mentioning that any model representation, be it graphical, tabular, or even textual, just is a representation of the same in-memory Abstract Syntax Tree (AST). Thus, regardless of which model editor has been used in the end, Eclipse always keeps the different representations synchronized and consistent, thus allowing to use personally preferred representations and editors. Again, the implementations of the modeling tools are publicly available as open-source on source-forge under: `https://sourceforge.net/p/smart-robotics/smartmdsd-v3`

Meanwhile, a new screencast[2] on YouTube demonstrates the usage of the novel modeling tools. The following two sections present and discuss individual component-models and their instantiation in system models.

7.1.1. Navigation component-model-diagrams

This section presents the navigation component models using a graphical notation whose element mappings with relevant meta-model elements are summarized in Table 7.1. Additionally, Table 7.2 lists the *ActivationConstraints* from all presented tasks in relevant components and Table 7.3 lists the over-/under-sampling configurations of related *InputLinkExtensions*.

Figure 7.1 (on the left) shows the *SmartPioneerBaseServer* component. *SmartPioneerBaseServer* wraps up the functionality of the Pioneer P3DX driver and provides generic, driver-independent services that can be used by other components in the system. Thus, *SmartPioneerBaseServer* acts as both a sensor in the sense that it constantly provides odometry updates (through the *BasePositionServer* output port) and as an actuator in the sense that it receives navigation commands (through the *VelocityClient* input port) that are internally propagated to the actual velocity controller. The core functionality of the component (i.e., the interaction with the actual base driver) is implemented within the *RobotTask* class. As shown in Table 7.2, *RobotTask* is not configurable, which means that it is triggered by an own internal trigger (in this case the communication interface of the internally used device driver). That is, the *RobotTask* must not be interrupted (e.g. due to waiting on incoming messages of a component), because this might lead to hardware communication errors due to violated timings. This is a common case for hardware

[1]Eclipse Sirius: `https://eclipse.org/sirius/`

[2]Toolchain screencast: `https://youtu.be/JIYPJXmop3U`

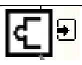
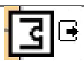
Graphical Element	Meta-Model Element
SmartPioneerBaseServer	*Component*
	*InPort*
	*OutPort*
RobotTask	*PreemptiveTask*

Table 7.1.: Graphical notation of component's meta-model elements

components where tasks directly interact with hardware devices. Therefore, it makes sense to separate the interaction with the hardware (implemented in the *RobotTask*) from the interaction with the component's input-port *VelocityClient* (implemented in the *VelocityCommandTask*) and output-ports (implemented in the *PoseUpdateTask*).

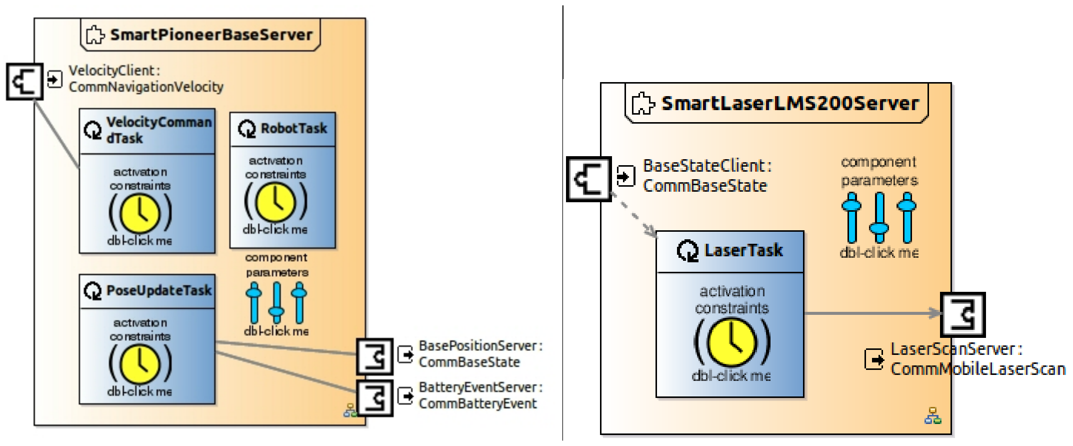


Figure 7.1.: Component diagrams for the SmartPioneerBaseServer (on the left) and the Smart-LaserLMS200Server (on the right)

At this point, it is worth mentioning that in the initial version of the meta-models (as introduced in the preceding chapters) potential dependencies between tasks within a component (such as e.g. between the *RobotTask* and the *PoseUpdateTask*) have not (yet) been made explicit at the model level. The reason is related to the focus in this dissertation, namely to introduce the management of non-functional aspects at the model level, as well as to keep the meta-models simple and generic. However, in a later implementation (shortly discussed in Chapter 8), the meta-models have been extended to allow specification of inter-task dependencies, which improves the readability of the models but can be considered only ornamental for this dissertation. For the discussion hereinafter, it is sufficient to know that the dependent tasks within a component interact with each other at the code level using regular mutual exclusion and synchronization mechanisms (such as mutexes, semaphores, and guards).

Component-name	**Task-name**	**is-configurable**	**MIN** Act. Freq.	**MAX** Act. Freq.
SmartPioneerBase-Server	PoseUpdateTask	true	10.0 Hz	40.0 Hz
SmartPioneerBase-Server	RobotTask	false	10.0 Hz	40.0 Hz
SmartPioneerBase-Server	Velocity-CommandTask	true	10.0 Hz	30.0 Hz
SmartLaserLMS200 Server	LaserTask	false	33.0 Hz	40.0 Hz
SmartJoystickServer	JoystickTask	false	1.0 Hz	50.0 Hz
SmartMapperGrid-Map	CurrMapTask	true	10.0 Hz	20.0 Hz
SmartMapperGrid-Map	LtmMapTask	true	2.0 Hz	10.0 Hz
SmartPlanner-BreadthFirstSearch	PlannerTask	true	4.0 Hz	10.0 Hz
SmartCdlServer	CdlTask	true	5.0 Hz	40.0 Hz

Table 7.2.: Overview of Activation-Constraints for all Tasks of the presented components

Furthermore, as shown in Table 7.2, the *PoseUpdateTask* is configurable within the interval from 10 to 40 Hz. Thus, a stable update frequency is guaranteed independent of the actual configuration of the *RobotTask*. This provides an additional variation point for system-integrators for adjusting the component's behavior to the later system design. Of course, undersampling or oversampling can occur depending on the according configurations. An alternative design might be to directly propagate odometry updates from the *RobotTask* to the *BasePositionServer*.

The *SmartLaserLMS200Server* (on the right in Figure 7.1) demonstrates another component with hardware-related functionality. That is, the *LaserTask* directly interacts with the locally used device driver of the respective laser range-finder sensor. Again, as shown in Table 7.2 the *LaserTask* is not configurable because it is triggered by the internal device driver. One particularity of the *SmartLaserLMS200Server* component is that it optionally uses odometry updates from the *BaseStateClient* input port. The reason is that a laser range-finder can be either mounted on a mobile (i.e., moving) platform or rigidly fixed on a stationary platform. In the former case, each new laser scan is stamped with a current odometry value (i.e., six-dimensional pose) so as to know at which coordinates this scan was recorded. In the latter case, the laser pose is statically configured during component startup and the *BaseStateClient* is not used.

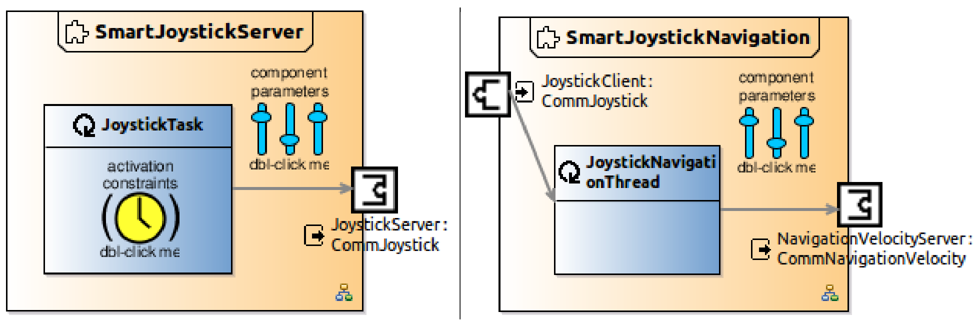


Figure 7.2.: Component diagrams for SmartJoystickServer (left) and SmartJoystickNavigation (right)

The last hardware related component of the presented navigation stack is the *SmartJoystick-Server* (on the left in Figure 7.2). This rather simple component directly interacts with a gamepad device driver to receive relevant commands (i.e., pushed buttons, joystick movements, etc.). Yet again, the *JoystickTask* is non-configurable (see Table 7.2). The interesting functional constraint of this task is that the gamepad driver might not respond for at most one second as long as no buttons are pressed and no joysticks are moved. As soon as any button is pressed or a joystick is moved, the gamepad driver immediately reacts and communicates the changes to the *Joy-stickTask* with a maximum frequency of 50 Hz. Therefore, the effective frequency at runtime dynamically varies between these two boundary values. This behavior is different from the ones in the *LaserTask* or the *RobotTask*, where the update frequency is rather stable during execution of a component but can be configured to a different value depending on the actually used device type (e.g. specified by an ini-file parameter of that component). At present these two cases are not distinguished in the model because in practice it is often not possible to specify the exact update frequency of a device driver in advance (i.e., whether the update frequency will be stable throughout the entire runtime of a component or will it be stable for certain periods of time or can the frequency change at any time?). However, it is mostly possible to define the corner cases, which are later used for the performance analysis. In order to keep the initial meta-models simple, this distinction has not yet been introduced within the component meta-model (but an extension in this direction is straightforward if needed). In consequence, the resulting performance analysis might be over-pessimistic for this part because there the upper bound (i.e., the max activation frequency in Table 7.2) always is considered regardless of whether these values actually will be reached at runtime.

In contrast to the so far presented hardware-related components, *SmartJoystickNavigation* (on the right in Figure 7.2) can be considered as a pure filter component. It receives joystick floatingpoint values (in the range from 0 to 1) through the *JoystickClient* input port, transforms the values into navigation commands (i.e., translation and rotation velocities in m/s) within the *JoystickNavigationThread* and publishes a new velocity message through the *NavigationVelocityServer*. In this case the *JoystickNavigationThread* is pretty lightweight. The implementation does not need any particular activation constraints, which are consequently entirely omitted in this model. The default implicit semantics for this case is that the task can be flexibly configured without any constraints (i.e., the highest possible flexibility). In most systems, the *JoystickNavigationThread* will be configured to synchronously await new incoming joystick messages, yet, this should remain a configurable option for system integrators (because this assumption might not hold true for all future systems).

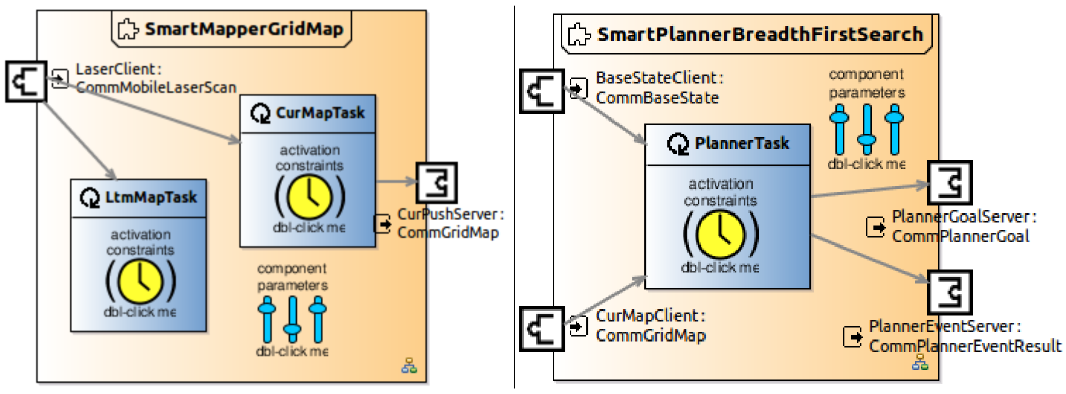

SmartMapperGridMap
LaserClient: CommMobileLaserScan
CurMapTask
activation constraints
dbl-click me
LtmMapTask
activation constraints
dbl-click me
CurPushServer: CommGridMap
component parameters
dbl-click me
SmartPlannerBreadthFirstSearch
BaseStateClient: CommBaseState
component parameters
dbl-click me
PlannerTask
activation constraints
dbl-click me
PlannerGoalServer: CommPlannerGoal
CurMapClient: CommGridMap
PlannerEventServer: CommPlannerEventResult

Figure 7.3.: Component diagrams for SmartMapperGridMap (left) and SmartPlannerBreadthFirstSearch (right)

The next two components, the *SmartMapperGridMap* and *SmartPlannerBreadthFirstSearch* (in Figure 7.3) are related to the grid-map-based path-planning functionality of the navigation stack. The *SmartMapperGridMap* is another example where the definition of more than one task makes sense. Within this component the *CurMapTask* calculates a local occupancy grid-map by aggregating incoming laser-scans and publishes the updated map through the *CurPushServer* output port. The local map is just a section of the overall map (e.g. the map of the current room but not the entire building). This map is replaced by another section as soon as the robot leaves the current section. The *LtmMapTask*, on the other hand, calculates and updates a long-term map of a bigger area such as the entire building. The long-term map typically is communicated to other components using the file system, rather than being communicated over the network (due to the often-extensive map size and comparably rather recent map changes, if any at all). The *CurMapTask* requires considerably less computation resources than the *LtmMapTask*, and thus the current (local) map can be updated more frequently than the long-term map. There are no implementation constraints for the update frequencies of these two tasks and thus the concrete frequencies should remain configurable to be refined in the later system integration. However, due to the rather high resource demands some boundary values have been empirically determined (see Table 7.2) to prevent unrealistic configurations.

Component-name	**InputPort-name**	**Task-name**	**over-sampling**	**under-sampling**
SmartPioneerBase-Server	VelocityClient	Velocity-CommandTask	permissible	permissible
SmartLaserLMS200-Server	BaseState-Client	LaserTask	permissible	permissible
SmartMapperGrid-Map	LaserClient	CurrMapTask	not reasonable	permissible
SmartPlanner-BreadthFirstSearch	BaseState-Client	PlannerTask	not reasonable	permissible
SmartPlanner-BreadthFirstSearch	CurrMap-Client	PlannerTask	permissible	permissible
SmartJoystick-Navigation	Joystick-Client	JoystickNavi-gationThread	not reasonable	permissible
SmartCdl-Server	Laser-Client	CdlTask	not reasonable	permissible
SmartCdl-Server	PlannerGoal-Client	CdlTask	permissible	permissible
SmartCdl-Server	Joystick-Client	CdlTask	permissible	permissible

Table 7.3.: Over-/under-sampling configuration of the different InputLinkExtensions

At this point, it is worth mentioning that the *CurMapTask* has an architectural particularity related to the interaction with the *LaserClient* input port. It does not make much sense to update the current map as long as no laser-scan updates are available because this would result in the very same current map. In other words, oversampling from the *CurMapTask* toward the *LaserClient* would inevitably lead to wasted resources and thus should be avoided. For this purpose, the component model extends the input-link specification by additional over-/undersampling constraints. Table 7.3 provides an overview of all the over-/undersampling specifications for all *InputLinkExtensions*. While in most cases, over- or undersampling are not harmful and can be permitted, in some (rare) cases, it needs to be restricted as e.g. in the case of the *CurMapTask* with the *LaserClient*. This restricts permitted configuration choices for the later system-integration phase.

The main recipient of the updated local grid-maps is the *SmartPlannerBreadthFirstSearch* component (or any other component implementing a map-based path-planning functionality). This component is a nice example where a task receives data from more than one input port. In this case, the *PlannerTask* requires an updated grid-map and the current position of the robot within that map to calculate a path toward the overall goal. At this point, it is worth noting that the meta-models defined in this dissertation omit details related to behavior coordination (such as e.g. shown in [Sta+16]). In particular, behavior coordination requires a generic coordination interface for each component, which at present is generated by default without being shown at the model level. For example, the overall destination to drive to is commanded to the *SmartPlanner-BreadthFirstSearch* component over such an interface from a higher behavior-coordination level (see Section 4.1.3 for further details).

For the two input ports of the *SmartPlannerBreadthFirstSearch* component, the *PlannerTask* requires the following design considerations. First, both input ports need to be non-optional because without a current position and the current grid-map, calculating a new path does not make much sense. Second, although it is likely that the update frequency of the *BaseStateClient* input port is higher than the update frequency of the *CurrMapClient* input port, this cannot be generally assumed for all future systems. In other words, a typical mistake in this use-case is to hard-code a strict binding in such a manner that e.g. the *PlannerTask* is dispatched by newly incoming messages on the *BaseStateClient* input port. This design is effectively prevented by not allowing such a binding already at this step in the overall development process. Still, as shown in Table 7.3, it is possible to express the constraint that it does not make much sense to oversample the *BaseStateClient*, because without knowing the current position, it does not make much sense to calculate a new path. Thus, the decision about which of the two input ports actually will trigger the execution of the *PlannerTask* (or even whether this task will be triggered by a periodic timer) is separated from the implementation-specific requirement of getting new positions and maps. Regardless of which trigger is selected, the resulting overall update frequency should be somewhere in-between the boundaries of 4 to 10 Hz (as shown in Table 7.2).

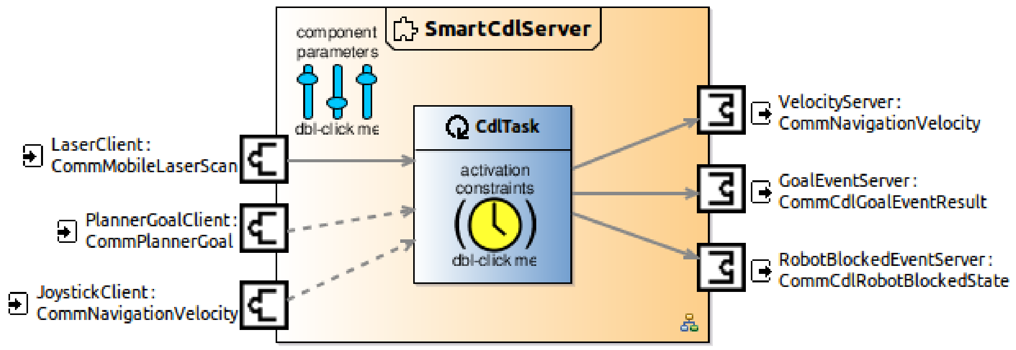

component parameters
dbl-click me
SmartCdlServer
CdlTask
activation constraints
dbl-click me
LaserClient : CommMobileLaserScan
PlannerGoalClient : CommPlannerGoal
JoystickClient : CommNavigationVelocity
VelocityServer : CommNavigationVelocity
GoalEventServer : CommCdlGoalEventResult
RobotBlockedEventServer : CommCdlRobotBlockedState

Figure 7.4.: Component diagram for SmartCdlServer

The last, not yet described, component of the navigation stack is the *SmartCdlServer* (see Figure 7.4). Briefly, *SmartCdlServer* implements a flexible obstacle-avoidance algorithm that uses laser-scans to calculate an adequate collision-free movement depending on different strategies as described below. For this purpose, *SmartCdlServer* has the three input ports: *LaserClient*, *PlannerGoalClient* and *JoystickClient*. The *LaserClient* is non-optional because it provides the minimally needed information-kind for the internal algorithm to function correctly. The other two input ports are both optional, which results in the following possible operational modes. The first operational mode is the purely reactive navigation, where the robot wanders freely in an environment without a certain direction and yet without causing any collisions. This mode is used in exploration phases of the robot where the environment is not yet known and e.g. the map needs to be generated. In this first mode, neither the *PlannerGoalClient* nor the *JoystickClient* are used. The second operational mode is where the *SmartCdlServer* tries to reach the next goal position received through the *PlannerGoalClient* by generating a path that does not cause a collision. This is a regular operational mode that is used to perform goal-oriented navigation. In this case the *JoystickClient* input port is not used. The last operational mode is the joystick-based navigation. In this case the *SmartCdlServer* tries to follow the given orientation and velocity command that is received through the *JoystickClient* input port as long as this command does not lead to a collision. If a joystick command would lead to a collision, then the given velocity is replaced by a slower value (in the worst case by zero), which effectively prevents a collision.

Just like with the *SmartPlannerBreadthFirstSearch* component, the three operational modes of the *SmartCdlServer* are commanded from a higher behavior-coordination level (shortly explained in Section 4.1.3). Yet, the three input ports of *SmartCdlServer* already specify the different implementation-specific constraints, namely the general optionality of the *PlannerGoalClient* and the *JoystickClient* as well as the non-optionality of the *LaserClient*. In addition, as shown in Table 7.3, oversampling the *LaserClient* is not permitted because it does not make much sense to calculate a movement without knowing the newest information (from the newest laser-scan) about possible obstacles in near surrounding of the robot. That way, a system integrator becomes able to flexibly use the *SmartCdlServer* component regardless of which modes will be needed (or not) in his system.

7.1.2. Navigation model-diagrams in the system-integration phase

The previous section (above) presented individual components of the navigation stack. Thus, the focus has been on modeling component-specific implementation constraints without foreseeing too many application-specific requirements. In this section, these individual components are combined (i.e., integrated) into a system. Thus, the remaining configuration options of components are utilized to decide on application-specific concrete configurations without violating any component-internal implementation constraints.

Figure 7.5 shows the system-configuration diagram for the navigation scenario and Figure 7.6 presents the related deployment diagram. The main concern in the system-configuration diagram is to select the right components to be used and to specify the initial wiring between them. It is worth noting that the wiring between components must not remain static at runtime but can be dynamically changed using the *wiring pattern* (see [SSL12b] for more details). Moreover, each instantiated component is given a unique name, which can be different to the component's

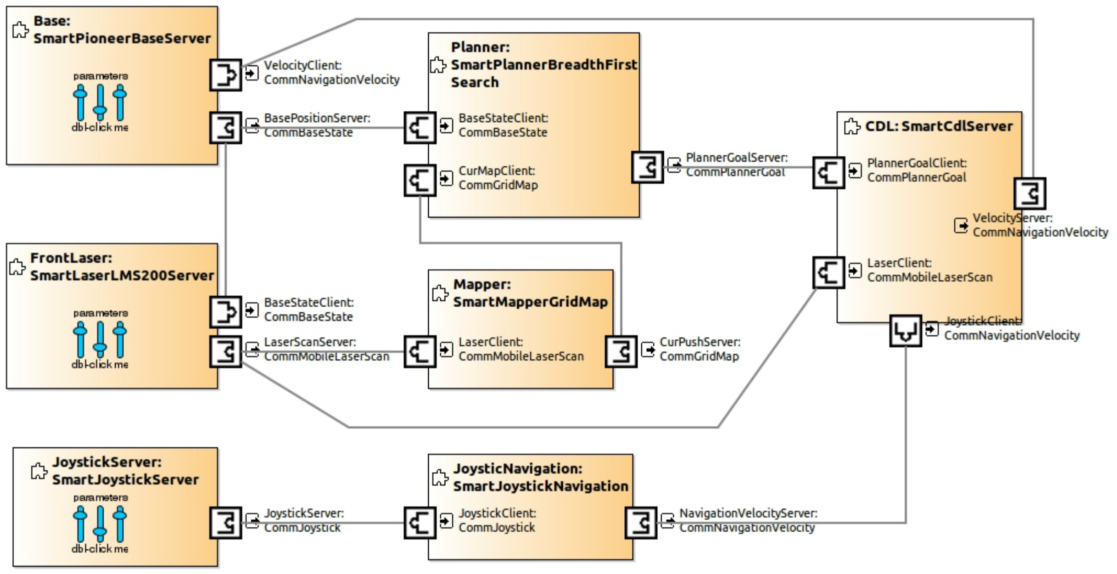

Base: SmartPioneerBaseServer
parameters
dbl-click me
VelocityClient: CommNavigationVelocity
BasePositionServer: CommBaseState
Planner: SmartPlannerBreadthFirstSearch
BaseStateClient: CommBaseState
CurMapClient: CommGridMap
PlannerGoalServer: CommPlannerGoal
CDL: SmartCdlServer
PlannerGoalClient: CommPlannerGoal
VelocityServer: CommNavigationVelocity
LaserClient: CommMobileLaserScan
JoystickClient: CommNavigationVelocity
FrontLaser: SmartLaserLMS200Server
parameters
dbl-click me
BaseStateClient: CommBaseState
LaserScanServer: CommMobileLaserScan
Mapper: SmartMapperGridMap
LaserClient: CommMobileLaserScan
CurPushServer: CommGridMap
JoystickServer: SmartJoystickServer
parameters
dbl-click me
JoystickServer: CommJoystick
JoysticNavigation: SmartJoystickNavigation
JoystickClient: CommJoystick
NavigationVelocityServer: CommNavigationVelocity

Figure 7.5.: System-Configuration Diagram

original name. Thus, the same component can be instantiated multiple times, which makes sense for components such as e.g. the *SmartLaserLMS200Server*, which can be instantiated for a laser mounted at the front of the robot and another laser component of the same type for a laser mounted at the back (or somewhere else) on the robot. However, as the presented navigation example is comparably simple, this feature was not (yet) needed.

Another concern of the system-configuration diagram is to specify the initial parameters of individual component instances. This can be done using another Xtext-based grammar not displayed in this dissertation (see [Sta+16] for more details).

Figure 7.6 shows the deployment diagram that is mainly responsible to define deployment-specific system attributes such as the mapping of individual component artifacts to relevant PC platforms on the robot. In this example, all the components are deployed to the same *PC1* using the folder location */tmp* and the login account *Guest* on that PC. A single CPU named *MainCPU* is specified and a naming-service with the port number 20002 is instantiated.

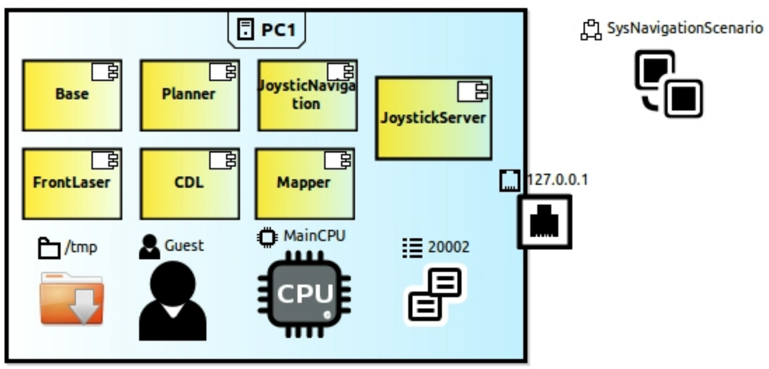


Figure 7.6.: System-Deployment Diagram

Graphical Element	**Meta-Model Element**	**Graphical Element**	**Meta-Model Element**
JoystickServer	*TaskNode*	✉	*DataTriggered*
Trg	*TriggerInputNode*	periodic-timer 10.0 Hz	*PeriodicTimer*
Reg	*RegisterInputNode*	Sporadic	*Sporadic*
94 us - 343 us	*ExecutionTime*	FIFO	*Scheduler*
JoystickLoop	*TaskChain*	300 ms	*MaxAge*
E2ESpecs	*End2EndSpecs*	500 ms	*Reaction*

Table 7.4.: Graphical notation of the performance meta-model elements

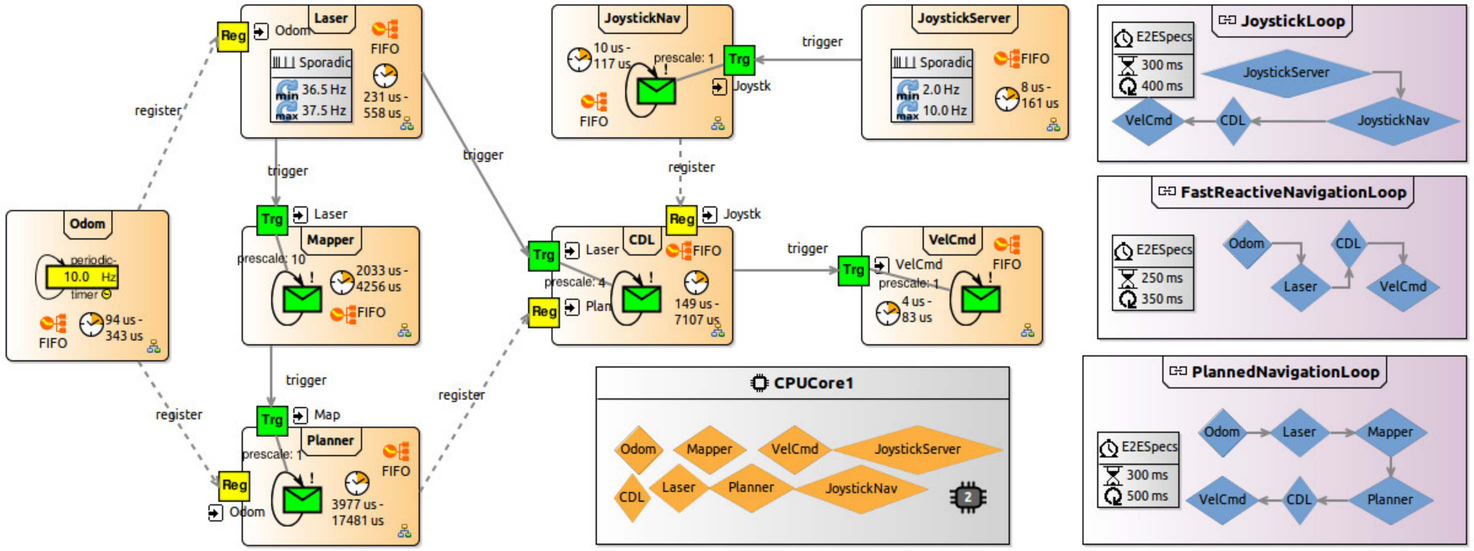


Figure 7.7.: System-Performance Diagram (see also [Lot+16])

Overall, the two modeling views in figures 7.5 and 7.6 are inspired from the previous SmartMDSD Toolchain Version 2 (mentioned in Chapter 2). However, they are required as a foundation for the novel *performance* view, which will be described by example next. Figures 7.7 and 7.8 show the two main diagram-layers of the *performance* view for the navigation scenario. Table 7.4 additionally provides an overview of the graphical notation mappings toward the relevant meta-model elements. The *performance* view shifts the level of granularity from the component level toward the task level. The main purpose of this view is to configure individual tasks within their configuration boundaries in such a manner that altogether they form *cause–effect chains* that adhere to overall end-to-end timing specifications. In other words, the previously specified tasks within components are now considered in combination, at the system level, independent of where the individual tasks are actually realized (i.e., regardless of whether some of the tasks are within the same component or in different components). This abstraction beyond component boundaries is important because in the end whether individual tasks need to synchronously follow the update rates of the preceding tasks or whether some of the tasks require an own timed trigger is absolutely independent of where the tasks are actually realized. Moreover, hiding component boundaries effectively removes some of the avoidable model complexity in this performance view as discussed next.

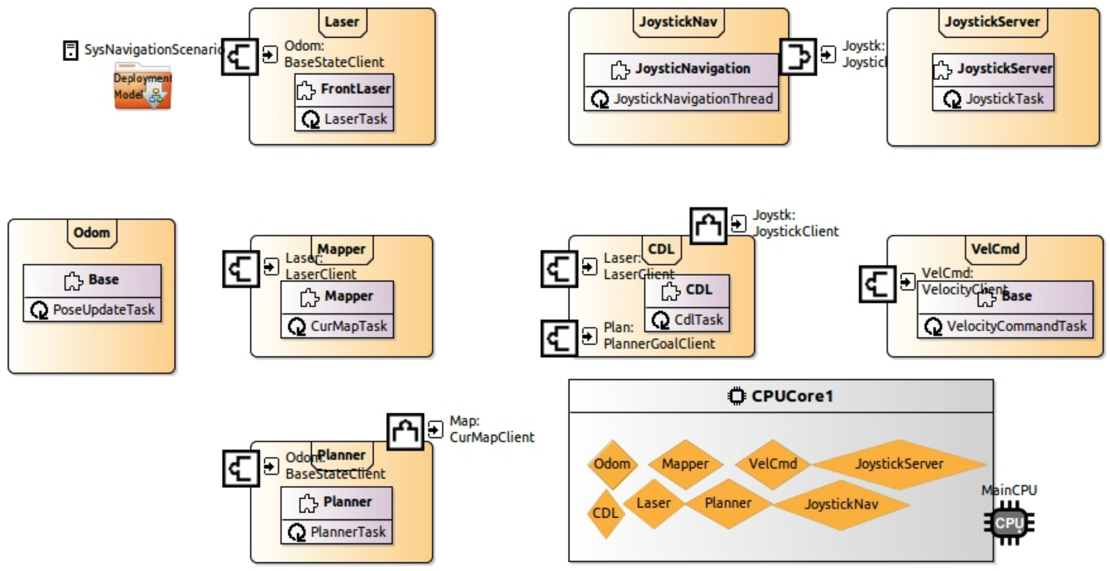


Figure 7.8.: System-Performance Diagram (Deployment-Reference view)

One of the aspects that deserves further attention is that the tasks are modeled “again” in the *performance* view, which might seem redundant at first glance. One might be asking, why to specify the tasks once more if they are already defined in the component models already?

Ultimately, it should not be possible to specify tasks, which are not implemented somewhere. Moreover, tasks are not instantiated independent of their surrounding component (i.e., component instantiation implicitly also instantiates the tasks). Nevertheless, the definition of *TaskNodes* (and *TaskChains*) still makes sense due to the following two different modeling use-cases. The first use-case is that it might make sense to start designing the overall performance-specification upfront consisting of hypothetical *TaskNodes*, which may or may not be later bound to actual *TaskRealizations*. Thus, one might already start with an initial coarse-grained performance analysis even before all of the components have been specified, selected, and instantiated. As soon as additional details about the system become known, the performance analysis can be refined until all the components and thus tasks are known and mapped. The mappings between hypothetical *TaskNodes* and the actual tasks are modeled in a separate diagram layer shown in Figure 7.8. It is important to note that the overall system first becomes deployable, and thus executable, when all the specified *TaskNodes* are also mapped to tasks of relevant components. Prior to that, the system is analyzable but not executable. The second use-case is that the *TaskChains* are specified after the system-configuration has been completed. Even in this case, it is just a matter of the right tooling to pre-generate the entire *TaskChains* out of the referenced system and thus component and task models. This pre-generation can be implemented as a code-completion step as e.g. shown in the Appendix in Listing A.6.

Another argument in support of the specification of *TaskNodes* is that not all defined tasks are actually time-critical and thus would need to be configured (i.e., refined) in the *performance* view. In other words, the number of *TaskNodes* might only be a sub-set of all the available tasks in all components. For all these reasons and to make the *performance* view self-reliant, the addition of the *TaskNode* specifications has been introduced in the performance meta-model despite the hypothetical redundancy. Owing to the removal of the component boundaries, each *TaskNode* requires a unique name that differentiates it from all the other *TaskNodes*, which are defined in the same global name-space.

InputLink attribute	**InputNode refinement**
optional=**true**	*RegisterInputNode*
optional=**false**	*TriggerInputNode* or *RegisterInputNode*

Table 7.5.: Relation between the InputLink and the InputNode refinements

After the *TaskNodes* have been defined, the next step is to specify and to refine the interactions between individual *TaskNodes*. This can be achieved by specifying different types of *InputNodes* for each individual *TaskNode*. An *InputNode* defines an interaction point of its *TaskNode* with another *TaskNode*. Although *InputNodes* might look like input ports of a component at first glance, they are mapped to a different element, namely the *InputLink*, which is the dependency between a task and an input-port (see also Section 5.3.1). Moreover, there is a relationship (as shown in Table 7.5) between the predefined *InputLink* parameters and the permissible *InputNode* refinements. If an *InputLink* has been specified as optional, the only permissible refinement is a *RegisterInputNode*. For example, the specification of the optional dependency between the *JoystickClient* and the *CDLTask* in Figure 7.4 results in the *Joystick RegisterInputNode* specification of the *CDL TaskNode* in Figure 7.7. This restriction is important to prevent using an optional *InputLink* as a trigger for a *TaskNode*, which would lead to awkward runtime behavior in case the optional *InputLink* is not connected to any publisher (which is a valid case).

available **InputNodes** at a *TaskNode*	permissible **ActivationSource** types
none	*PeriodicTimer* or *Sporadic*
only *RegisterInputNode(s)*	*PeriodicTimer* or *Sporadic*
at least one *TriggerInputNode*	all three *ActivationSource* types

Table 7.6.: Relation between the availability of *InputNodes* of a *TaskNode* and the permissible *ActivationSource* types

The next step is the selection of an *ActivationSource* for each individual *TaskNode*. The choice of a respective *ActivationSource* thereby depends on the availability of relevant *InputNodes* as shown in Table 7.6. For instance, *TaskNodes* such as e.g. *Odom* or *JoystickServer* (in Figure 7.7) that do not have any *InputNodes* can only be triggered by a *PeriodicTimer* or by an own, internal hardware trigger, which is modeled as the *Sporadic* trigger type. The same choices are available in case that a *TaskNode* only specifies *RegisterInputNodes* such as e.g. in case of the *Laser TaskNode* in Figure 7.7. The only permissible choice for a *DataTriggered ActivationSource* is the connection with a *TriggerInputNode* such as e.g. demonstrated with the *CDL TaskNode* in Figure 7.7. However, not every *TriggerInputNode* must necessarily trigger the respective *TaskNode* (i.e., the *TaskNode* can still use another *ActivationSource* despite the availability of *TriggerInputNodes*).

The next two modeling elements are the *Scheduler*, which allows specifying a scheduling policy and a scheduling priority for each individual *TaskNode*, and the *CPUCore*, which allows binding of individual *TaskNodes* to a certain CPU core. As shown in Figure 7.7, the navigation scenario uses the FIFO scheduling policy for all *TaskNodes*, which was reasonable for conducting a Compositional Performance Analysis (CPA) (see Section 7.2). Besides, a single CPU core has been used for all tasks because, on the one hand, the overall CPU load is quite low (around 17% for this scenario), and on the other hand, the sampling and scheduling effects were easier to isolate in the abovementioned CPA.

The model elements that have been explained so far are directly related to the runtime configuration of tasks. This means that out of these modeling elements, configuration files are generated and later are loaded at startup of components. By contrast, the next two modeling elements: *ExecutionTime* and *TaskChain* are rather related to providing additional information that is used for conducting a CPA (see next section). As argued in Section 5.3, the approach in this dissertation is to use profiling for specifying the *ExecutionTime* of *TaskNodes*. Hence, the *ExecutionTime* is directly measured in components during e.g. the testing phase and then directly annotated as modeling parameters in the *performance* view as shown in Figure 7.7.

The remaining (i.e., not yet described) modeling element is a *TaskChain* that references several interlinked *TaskNodes*. *TaskNodes* with interaction links in between altogether form nets of tasks with multiple data-flow paths. In practice, not all of the potential paths are actually time-critical. Therefore, it makes sense to select those paths that are of particular interest for a CPA and to annotate additional attributes for these paths such as e.g. the overall end-to-end *MaxAge* or *Reaction* time as is demonstrated with the *FastReactiveNavigationLoop* in Figure 7.7. These end-to-end times are used as direct application-specific requirements that are compared against the results from a CPA. The results then are either within the specified boundaries or not. In the latter case, the configuration of individual tasks can be changed in such a manner that the overall end-to-end timings are achieved. For example, the *FastReactiveNavigationLoop* in Figure 7.7 requires rather short end-to-end timings. Thus, for the involved *TaskNodes*, the *TriggerInputNodes* with the *DataTriggered* activation-semantics will be preferred wherever possible. In general, using *TriggerInputNodes* with the *DataTriggered* activation-semantics reduces the overall end-to-end time while increasing the overall end-to-end jitter. By contrast, using *RegisterInputNodes* with the *PeriodicTimer* activation-semantics stabilizes the end-to-end jitter at the expense of a greater overall end-to-end data-flow time. While this general relation is rather clear, in practice individual *TaskChains* rarely consist entirely of only synchronous or only asynchronous links. In the end, it is rather difficult to manually estimate the actual end-to-end timings of a *TaskChain* consisting of a combination of synchronous and asynchronous links. For this purpose, a CPA (as explained in the next section) simulates platform-specific runtime conditions (such as scheduling and sampling effects, or network arbitration) and as result calculates the expected end-to-end timings.

Further configuration options that affect the overall end-to-end timings are the distribution of individual *TaskNodes* over several (i.e., parallel) CPU cores or the usage of a different scheduling policy and scheduling priorities. The overall idea is that effects of individual configuration options can be directly investigated by conducting a CPA and investigating the results. Thus, the overall development cycle becomes efficient and performance-related system aspects become manageable early in an overall robotic development workflow.

7.2. SymTA/S Performance Analysis and its Results

The preceding section presented several models of the navigation scenario. One particularity in these models is related to the specification of performance-related aspects that influence the overall end-to-end times in a system. Since one of the general objectives in this dissertation is to reuse the vast knowledge and matured tools from outside domains close to robotics (which is also stated as the Objective 1.4 in Chapter 1), this section conducts a Compositional Performance Analysis (CPA) based on the SymTA/S approach [Hen+05] as an example. SymTA/S simulates the different runtime conditions of a system (such as scheduling and sampling effects, network arbitration, etc.) based on the input information derived from the component and system models that have been presented in the preceding section. As result from the CPA the effects of individual configuration options onto the overall end-to-end timings become visible and clear, which is a first step toward gaining control over application-specific and performance-related system aspects.

For conducting a *structured* CPA this section follows the guidelines proposed by Kitchenham et al. in [KPP95]. In this context, the CPA is structured as a *case-study*. Therefore, the *case-study* states some hypotheses at the beginning that allow evaluation of the CPA results in comparison with ground truth measurements from a Pioneer P3DX robot that has executed the navigation scenario for around half an hour in a realistic environment. There are some comparable *case-studies* such as in [Kol+10] that give first insights about the usefulness of a CPA—using SymTA/S among others—within the automotive domain. However, this *case-study* is not necessarily applicable to the domain of service robotics. As stated in [Lot+16], robotics differs from automotive in terms of the following: operating system (Linux vs. AUTOSAR/OSEK), scheduling (dynamic vs. static), memory management (dynamic memory allocation on heap vs. static memory allocation), and types of processes (POSIX-compliant tasks with multiple threads and memory protection vs. mostly single-threaded OSEK basic tasks, some of them without memory protection).

A summary of the results has already been published in the core publication [Lot+16]. This section uses the same results, however with a much deeper level of details with respect to the collected data, results of the analysis, and derived conclusions. The two main objectives of this *case-study* is to (i) evaluate whether the proposed abstraction level of the *performance* view is adequate (i.e., detailed enough) for conducting a CPA and (ii) whether the selected SymTA/S & Trace Analyser tool yields helpful and realistic results also for robotic scenarios.

7.2.1. Case-study context

The *case-study* context is the navigation stack that has been executed on the Pioneer P3DX mobile robot in a home-like environment in our robotic lab in Ulm (see Figure 7.9).

The navigation stack has been selected for the following reasons. First, it represents a sub-set of components that have been used in many other, more complex scenarios in our past projects and demonstrations. One of such demonstrations is for instance our *collaborative robot-butler scenario*[3], where two robots collaborate in making and bringing coffee (by operating a coffee machine, opening a cupboard, etc.). Both robots have been using the same navigation components (except for the exchanged hardware-related components such as the *SmartRMPBaseServer* instead of the *SmartPioneerBaseServer* due to a different base platform). Altogether, the navigation stack provides enough variability and complexity to demonstrate the different architectural configuration options, while at the same time the overall features, requirements and limitations of the scenario are well understood.

[3]Robot-butler video: `https://youtu.be/DjjNUPpj36E`

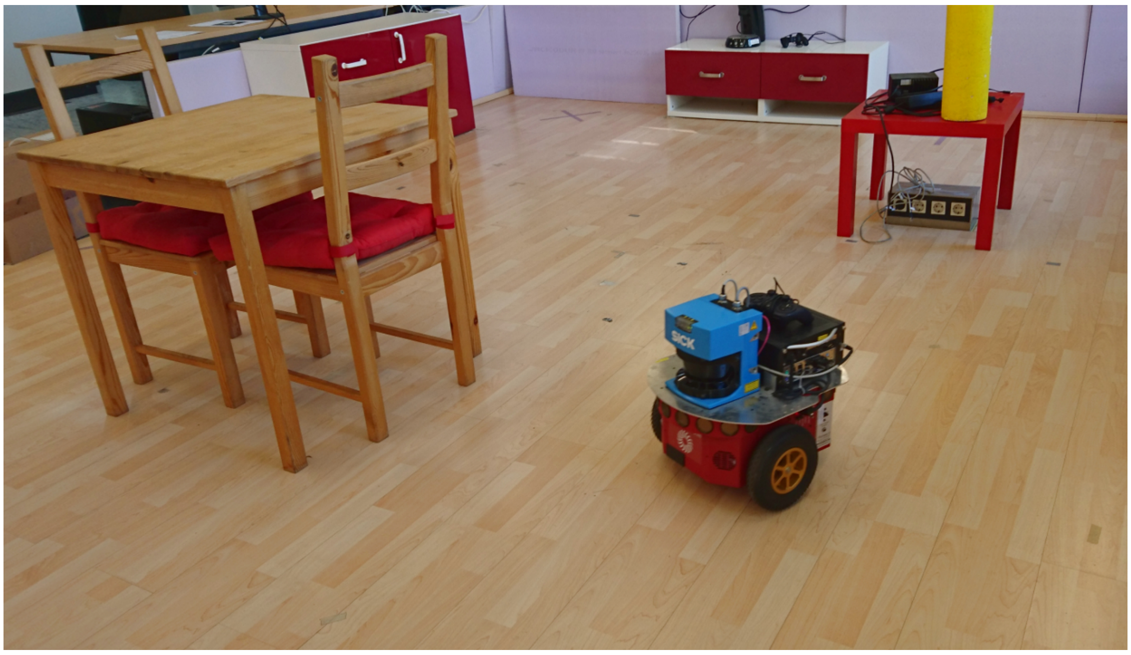


Figure 7.9.: Pioneer P3DX moving in a home-like environment

Moreover, the navigation components have been executed on Ubuntu Linux 12.04 using the FIFO real-time scheduler that is provided in Linux. This allows executing the navigation components with a higher priority than any other processes in the system. Thus, all other components that might be executed in addition to the navigation components are executed in the spare time not used by the real-time scheduler and thus do not affect the presented performance results. As for the target platform, a common mini-ITX PC with a regular Intel dual-core CPU has been used. This platform is considered representative for many robotic applications where best-effort is sufficient enough and only small parts of the overall system are designed with hard real-time guaranties in mind. While there are many works addressing the latter part (see [BS05] for an overview), even for the best-effort, it still is of value to design and to know the overall end-to-end times (even if these times might not be reached in all cases, which might in average still be acceptable). There are various possible gradations in between the best-effort and hard real time. As one of the extreme examples, the navigation stack has also been used on a PowerPC CPU with the QNX OS in one of our past industrial collaboration projects.

In terms from [KPP95], the conducted *case-study* consists of a single project, performed by a single person. However, even this single project offers enough variation that comes from the different configuration options of components and tasks. For instance, each of the 8 *TaskNodes* in Figure 7.7 can be theoretically configured in three different ways with respect to selecting an

ActivationSource. This results in a combinatorial total of $3^8 = 6561$ theoretically possible configuration alternatives. Of course, due to the specified *ActivationConstraints* in individual components the practically reasonable configuration options result in a total of 16 alternatives. The modeling tooling supports in filtering out all the unreasonable alternatives using model checks and code completions. Yet, even the 16 remaining configuration alternatives leave open enough design-space to optimize the overall execution performance of a system. Testing each of these 16 alternatives in a real system would be a huge effort though, because for each alternative one would need to (1) configure the components accordingly, (2) deploy them into the target, (3) execute the robot for a realistic period of time and then to (4) analyze the log files afterward. Obviously, this is too much effort to just determine the effects of the different configuration options. Therefore, using an analysis tool that is able to simulate the effects of different configurations in a few seconds is reasonable.

The presented navigation scenario (shown in Figure 7.7) consists of three *TaskChains* (i.e., data flow paths of particular interest), namely the *FastReactiveNavigationLoop*, the *PlannedNavigationLoop* and the *JoystickLoop*. The first *TaskChain* selects those *TaskNodes* that are directly related to the local collision-avoidance functionality. This means that the configuration of the involved *TaskNodes* directly impacts onto the actual reaction time of the robot in case of an obstacle suddenly appearing in front of the robot. For a real application, an overall admissible breaking distance S_{break} for a robot could be determined by the following formula:

$$S_{break} = V_{max} \cdot t_{react} + \frac{V_{max}^2}{2 \cdot a_{decel}} \tag{7.1}$$

The maximum allowed velocity V_{max} is a safety-critical application requirement that involves trading off between maximizing the overall (physical) performance of the robot and at the same time restricting the velocity value due to safety considerations. Moreover, the deceleration a_{decel} constant is defined by the kinematic constraints of a real robot (i.e., its weight and maximum possible torque without the wheels to start slipping or preventing the danger of the robot falling over). The only unknown variable is the reaction time t_{react}, which is basically the end-to-end *MaxAge* time of the *FastReactiveNavigationLoop* in our example. From the system design point of view, an application designer might decide on a certain admissible t_{react} to achieve an acceptable maximal breaking distance S_{break}. However, whether the system will actually respond within that time or not is not obvious just from investigating the model. In conclusion, this time needs to either be tested in a real system or calculated within a performance analysis as shown in this section further below. For this purpose, as part of the *case-study* the overall end-to-end time of the *FastReactiveNavigationLoop* will be determined in both, simulation, and reality.

7.2.2. Setting the Hypothesis

The overall goal of the *case-study* is to evaluate whether a SymTA/S-based performance analysis sufficiently predicts the actual execution performance of a real robot. In order to assess this, the following two hypotheses are specified:

Hypothesis 7.1: *The SymTA/S & Trace Analyser yields realistic results for robotic applications and systems*

Hypothesis 7.2: *Different configuration alternatives in the performance model similarly affect the overall end-to-end timings in both the SymTA/S-based CPA and in reality*

For evaluating these two hypotheses, the upper and lower bounds of the simulated and measured end-to-end response-time values are compared by accordingly using two metrics m_1 and m_2. Metric m_1 denotes the distance between the actually measured minimal end-to-end response time on the robot rt_{min} and the simulated Best-Case Response-Time (BCRT) from the SymTA/S-based analysis: $m_1 = rt_{min} - \text{BCRT}$. Metric m_2 denotes the distance between the simulated Worst-Case Response-Time (WCRT) and the actually measured end-to-end response time rt_{max}: $m_2 = \text{WCRT} - rt_{max}$.

For validating the Hypothesis 7.1 the two metrics m_1 and m_2 are normalized according to the overall distribution of simulated end-to-end times RT_{dist}, which is calculated by $RT_{dist} = \text{WCRT} - \text{BCRT}$. This results in two derived metrics d_1 and d_2 that are calculated as shown in Equation (7.2). The Hypothesis 7.1 is considered fulfilled if both distance metrics d_1 and d_2 are below a threshold of 10%.

$$d_1 = \frac{rt_{min} - \text{BCRT}}{\text{WCRT} - \text{BCRT}} = \frac{m_1}{RT_{dist}} \;\Big|\; d_2 = \frac{\text{WCRT} - rt_{max}}{\text{WCRT} - \text{BCRT}} = \frac{m_2}{RT_{dist}} \tag{7.2}$$

For validating the Hypothesis 7.2, the navigation performance model is adjusted in such a manner that one of the intermediate *TaskNodes* of the *FastReactiveNavigationLoop* changes its activation semantics from a synchronous to an asynchronous mode. Again, the same distance metrics d_1 and d_2 are calculated and if they are within the 10% threshold, then the Hypothesis 7.2 is considered fulfilled as well.

7.2.3. Preparing the case-study

In preparation of the *case-study* all *TaskNodes* in Figure 7.7 have been assigned to one CPU core and configured to use the FIFO scheduling strategy (with the same FIFO priority for all specified *TaskNodes*). This has the advantage that, on the one hand, all specified tasks are automatically executed with a higher priority than any other task or process in the system and on the other hand, the measurements from the robot can be used for two purposes: (1) to incorporate the measured execution times back into the performance model in Figure 7.7 and (2) to calculate the real end-to-end times for the relevant *TaskChains*. Because of the FIFO scheduling strategy, all *TaskNodes* always execute their individual cycles to completion without being interrupted or preempted by other tasks, which leads to realistic execution-time measurements for (1). For deriving the real end-to-end times in (2), a simple MATLAB script is used that concatenates the individual “raw” execution times according to the used trigger semantics. Therefore, the individual messages that flow through the specified *TaskChains* are traced using the mechanism described in Section 6.2.2.

To execute the navigation scenario, a simple high-level coordination component has been implemented (not further detailed here) that randomly generates different locations (in the home-like environment in Figure 7.9) for the robot to drive to. While driving, several dynamic obstacles and blockages have been put in front of the robot to trigger different strategies for obstacle-avoidance and path-(re)planing and thus to simulate a realistic operation.

7.2.4. Validating the Hypothesis

For deriving the metrics m_1 and m_2 the navigation scenario has been executed on the robot for 21 minutes that resulted in a total of 11726 overall cycles of the *FastReactiveNavigationLoop*. In the second step, the logged execution times have been incorporated back into the performance model. The figures A.1 to A.4 in the Appendix A.8 show the histogram plots of the execution-time distributions and update-frequencies for each individual *TaskNode*. Then the performance model is transformed using (1) a model-to-model transformation step and then (2) a model-to-text transformation step (as described in Section 6.1) into a representation (i.e., a Python script) that is interpretable by the SymTA/S & Trace Analyser. After that, a CPA is triggered in the tool, which basically calculates the overall end-to-end times for the three *TaskChains* in Figure 7.7 (on the right). In parallel, the real end-to-end times are determined out of the logged data using a simple MATLAB script (not further detailed here).

As argued in Section 6.2, the execution time of each individual *TaskNode* includes the communication time of publishing the task's results over its associated output port(s). Hence, the performance analysis is realistic without the need to model every single platform detail at the lowest abstraction level. An obvious disadvantage is that the effects from computation and from communication are not distinguished in the performance analysis. Either way, the results only deviate in the level of details. In order not to make the approach too complex from the very beginning, simplicity has been preferred in the initial implementation over completeness (in the sense of a possible level of details). Moreover, from the first impression of using the SymTA/S & Trace Analyser tool for robotic applications, it seems that this approach is sufficiently rich for many robotic cases and already provides great assistance. However, in future works the model-to-model transformation might also include the generation of additional platform details with respect to communication.

In order to evaluate the two hypotheses 7.1 and 7.2, the navigation scenario has been executed and evaluated twice, with the difference that the activation-source of the *CDL TaskNode* in Figure 7.7 has been changed from *DataTriggered* to *PeriodicTimer*, which allows investigating the effects of this modeling alternative onto the overall end-to-end execution time. Figure 7.10 shows the MATLAB plots that are derived from real measurements of the *CDL TaskNode*. The two diagram plots on the top in the figure are the results from the first scenario run with *CDL* using the *DataTriggered* activation semantics, and the two bottom plots are respectively the *CDL TaskNode* with the *PeriodicTimer* activation semantics. Figure 7.11 shows the same comparison as in Figure 7.10, this time however, using the results from the SymTA/S-based CPA.

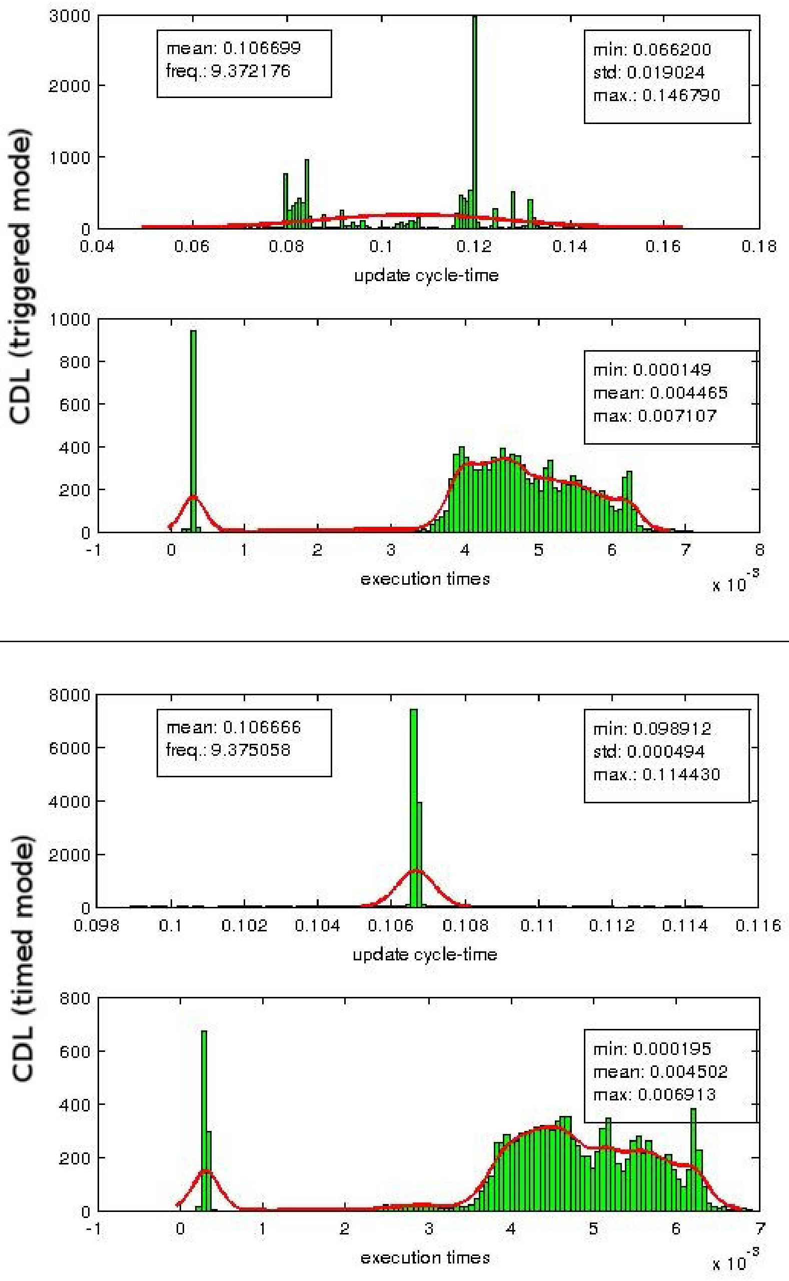
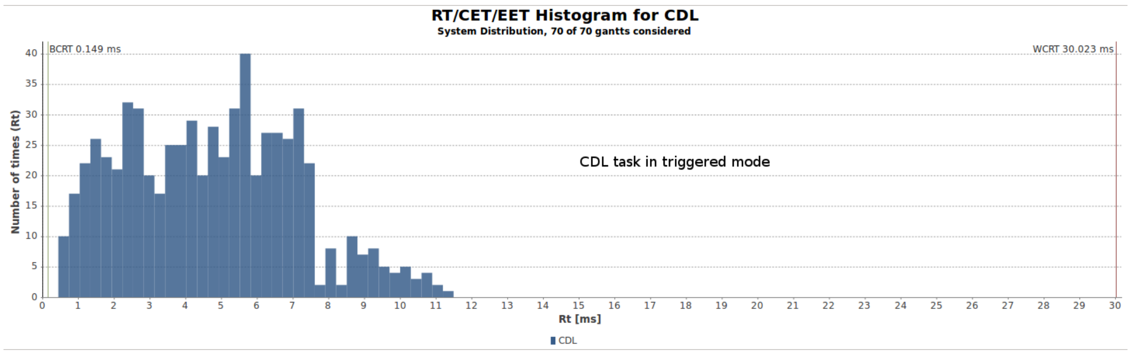


Figure 7.10.: Histogram plots of the logged execution times for the CDL task with two different configurations



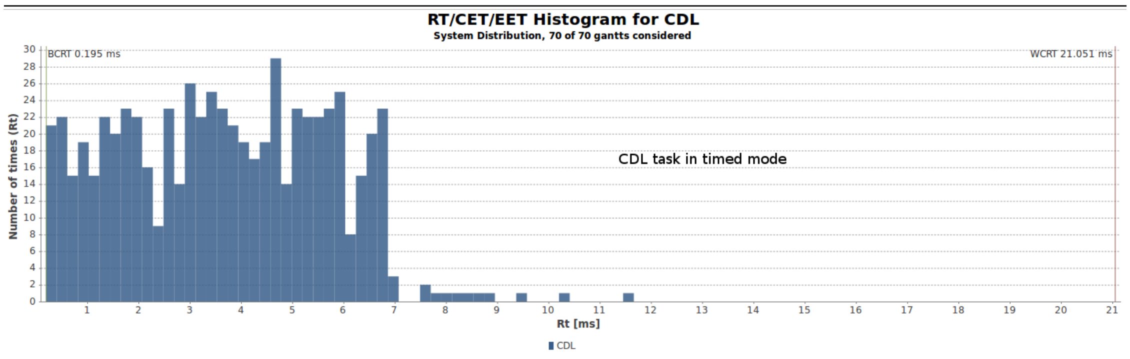


Figure 7.11.: Histogram plots of the simulated execution times (using SymTA/S) for the CDL task with two different configurations

The next two figures 7.12 and 7.13 show the distribution of accumulated end-to-end times for the *FastReactiveNavigationLoop*, again for both scenario runs. In the first scenario run, the measured overall end-to-end times are $101.749ms$ (min) and $249.114ms$ (max). The accordingly simulated values are $0.478ms$ (BCRT) and $267.995ms$ (WCRT). First, an anomaly can be observed in Figure 7.13 (on top) showing a gap of around $100ms$ between the BCRT time and the lowest histogram bar in the diagram. The reason is that due to the $MaxAge$ semantics all simulated values by default consider the worst case, where the first *TaskNode Odom* of the *FastReactiveNavigationLoop* automatically adds its update cycle time to the overall end-to-end time. Because the *Odom TaskNode* is configured to use a *PriodicTimer* with an update frequency of $10Hz$, this results in the initial delay of $1/10Hz = 100ms$, which explains the graphical gap on the left side of the histograms in Figure 7.13. As for the BCRT, it considers the overall (theoretical) best case, where the first *Odom TaskNode* finishes its cycle just right before the next *Laser TaskNode* begins its own asynchronous cycle. In order to make the simulated and the measured results comparable, the initial delay of $100ms$ has been added (as a static offset) to all measured values. However, in order to accurately calculate the metric m_1, this offset needs to be subtracted once for the minimal measured value that altogether equals to $m_1 = (rt_{min} - 1/10Hz) - \text{BCRT} = (101.749ms - 100ms) - 0.478ms = 1.001ms$. The calculation of the metric m_2 is straightforward $m_2 = \text{WCRT} - rt_{max} = 267.995ms - 249.114ms = 18.881ms$.

The overall distribution RT_{dist} of simulated end-to-end results is $RT_{dist} = \text{WCRT} - \text{BCRT} = 267.995ms - 0.478ms = 267.517ms$. Now, the distance values can be calculated by the formula:

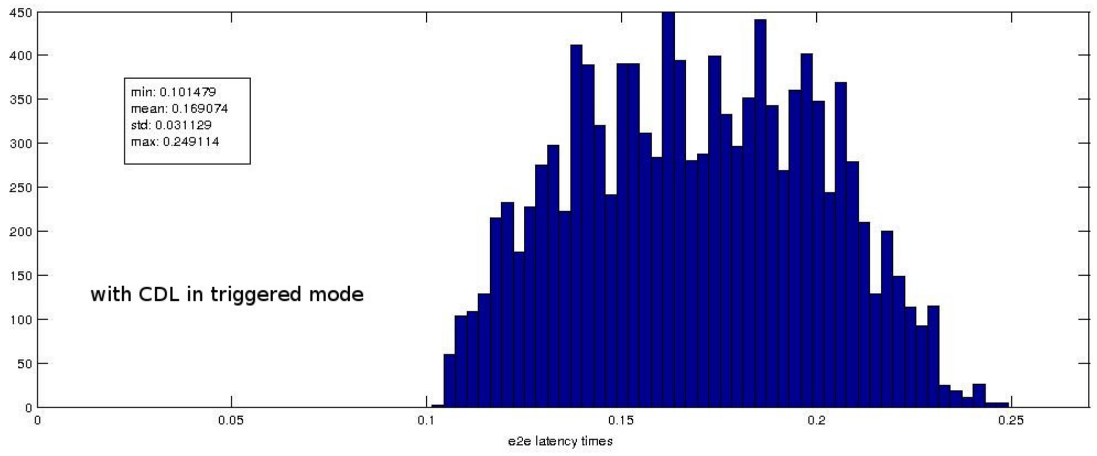
$$d_1 = \frac{1.001}{267.517} = 0.0037 \equiv 0.37\% \;\Big|\; d_2 = \frac{18.881}{267.517} = 0.070578 \equiv 7.06\% \tag{7.3}$$

Both distance values $d_1 = 0.37\%$ and $d_2 = 7.06\%$ are below the 10% threshold, which confirms the initial Hypothesis 7.1. For evaluating the Hypothesis 7.2 the activation semantics of the *CDL TaskNode* has been changed from *DataTriggered* to *PeriodicTimer* with an update frequency of $9.375Hz$, which roughly represents the same, measured update frequency from the first scenario run (see top diagram-plot in Figure 7.10). The two distance metrics now have the following values:

$$d_1 = \frac{(105.411 - 100) - 0.525}{283.82 - 0.525} = \frac{4.886}{283,295} = 0.01718 \equiv 1.72\% \tag{7.4}$$

$$d_2 = \frac{283.82 - 268.513}{283.295} = 0.05403 \equiv 5.4\% \tag{7.5}$$

Again, as in the scenario run before, both distance values are within the 10% threshold, which confirms the Hypothesis 7.2. An explanation and interpretation of the results is provided in the next section.



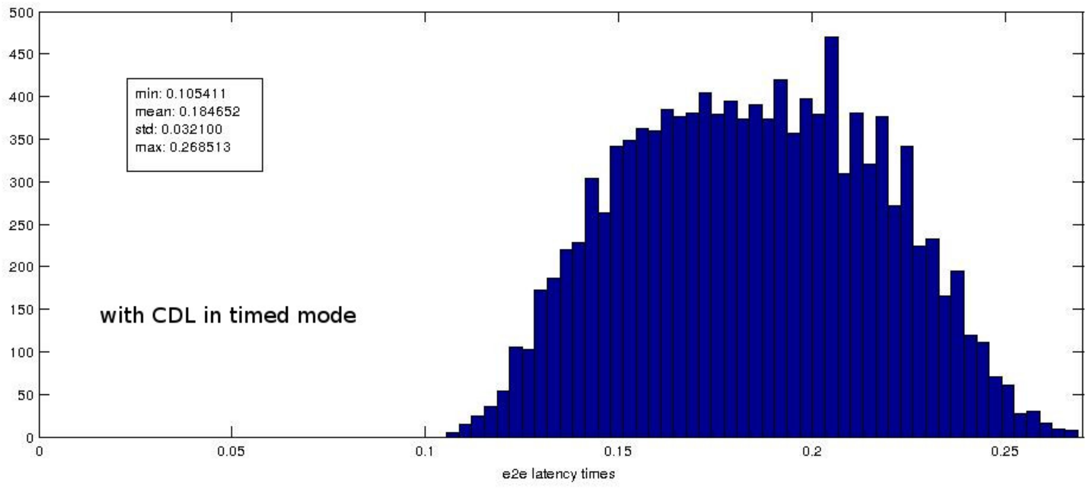


Figure 7.12.: Histogram plots of the overall end-to-end data-flow times (based on logged raw data that has been aggregated using a MATLAB script) for the FastReactiveNavigation loop with two different configurations of the CDL task

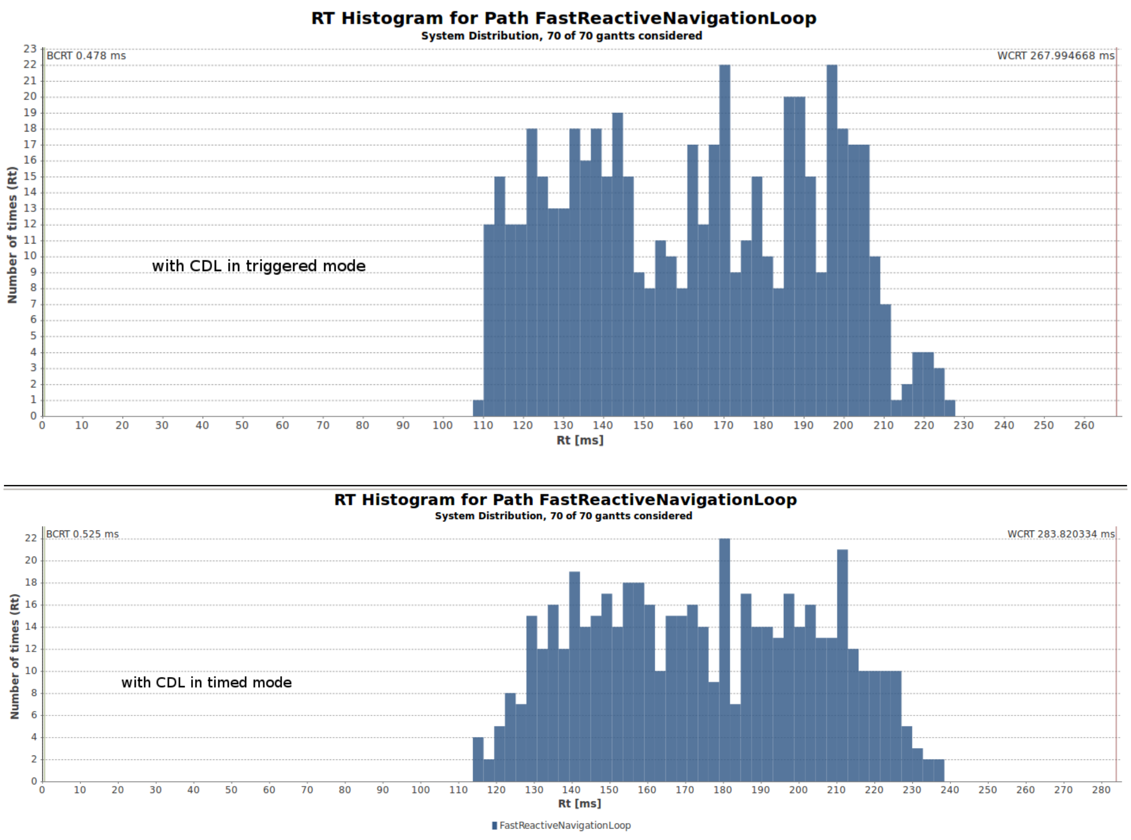


Figure 7.13.: Histogram plots of the simulated overall end-to-end data-flow times (as result from the SymTA/S performance analysis) for the FastReactiveNavigation loop with two different configurations of the CDL task

7.2.5. Analysing the results

The objective metrics m_1 and m_2 already confirm both hypotheses 7.1 and 7.2. However, the results in the diagrams in figures 7.10 to 7.13 include a lot more information about the overall behavior and performance of the navigation scenario, which are discussed in this section hereinafter.

At the beginning of this overall section, the Equation (7.1) to calculate the maximal breaking-distance has been presented. Knowing the WCRT value, the breaking distance value can be calculated for the presented navigation scenario. Before doing that, the maximal velocity and deceleration values need to be defined. The data-sheet of the Pioneer P3DX robot reveals a maximal possible velocity of $1.4m/s$ and no limitation for the deceleration value (i.e., in theory the robot can stop virtually instantly). In practice and for realistic scenarios these values are typically considerably reduced, e.g. because often additional equipment is mounted on the robot platform, which increases the overall payload and heightens the center of mass. For the scenario runs in the above *case-study*, the maximal velocity has been reduced to $1.0m/s$ and deceleration to $400mm/s^2$. With these values, the maximal breaking distance S_{break} results in the following value:

$$S_{break} = V_{max} \cdot \text{WCRT} + \frac{V_{max}^2}{2 \cdot a_{decel}} = 1.0m/s \cdot 0.268s + \frac{(1.0m/s)^2}{2 \cdot 0.4m/s^2} = 1.518m \tag{7.6}$$

Overall, a maximal breaking distance of around $1.5m$ is acceptable in many scenarios. Interestingly, in the initial configuration of the navigation scenario, the influence of the software-related reaction time is less than a third of the overall breaking distance, which appears to be a good value. In case that an even slower reaction time is acceptable then a slower CPU can be used, other components can be executed in parallel and alternative task-configurations can be selected. For instance, even the slightly less optimal configuration in the second scenario run with a WCET of $283.82ms$ results in the braking distance of $1.533m$, which only marginally increases by around $15mm$. Of course, depending on the specified update-frequencies and activation-semantics of individual *TaskNodes* this difference might increase a lot more.

Another interesting observation is the effect of switching between the different activation semantics of a task. For instance, the *CDL* task has been reconfigured from using a *DataTriggered* activation semantic toward using a *PeriodicTimer*. As a result, the overall WCET increases (compare the plots in Figure 7.12), but at the same time the jitter in the update frequency of the *CDL* task considerably decreases as can be seen by comparing the first and the third plots in Figure 7.10. This is because in the first configuration the *CDL* task follows the update frequency of the preceding *Laser* task, which reduces the latency but at the same time accumulates the jitter. In the second configuration, the *CDL* task becomes independent of the preceding *Laser* task, which considerably reduces the jitter, but in the worst case an additional latency of one update cycle needs to be included. In conclusion, none of the above two configuration options is universally good or bad. It rather depends on what is more important in the current application, i.e., a rather stable and deterministic execution behavior, or rather short overall end-to-end latencies. Therefore, being able to influence these options and to directly see the impacts onto the overall execution performance is considered a great help in - and a general improvement of the overall development process.

As also shortly discussed in the analysis section of [Lot+16], the shapes of the simulated and measured histogram plots (compare figures 7.12 and 7.13) are quite different. The main reason is related to the simulated distribution of execution times for a *TaskNode* in the SymTA/S & Trace Analyser. More precisely, SymTA/S by default assumes and simulates a uniform distribution of execution times for each task between the given min/max boundaries. By contrast, real-world distributions are not uniform but have peaks and gaps. This difference does not affect the best-/worst-case calculation though and only leads to a different distribution in-between. Since the corner cases are not affected by the distribution, the shape of the histogram plots can be considered ornamental. However, it is also conceptually straightforward to add other distributions (such as Gamma or Weibull) to the SymTA/S & Trace Analyser to better reflect the actual execution characteristics of real algorithms [Lot+16].

7.3. Summary and Discussion of the Scenario Results

Overall, this chapter presented and discussed the navigation scenario as a representative for common robotic applications. This scenario has been used for the following main purposes:

1. to assess that the design and implementation of individual software components is possible using the presented component model abstractions and that the design and configuration of *cause–effect chains* is practically feasible at the model level only (i.e., without the need to investigate individual component's internal implementations)

2. to further demonstrate and to empirically evaluate that the chosen level of granularity and abstraction of the *cause–effect chains* is reasonable for conducting a Compositional Performance Analysis (CPA)

To begin with, the navigation scenario has been selected for various reasons. Among others, all the functional building blocks of the navigation scenario already existed beforehand and were easily accessible and modifiable for the scope of this dissertation. This allowed to test the developed modeling tools in all relevant phases of the overall robotic development process. This gives a certain confidence that none of the important functional and non-functional aspects have been overseen (except those that have been explicitly excluded on purpose).

As a general result for (1), all the components of the navigation scenario have been successfully ported using the novel modeling tools and all relevant functional boundaries with respect to components' performance related constraints have been made explicit as part of the component models. Moreover, the system integration phase completely relies on component models only for instantiating, configuring, and composing the components within the navigation scenario. Three *cause–effect chains* have been identified, modeled, and configured, while the previously defined constraints from the individual component models have also been considered.

While the navigation scenario certainly is a good representative for typical robotic applications, ultimately it still is only one single scenario so far that has been designed using the proposed modeling tools. Nevertheless, this scenario alone covers many common and recurring design problems related to system-level performance aspects, which lends some confidence with respect to the universality of the proposed approach. Yet, modeling further examples will allow refinement and improvement of the overall structures and abstractions.

As for (2), the three modeled *cause–effect chains* have been directly connected with (a) the actual configurations of components in an automated model-to-text transformation step and (b) a Compositional Performance Analysis that can be easily triggered in an automated model-to-model transformation step. Regarding (a), this shows that the chosen abstractions not only are theoretically sound but also are practically feasible. As for (b), this practically demonstrates that even tools such as SymTA/S & Trace Analyser that have been designed for a different domain can still be integrated and used as part of the overall robotic development process. Roboticists are not burdened with a steep learning curve. This is important because they are mainly interested in the analysis results and not the underlying scheduling mechanics or the analysis capabilities of the selected analysis tool. As shown in the next Chapter 8, the proposed approach can also be easily mapped to an entirely different analysis approach such as AADL flow-latency analysis using the OSATE2 tool. Overall, the proposed approach enables roboticists to gain from the analytic power of matured analysis tools and makes them easily accessible within the robotic community.

There is another interesting observation from logging the end-to-end latencies directly in the system. The measurements of execution times for individual tasks are mostly straightforward and a relevant logging infrastructure can even be automatically generated into each component by default. However, the calculation of the end-to-end times—using a MATLAB script—has been very tedious and error-prone. In this dissertation, these calculations have only been done to get ground-truth values for comparison, which would be entirely impractical as part of a regular development workflow. For a regular development workflow a CPA is much more appropriate because it allows simulation of the scheduling and sampling effects and provides the overall results with an acceptable precision in a few seconds in contrast to recording and manually evaluating the log-files, which took a couple of weeks (which would need to be repeated for each new application).

8 Borderline Topics and Future Works

Working on a certain scientific topic for a longer period of time gives one unique perspective with a deep understanding of alternative approaches and potential solutions. It is thus only natural that at the end of an endeavor such as this dissertation new ideas emerge and a lot more interesting research questions arise. At some point, it is necessary to restrain the overall scope in order to allow finishing a dissertation in a limited amount of time. Therefore, this chapter covers some borderline topics that could not be fully investigated as core topics of this dissertation, yet provide enough insides for follow-up research directions.

8.1. Recent Modeling Refinements

To begin with, the presented modeling tools go through continuous refinements and evolutions (that will continue even after this dissertation). These refinements can be considered as extensions to the presented consistent set of modeling views. This section shows some recent meta-model extensions that are interesting enough to be presented.

Figure 8.1 shows an extended and refined *component* meta-model (compare with Figure 4.6). In summary, the extensions comprise a new *UpcallHandler* as an alternative for a lightweight *Task* that does not need its own thread of execution but provides the same structure and API. Moreover, *Tasks* and *UpcallHandlers* within a component can now interact with each other. For this purpose, an *Observer* design pattern is used, whose infrastructure is generated into the component implementation (which reduces the error-prone handwritten parts).

Figure 8.2 shows the new model of the *SmartPioneerBaseServer* as an example (compare Figure 7.1 on the left). In this example, the interaction between the input-port(s) and the *RobotTask* is not performed over additional tasks; instead, upcall-handlers are used. There are two main reasons to prefer upcall-handlers instead of other task-definitions. First, upcall-handlers are extremely lightweight entities that are synchronously triggered each time a new message arrives at the associated input port. If the business logic is lightweight as well, as in the *LocalizationUpdateUpcall* which only propagates a copy of the message to the *RobotTask*, then the usage of an upcall-handler considerably saves resources. Second, an upcall-handler semantically defines an

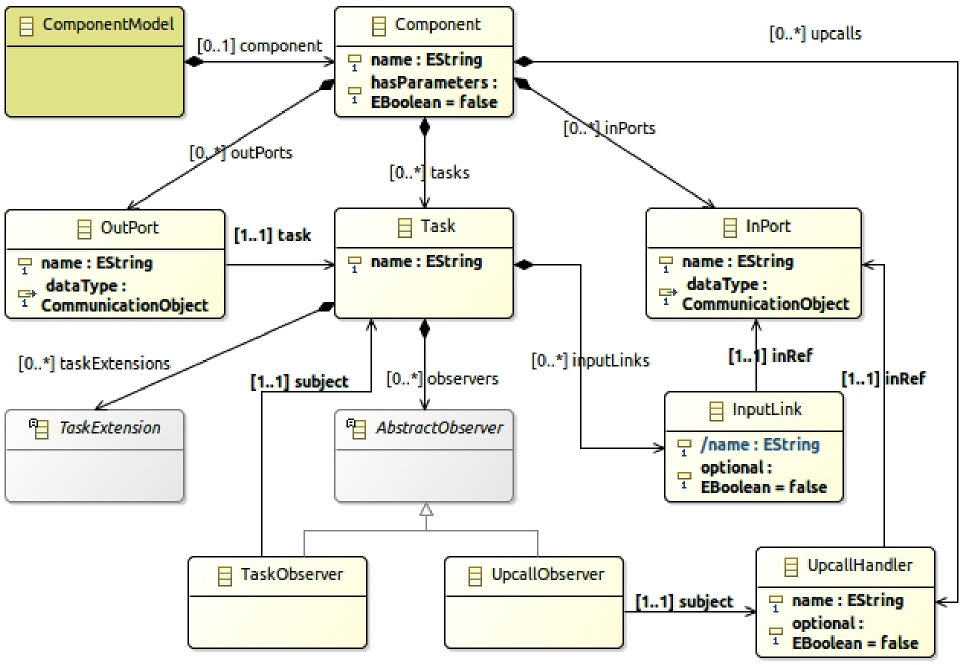


Figure 8.1.: Refined component meta-model including upcall handlers and task/upcall observer pattern

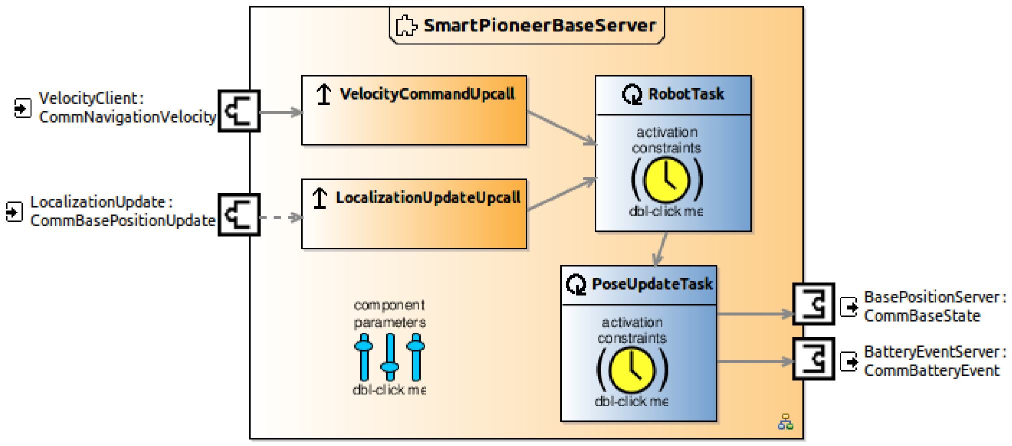


Figure 8.2.: Component diagram for SmartPioneerBaseServer

endpoint which cannot be used to generate or propagate data to other components (by contrast, a task can do this over an output port). Yet, an upcall can define optionality for input data. For example, the *SmartPioneerBaseServer* component cannot function correctly without regularly receiving navigation command updates, but the component works fine without receiving localization updates (although the odometry precision degrades accordingly). This latter optionality of data is visualized in the component diagram (in Figure 8.2) by a dashed arrow from the input port *LocalizationUpdate* toward the *LocalizationUpdateUpcall* handler. All these design choices either restrict or open up design flexibility for the later system-integration phase, where system integrators can adjust the component configurations without violating any functional constraints.

An arrow between an upcall-handler and a task, or between a task and another task (see Figure 8.2), represents local interaction between upcalls and tasks. For each such arrow, an observer design pattern is generated where the source entity (task or upcall) of the arrow implements the subject (or model) part while the destination of the arrow (another task) implements the observer part. In other words, the observer task implements an update method, which is called by the source entity each time the source entity finishes its current cycle. For example, each time the *VelocityCommandUpcall* handler is triggered from the *VelocityClient*, it directly propagates the received message to the *RobotTask* (which stores a copy of this message as its local member). The *RobotTask* itself triggers an update method of the *PoseUpdateTask* each time the *RobotTask* finishes its current cycle.

On the whole, the upcall semantics can be considered as a useful comfort feature that seamlessly extends the meta-models without breaking the semantics presented so far.

8.2. Conceptual Mapping Toward AADL

One of the recent works that has been conceptually elaborated without providing a profound implementation is the option to use AADL [AAD04] for modeling robotic systems. AADL is a holistic modeling language originating in the avionics domain for modeling time-critical systems on all levels. One particularly interesting extension of AADL for the scope of this dissertation is the specification of AADL-Flows [Han07]. As demonstrated in [BFA14], AADL-Flows can be used for expressing and analyzing the end-to-end times, which is generally comparable to the performance analysis in this dissertation. However, in contrast to the separated, focused robotic-specific views in this dissertation, AADL can be considered as a General-Purpose (Modeling) Language (GPML). In this sense, AADL does not impose a specific structure and consequently does not provide much help in coming up with consistent overall systems in an overall, structured development workflow that supports the separation of roles. Owing to the high flexibility and generality of AADL, it can be used as a generic bottom layer, underneath the presented modeling views. In other words, just like the robotic modeling views are transformed into source code and into other, tool-specific representations, the same modeling views can be transformed into an AADL representation. To assess whether this can be achieved, this section presents a conceptual transformation of the presented navigation scenario into an AADL representation.

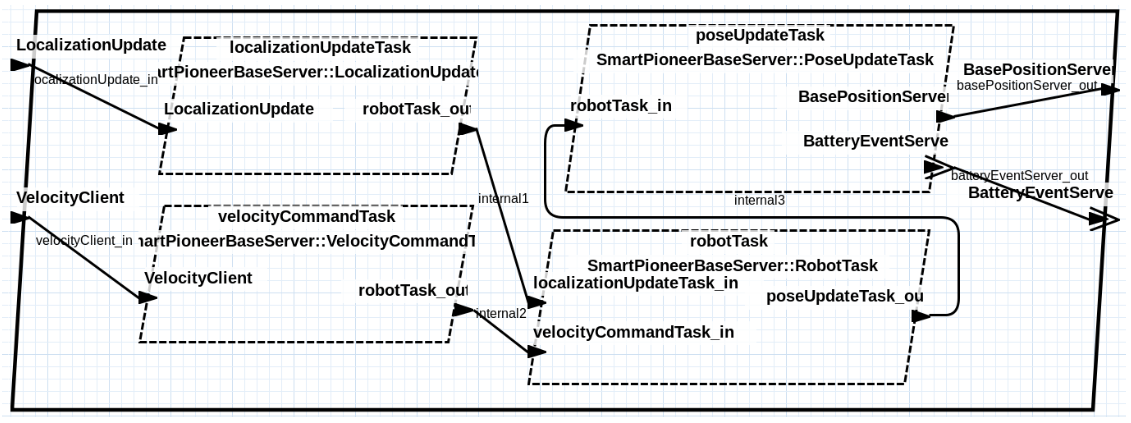


Figure 8.3.: SmartPioneerBaseServer (modeled with the AADL OSATE2 tool)

Figure 8.3 shows the representation of the *SmartPioneerBaseServer* component (as an example) using AADL. Briefly, the main component-elements can be transformed into AADL as shown in Table 8.1. In addition, in case that the component's internal tasks need to interact with each other, they need to specify additional in/out data ports and wire them accordingly.

Component meta-model element	**AADL** model element
Component	*process*
Task	*thread*
InPort	*in data port*
OutPort	*out data port*

Table 8.1.: Mapping between Component and AADL model elements

Figure 8.4 shows the AADL representation of the system configuration and deployment diagrams for the navigation scenario (cf. Figure 7.5 and Figure 7.6). Basically, AADL does not distinguish between the system definition/configuration and deployment, so both views are combined in a single view. Other than that, the mapping is rather straightforward: the component processes are instantiated in an AADL *system* along with an AADL *processor* and *bus*.

It becomes more interesting when the *performance* view from Section 5.3 is transformed into an AADL representation. Using the navigation scenario as an example, this transformation can be done in two steps. First, as shown in Figure 8.5 (diagram on top), the previously defined component *threads* are extended using derivation. This means that new (local) threads derive from the existing component threads, adding new attributes (such as the according activation-source and update-frequency).

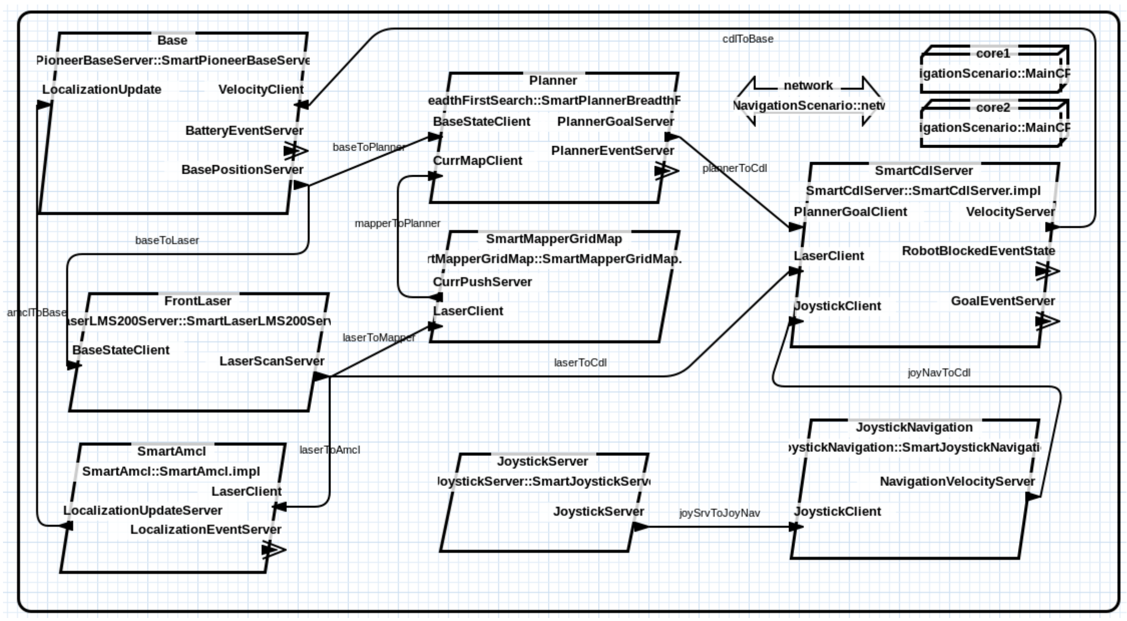


Figure 8.4.: Navigation-scenario system-configuration view (modeled with AADL OSATE2)

Second, as shown in Figure 8.5 (diagram on bottom), data flows are specified, both within the derived threads and in a new “dummy” system (which is only required as a composite for the derived threads but has no real representation in the actual implementation). The “dummy” system helps specify overall end-to-end flows that concatenate the individual thread’s data flows into paths. Moreover, the synchronous and asynchronous interaction semantics are reproduced in AADL using the connection-attribute *immediate* or *delayed* between the involved threads. Combined with the *dispatch protocol* set to *periodic* and an accordingly specified *period* for each derived thread, this generally resembles the *PeriodicTimer* and *DataTriggered* semantics of the *performance* view in this dissertation. However, the *performance* view is more focused and straightforward than the individual modeling parameters in AADL, which are distributed over several model elements (which make the specification of consistent AADL models unnecessarily complex). This focused *performance* view, which is consistent in terms of construction, is specifically important for robotic experts, who need to be efficient in the development of robotic systems (see also Chapter 3).

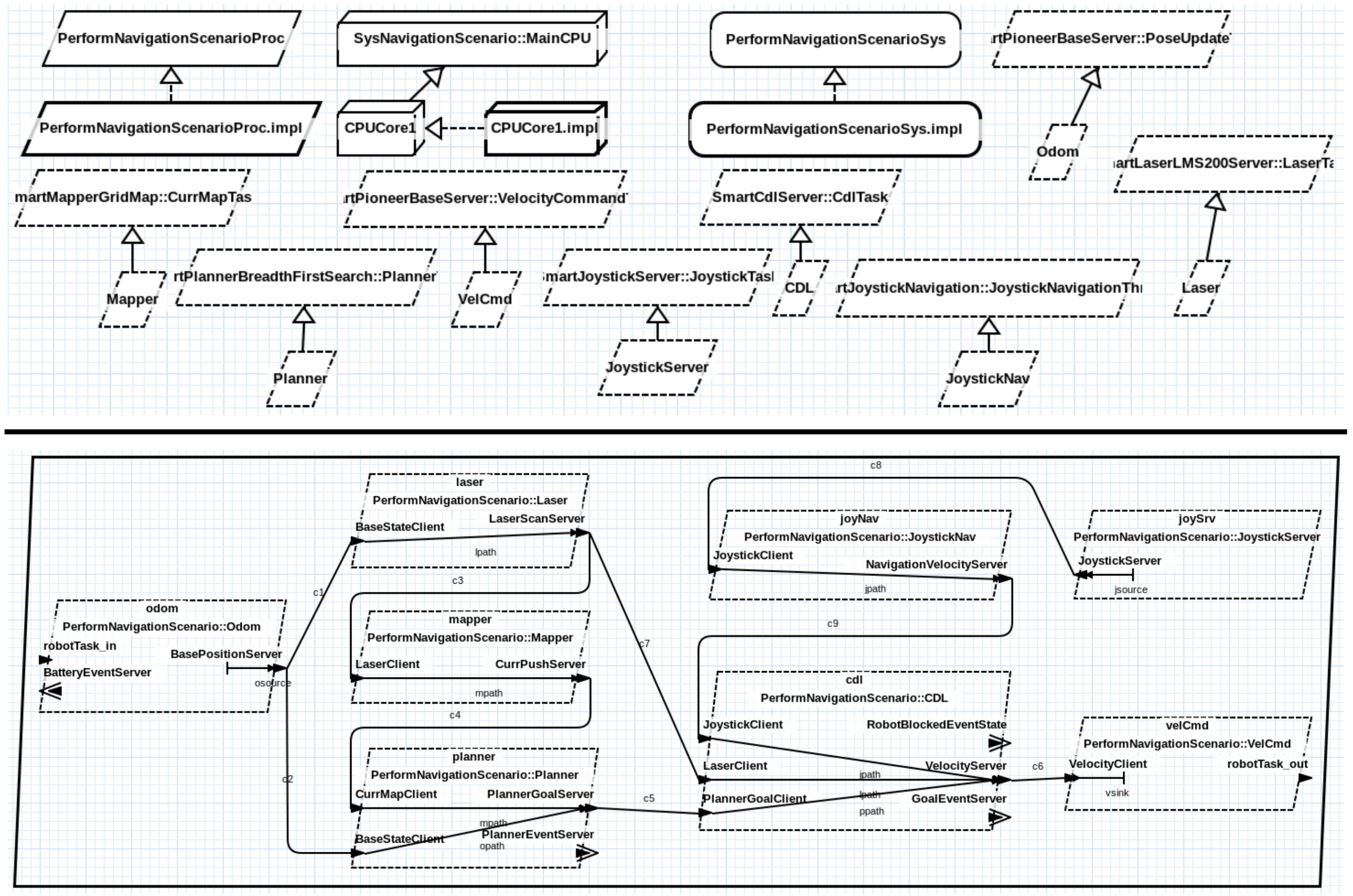

PerformNavigationScenarioProc
PerformNavigationScenarioProc.impl
SysNavigationScenario::MainCPU
CPUCore1
CPUCore1.impl
PerformNavigationScenarioSys
PerformNavigationScenarioSys.impl
rtPioneerBaseServer::PoseUpdate
Odom
artLaserLMS200Server::LaserTa
martMapperGridMap::CurrMapTas
rtPioneerBaseServer::VelocityCommand
SmartCdlServer::CdlTask
Mapper
rtPlannerBreadthFirstSearch::Planner
VelCmd
martJoystickServer::JoystickTask
CDL
rtJoystickNavigation::JoystickNavigationThr
Laser
Planner
JoystickServer
JoystickNav
laser
PerformNavigationScenario::Laser
LaserScanServer
BaseStateClient
lpath
c8
joyNav
PerformNavigationScenario::JoystickNav
JoystickClient
NavigationVelocityServer
jpath
joySrv
PerformNavigationScenario::JoystickServer
JoystickServer
jsource
c1
c3
odom
PerformNavigationScenario::Odom
robotTask_in
BasePositionServer
BatteryEventServer
osource
mapper
PerformNavigationScenario::Mapper
LaserClient
CurrPushServer
mpath
c7
c9
cdl
PerformNavigationScenario::CDL
JoystickClient
RobotBlockedEventState
c4
planner
PerformNavigationScenario::Planner
CurrMapClient
PlannerGoalServer
c2
LaserClient
VelocityServer
jpath
lpath
PlannerGoalClient
GoalEventServer
ppath
c5
c6
velCmd
PerformNavigationScenario::VelCmd
VelocityClient
robotTask_out
vsink
mpath
BaseStateClient
PlannerEventServer
opath

Figure 8.5.: Navigation-scenario extending tasks (top) and defining data-flows (bottom) in the system-performance view (modeled with the AADL OSATE2 tool)

From the data-flow model in Figure 8.5 (bottom), an AADL flow-latency analysis can be triggered that is implemented as a plugin within the OSATE2[1] tool. OSATE2 is the open-source, reference implementation of the *AADL* modeling tools and includes many auxiliary plugins for analyzing the models and generating certified code. Compared with the SymTA/S & Trace Analyser, similar analysis features can be found in the flow-latency analysis plugin of OSATE2 coupled with a scheduling analysis (e.g. the CHEDDAR analysis over the OCARINA interface).

To sum up, this conceptual investigation shows that the proposed abstraction of the *performance* view is generic and flexible enough (yet at the same time detailed enough) to be mapped to approaches such as SymTA/S and to be transformed into different representations such as AADL. The open-source implementation of the flow-latency analysis within the OSATE2 tool showed some instabilities (in the used version 2.2.1), which might be fixed in future versions. As an alternative, a different tool for AADL can be used such as e.g. Ellidiss STOOD[2] or others. Yet another analysis tool (beyond AADL) is MAST [MAS], which is also potentially interesting for future investigations.

8.3. Toward a Combination of the Request-Response - and the Data-Flow Communication Semantics

One specific issue that has not (yet) been addressed in this dissertation is related to the *Query* (aka request–response) communication semantics. Although the publish–subscribe communication semantics dominate various communication middlewares and component models due to its rather simple and easily analyzable semantics, there are still many robotic use-cases where it makes more sense and is more efficient to request data values on demand rather than receiving any available update and filtering out what is not needed. An interesting research question for future works might be "whether request–response can actually be analyzed in chains of components in a similar way much like the data-flow communication as presented in this dissertation?" This section analyzes some of the involved challenges, proposes potential solutions, which are not yet consolidated (but give some early hints), and discusses open problems.

Figure 8.6 illustrates a schematic example with two components on the left that use a *QueryClient* and one component on the right that provides a *QueryServer*. A common implementation of a *QueryServer* is to use a handler that processes all the incoming query requests. In order not to affect the connected clients, this handler can be made active, meaning that a *QueryServer* pushes all incoming query requests into a queue, and an active task processes each query request individually (in any imaginable order such as FIFO, prioritized, earliest deadline, etc.) and replies with an answer to the client. Hence, the initiators and consequently the triggers of that handler task are all the connected clients. If there were only one *QueryClient*, one could say that the query time on the client side would be the communication time for the request and the response plus the worst-case execution time of the query handler. However, as there might be an arbitrary

[1]OSATE2: `http://osate.org/`
[2]STOOD: `www.ellidiss.fr/public/wiki/wiki/stood`

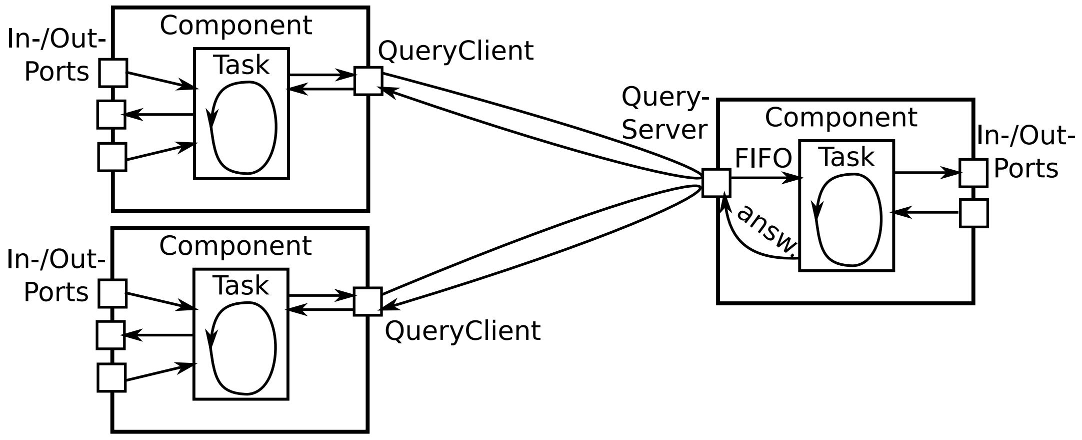


Figure 8.6.: Query communication - schematic example

number of query clients, with each triggering queries on demand, the query time for individual clients is not easy to determine.

One could start making further assumptions about the *QueryServer* so that there is a typical worst-case execution time to answer a query request independent of the other requests or any other factors, and that the query queue is strictly FIFO. Moreover, another assumption could be that at some point all the connected clients are known with their individual update frequencies (at least their worst cases). This might allow a reasonable worst-case execution analysis; however, the question is how pessimistic this analysis would be and thus how helpful in the overall system design it might be.

Regardless of whether the *Query* pattern should be used for time-critical parts (or not), this pattern can be used for the regular uncritical parts that are not related to the end-to-end data-flow chains.

9 Summary and Conclusions

The approach presented in this dissertation addresses two common engineering needs in robotic software development. The first engineering need is related to complexity management by means of CBSE and MDSE. The second is related to managing selected non-functional system aspects by using external analysis tools, which are seamlessly integrated into the overall robotic development process.

This rest of the chapter is structured as follows. The next section sums up the contributions of this dissertation. Section 9.2 discusses the results by giving answers to the research questions stated in the introduction. Section 9.3 draws some conclusions from a bird's-eye perspective and highlights the achieved step change in the field of robotic software development.

9.1. Summary of the Core Contributions

This section summarizes the dissertation's contents and contributions. Consequently, individual core chapters are summed up and the related contributions are highlighted.

After a general introduction in Chapter 1, Chapter 2 presents the fundamentals of this dissertation, which are related to the tools, structures, and implementations around the overall SmartSoft idea. SmartSoft is an umbrella term that nowadays stands for an overall idea and methodology for building robotic software systems. This methodology has been refined and extended in this dissertation to also seamlessly support the management of selected non-functional system aspects as part of an overall, systematic robotic development workflow.

The next Chapter 3—as the first core chapter—introduces the overall methodology of this dissertation. First, the overall problem domain of building complex robotic systems is analyzed and some general guiding rules are derived. In conclusion, it is clear that non-functional system aspects need to be considered from the very beginning of a robotic development process, and that they affect an overall software system at different levels. After that, a common robotic use case is presented, which helps to categorize available and related approaches while at the same time allowing the identification of white spots for investigation in this dissertation. For instance, while there are some common robotic development approaches that use component models and modeling tools, these tools and approaches generally lack structures and abstractions to express and manage non-functional aspects such as end-to-end latencies and jitters in chains of components. Moreover, the relation between component- and task-level composition is discussed. Thereafter, this chapter concludes by presenting the central role of MDSE in this dissertation as the main methodology used throughout the next three core chapters. Overall, Chapter 3 addresses the Contributions 1.1–1.3 at an abstract level.

Chapter 4 focuses on the component-development view as a distinct development phase where individual building blocks are developed without presuming too many application-specific system aspects, which would otherwise hinder the reusability of these building blocks in different systems. Consequently, a consistent meta-model is designed that exposes vital aspects, structures, and abstractions of a component at the model level in such a way that enough details are provided to fully implement the component's functionality, while, at the same time, enough configuration flexibility remains open, which is later used in the system-integration phase to tailor that component for a certain system. Moreover, a flexibly configurable task-trigger design at the framework level is presented, which allows flexibly configuring performance-related component attributes even after the component has been entirely developed and implemented. Overall, Chapter 4 realizes Contribution 1.1.

Chapter 5 focuses on the system-integration phase where entire systems are composed of reusable building-blocks. The system-integration phase in itself comprises (at least) three distinct sub-views for which individual meta-models are designed as part of this chapter. The first two sub-views are related to selecting, wiring and configuring components, as well as providing platform-specific details for deployment. While these two views are rather generic and can be found in a similar form in other system-modeling solutions, they are required as a foundation for the novel modeling view called *performance view*, which is considered as the core contribution of this chapter and of this dissertation as a whole. The *performance view* allows systematic design and management of performance-related system aspects that directly impact end-to-end guarantees in sensor-to-actuator control loops. The required structures, abstractions, and semantics are formalized by a meta-model and model-checks, which, on the one hand, consider configuration constraints from the preceding component meta-model, and on the other hand, are detailed enough to trigger a Compositional Performance Analysis (CPA) (which is discussed in the follow-up Chapter 6). Overall, Chapter 5 realizes Contribution 1.2.

Chapter 6 switches the focus from aspects of design time to modeling and simulating dynamic runtime conditions of a system. This includes, among other things, the analysis of scheduling and sampling effects. Consequently, the above defined *performance view* is transformed—via a model-to-model transformation step—into a representation that can be used to trigger a CPA within the external SymTA/S & Trace Analyser tool. By so doing, the SymTA/S & Trace Analyser tool is integrated into the overall development workflow, thereby using the performance models as input. As a second contribution of this chapter, generic logging and monitoring solutions are presented that allow a direct measurement of end-to-end timings in a real system. This is useful to get ground truth values for comparing the results from the CPA with real measurements (as is done in the next Chapter 7). Overall, Chapter 6 realizes Contribution 1.4.

While the preceding core chapters mainly focus on conceptualization and meta-model design, Chapter 7 shifts the focus to the application of the developed modeling tools in a real-world system example. Consequently, the navigation scenario is modeled using the developed modeling tools, and the relevant models are discussed in the first part of Chapter 7. After that, a Compositional Performance Analysis (CPA) is conducted for the presented navigation scenario. Additionally, the results of the analysis are compared with ground-truth measurements recorded using a real robot that has been executed in a realistic environment. The results show that the chosen abstraction level of the *performance view* is suited for robotic needs and can be linked to state-of-the-art approaches such as SymTA/S. Overall, Chapter 7 realizes Contribution 1.5.

Finally, Chapter 8 briefly discusses some borderline topics mentioned in this dissertation that are interesting and detailed enough to be discussed as promising future works. For instance, the mapping of the *performance view* to AADL is conceptually demonstrated, which underlines the flexibility and generality of the specified *performance view*.

9.2. Discussion of the Achieved Results

The aim of this section is to reflect the overall achievements of this dissertation by giving a direct answer to the initial research questions mentioned at the outset.

> Research Question 1.1: *What exactly are the established means of composition that need to be implemented in a robotic software-development workflow? What are the important steps in such a development workflow?*

Regarding the established means of composition, Section 3.2.2 identifies CBSE and SOA as the main general paradigms that allow finding coarse-grained structures and basic abstractions. However, these paradigms alone do not yet give enough guidance about how to exactly structure the overall system. Instead, as further discussed in Section 2.2.1 and in Section 4.1.1, it then depends on a careful definition of the component's services (using well-defined Communication Patterns), which, depending on their design, either enables or disables composition. Moreover, composition also depends on a systematic development workflow where the involved developer roles and their concerns are thoroughly addressed in dedicated modeling views (see next).

As for the robotic development workflow, Section 3.2.3 identifies at least three main phases, namely: *component development*, *system-integration*, and *runtime*. These three phases have been sufficient for the scope of this dissertation. However, there might be additional phases, most notably an initial design phase even before the *component development*, where the general data structures for Communication Objects are designed (see [Sta+16] for more details). While this phase falls outside the scope of this dissertation, it is conceptually straightforward to attach to this phase without breaking the presented structures. Moreover, the discussed phases themselves can be further sub-divided into additional sub-steps. For example, the *system-integration* phase is subdivided into *system-configuration*, *deployment*, and *performance* sub-steps. Additional sub-steps are also possible, such as e.g. *behavior modeling* (see again [Sta+16] for more details) as part of the overall *system-integration* phase. An important point is that these phases and steps are interlinked at the model level, which allows preservation of overall system consistency throughout the entire development of a robotic software system. In addition, interlinking the phases helps to reduce overlapping concerns between the different modeling views, which effectively separate the responsibilities of the involved developer roles.

> Research Question 1.2: *How can collaborative and stepwise design of data-flow chains be enabled without conflicting with (or breaking) required structures and abstractions of that development workflow?*

This has been the most effective guiding question for the meta-model design in the two core chapters 4 and 5. In brief, the proposed component meta-model is detailed enough in such a way that a component can be entirely designed and implemented without prematurely assuming any application-related system details. Additionally, the component meta-model supports the specification of functional configuration boundaries so that, in a later system-integration phase, a component can be tailored to system needs at a model level without conflicting with the component's internal implementation. All the system-level meta-models ensure consistency by design, which is achieved through model-checks and code completions that filter out all the invalid (i.e., inconsistent) modeling options.

> Research Question 1.3: *What are common patterns of typical data-flow chains and how can these patterns be formalized?*

This question has been addressed by analyzing a real-world use case in Section 3.2. Consequently, Section 4.1.2 derives two common patterns of computation within a component that can be configured in a later *performance* modeling view presented in Section 5.3. This novel *performance* view allows configuration of individual component-tasks so that overall end-to-end guarantees can be met. Whether these guarantees are actually met in a real system is validated using the SymTA/S & Trace Analyser tool, which has been linked with the robotic modeling views as described in Section 6.1. The formalization of the configuration patterns in the *performance* view is done using MDSE methods as introduced in Section 3.3.

> Research Question 1.4: *How can such a workflow be effectively supported with integrated tooling, thereby supporting and guiding different developer roles in the development of realistic, real-world scenarios with real robots?*

This dissertation relies on MDSE methods and tools (see Sections 3.3.1 and 3.3.2) to develop an IDE that combines several interlinked modeling views. The individual modeling views ensure overall system consistency by design, which is achieved through interlinking the individual views at a meta-model level. Thus, individual developer roles of an overall robotic development workflow are guided and supported in their related development phases. Moreover, the modeling views have been entirely implemented in an Eclipse-based environment (see also the attached

Xtext grammars in Appendix A) and a real-world scenario has been developed in Chapter 7 using the new modeling tools (from above). As further argued in Section 7.3, the selected navigation scenario provides enough variability, giving the confidence that the modeling tools are sufficient and rich enough to systematically manage performance-related aspects in a robotic development process.

9.3. Conclusions

In conclusion, the modeling tools presented in this dissertation enable the management of selected non-functional system properties—related to system performance aspects such as end-to-end guarantees—in a consistent and systematic way as part of an overall component-based, robotic software-development workflow. While some existing component-based approaches allow the development of robotic systems, these approaches do not support the management of this kind of non-functional properties. Other approaches, derived from fields other than robotics, such as AADL from the avionics domain, or SymTA/S from the automotive domain, allow the design and analysis of end-to-end timings. However, these external approaches and tools are not common yet, nor are they easily accessible to the domain of robotics. This dissertation presents systematic methods for a consistent integration and usage of such tools and approaches as part of a structured robotic development workflow.

On the whole, this dissertation facilitates the challenging transition from hand-crafted lab prototypes to qualitative, industry-strength robotic products by lifting (at least some of the fundamentally important) QoS-related system aspects to the model level, thereby making them manageable. Of course, many additional QoS aspects might be relevant as well. However, this dissertation makes the first step in this direction with a strong focus on working engineering models instead of scientific ones. In this sense, it can be considered as a practice-oriented work where the developed solutions are evaluated against real-world problems and real-world systems.

Since all the developed modeling tools are published as open-source, they can be easily refined, extended, and used in future projects, which together would hopefully make life easier for roboticists (such as myself) in the long run.

A Appendix

This appendix lists all relevant Xtext grammar specifications and some Matlab plots of the measured performance results.

A.1. Communication Objects Xtext Grammar

Listing A.1 below presents the `Communication Objects` Xtext grammar.

```
grammar org.xtext.commObj.CommObj with org.eclipse.xtext.common.Terminals

import "http://www.eclipse.org/emf/2002/Ecore" as ecore

generate commObj "http://www.xtext.org/commObj/CommObj"

CommObjModel:
   (imports += ImportUri)*
   (elements += PackageDeclaration)*
;

CommObjDocumentation:
  '@doc' text=STRING
;

ImportUri:
   (doc=CommObjDocumentation)?
  'ImportUri' importURI=STRING
;

QN:
  ID ('.' ID)*
;

PackageDeclaration:
   (doc=CommObjDocumentation)?
  'CommObjectRepository' name = ID 'Version' version=Version '{'
```

```
    ('Dependency' dep=STRING)?
    (packagedElements += PkgableElement)*
  '}'
;

Version:
  major=INT '.' minor=INT '.' patch=INT
;

PkgableElement:
  CommunicationObject | Struct | Enumeration
;

CommunicationObject:
  (doc=CommObjDocumentation)?
  'CommObject' name = ID '{'
    (elements += Element)+
  '}'
;

SignedIntValue returns ecore::EInt:
  ('-'|'+')? INT
;
RealValue returns ecore::EDouble:
  ('-'|'+')? INT? '.' INT (('E'|'e') ('+'|'-')? INT)?
;

Element:
  (doc=CommObjDocumentation)?
  name = ID ':' type=AbstractType
;

Cardinality:
  '*' | INT
;

enum INTENUM :
  INT8="Int8" | INT16="Int16" | INT32="Int32" | INT64="Int64"
;
enum UINTENUM :
  // UInt8 is an extra type named OctetType
  UINT16="UInt16" | UINT32="UInt32" | UINT64="UInt64"
;
enum REALENUM :
  FLOAT="Float" | DOUBLE="Double"
;

AbstractType:
  PrimitiveType | EnumerationRef | CommObjRef | StructRef
;
```

```
PrimitiveType:
  OctetType | SignedDecimalType | UnsignedDecimalType | RealType | StringType
    | BooleanType
;

SignedDecimalType:
  name=INTENUM (many?='[' cardinality=Cardinality ']')? (defval ?= '=' value=
    SignedIntValue)?
;
OctetType:
  name='UInt8' (many?='[' cardinality=Cardinality ']')? (defval ?= '=' value=
    SignedIntValue)?
;
UnsignedDecimalType:
  name=UINTENUM (many?='[' cardinality=Cardinality ']')? (defval ?= '=' value=
    SignedIntValue)?
;
RealType:
  name=REALENUM (many?='[' cardinality=Cardinality ']')? (defval ?= '=' value=
    RealValue)?
;
StringType:
  name='String' (many?='[' cardinality=Cardinality ']')? (defval ?= '=' value=
    STRING)?
;
BooleanType:
  name='Boolean' (many?='[' cardinality=Cardinality ']')? (defval ?= '=' value
    =('true'|'false'))?
;

CommObjRef :
  'CommObjectRef' '(' ref = [CommunicationObject|QN] ')' (many?='['
    cardinality=Cardinality ']')?
;

StructRef :
  'StructRef' '(' ref = [Struct|QN] ')' (many?='[' cardinality=Cardinality ']'
    )?
;

EnumerationRef :
  'EnumRef' '(' ref = [Enumeration|QN] ')' (many?='[' cardinality=Cardinality
    ']')? (defval ?= '=' value=[EnumElement])?
;

Struct:
  (doc=CommObjDocumentation)?
  'Struct' name = ID '{'
    (elements += Element)+
  '}'
;

Enumeration:
  (doc=CommObjDocumentation)?
  'Enum' name = ID '{'
    (elements += EnumElement)*
  '}'
;
EnumElement:
  (doc=CommObjDocumentation)?
  name=ID
;
```

Listing A.1: Communication Objects Xtext grammar

A.2. Component Xtext Grammar

Listing A.2 below presents the `Component` Xtext grammar.

```
grammar org.xtext.component.CompDef with org.eclipse.xtext.common.Terminals

import "http://www.eclipse.org/emf/2002/Ecore" as ecore
import "http://www.ecore.org/component" as comp
import "http://www.ecore.org/performExtension" as perf
import "http://www.xtext.org/commObj/CommObj" as commObj

CompDefModel returns comp::ComponentModel:
  {comp::ComponentModel}
  (component=Component)?
;


Component returns comp::Component:
  {comp::Component}
  'Component' name=EString (hasParameters?='HasParameters')?
  '{'
    (inPorts+=InPort)*
    (tasks+=Task)*
    (outPorts+=OutPort)*
  '}';

QID:
  ID ('.' ID)*
;

EString returns ecore::EString:
  STRING | ID;
```

```
EBoolean returns ecore::EBoolean:
  'true' | 'false';

EDouble returns ecore::EDouble:
  '-'? INT? '.' INT (('E'|'e') '-'? INT)?;


InPort returns comp::InPort:
  'InPort' name=EString 'dataType' dataType=[commObj::CommunicationObject|QID
    ];

OutPort returns comp::OutPort:
  'OutPort' name=EString 'dataType' dataType=[commObj::CommunicationObject|QID
    ] 'dataProviderTask' dataProviderTask=[comp::Task|QID];

Task returns comp::Task:
  PreemptiveTask | CooperativeTask
;

PreemptiveTask returns perf::PreemptiveTask:
  'PreemptiveTask' name=EString
  '{'
     (inputLinks+=InputLink)*
     (taskExtensions+=TaskExtension)*
  '}'
;

CooperativeTask returns perf::CooperativeTask:
  'CooperativeTask' name=EString
  '{'
     (inputLinks+=InputLink)*
     (taskExtensions+=TaskExtension)*
  '}'
;

InputLink returns comp::InputLink:
  InputLinkExtension
;


InputLinkExtension returns perf::InputLinkExtension:
  {perf::InputLinkExtension}
  'InputLinkExtension' inRef=[comp::InPort|QID]
  '{'
     ((optional?='optional')? &
     (oversamplingOk?='oversamplingOk')? &
     (undersamplingOk?='undersamplingOk')?)
  '}'
;
```

```
TaskExtension returns comp::TaskExtension:
  ActivationConstraints
;

ActivationConstraints returns perf::ActivationConstraints:
  {perf::ActivationConstraints}
  'ActivationConstraints'
  '{'
    'configurable' configurable=EBoolean
    (('minActFreq' minActFreq=EDouble 'Hz')? &
    ('maxActFreq' maxActFreq=EDouble 'Hz')?)
  '}';
```

Listing A.2: Component Xtext grammar

A.3. System Xtext Grammar

Listing A.3 below presents the `System` Xtext grammar.

```
grammar org.xtext.system.SysConfig with org.eclipse.xtext.common.Terminals

import "http://www.eclipse.org/emf/2002/Ecore" as ecore
import "http://www.ecore.org/component" as comp
import "http://www.ecore.org/system" as sys

SystemModel returns sys::SystemModel:
  {sys::SystemModel}
  'SystemModel'
  name=EString
  '{'
    (elements+=AbstractSysElem)*
  '}';

AbstractSysElem returns sys::AbstractSysElem:
  ComponentInstance | Connection;


QID:
  ID ('.' ID)*
;

EString returns ecore::EString:
  STRING | ID;

ComponentInstance returns sys::ComponentInstance:
  'ComponentInstance' name=EString 'instantiates' compRef=[comp::Component|QID
    ]
  '{'
    (hasRefinedParameters?='HasRefinedParameters')?
    ('inPorts' '(' inPorts+=InPortInstance ( "," inPorts+=InPortInstance)* ')'
     )?
    ('outPorts' '(' outPorts+=OutPortInstance ( "," outPorts+=OutPortInstance)
    * ')' )?
  '}';

Connection returns sys::Connection:
  {sys::Connection}
  'Connection'
  '{'
    (('in' inRef=[sys::InPortInstance|QID])? &
    ('out' outRef=[sys::OutPortInstance|QID])?)
  '}';

InPortInstance returns sys::InPortInstance:
  inRef=[comp::InPort|QID]
;

OutPortInstance returns sys::OutPortInstance:
  outRef=[comp::OutPort|QID]
;
```

Listing A.3: System Xtext grammar

A.4. Deployment Xtext Grammar

Listing A.4 below presents the `Deployment` Xtext grammar.

```
grammar org.xtext.system.deployment.SysDeploy with org.eclipse.xtext.common.
    Terminals

import "http://www.eclipse.org/emf/2002/Ecore" as ecore
import "http://www.ecore.org/deployment" as deploy
import "http://www.ecore.org/system" as system

DeploymentModel returns deploy::DeploymentModel:
  {deploy::DeploymentModel}
  'DeploymentModel' name=EString ('systemModel' systemModel=[system::
    SystemModel|EString])?
  '{'
    (devices+=Device)*
    (networkConns+=NetworkConnection)*
  '}';
```

```
QID:
  ID ('.' ID)*
;

EString returns ecore::EString:
  STRING | ID;

EInt returns ecore::EInt:
  INT;

NetworkConnection returns deploy::NetworkConnection:
  'NetworkConnection' '{'
    device1=[deploy::Device|QID] '<->' device2=[deploy::Device|QID] ('kind'
    kind=EString)?
  '}'
;


Device returns deploy::Device:
  'Device' name=EString
  '{'
     (('IP-Addr' ip=EString ';') &
     ('Login-Name' loginName=EString ';') &
     ('Deployment-Dir' deploymentDir=EString ';') &
     (namingservice=NamingService ';')? &
     (cpu=CPU ';')?)
     ('ComponentArtifacts' '{' artifacts+=ComponentArtifact ( "," artifacts+=
    ComponentArtifact)* '}' )?
  '}';

CPU returns deploy::CPU:
  {deploy::CPU}
  'CPU' name=EString ('kind' kind=EString)?
;


ComponentArtifact returns deploy::ComponentArtifact:
  componentInstance=[system::ComponentInstance|QID]
;

NamingService returns deploy::NamingService:
  'NamingService' 'port' portNr=EInt
;
```

Listing A.4: Deployment Xtext grammar

A.5. Performance Xtext Grammar and Two Completion Proposal Providers

Listing A.5 below presents the `Performance` Xtext grammar.

```
grammar org.xtext.system.performance.SysPerform with org.eclipse.xtext.common.
    Terminals

import "http://www.ecore.org/performance"
import "http://www.eclipse.org/emf/2002/Ecore" as ecore
import "http://www.ecore.org/component" as component
import "http://www.ecore.org/performExtension" as performExtension
import "http://www.ecore.org/deployment" as deployment

PerformanceModel returns PerformanceModel:
  {PerformanceModel}
  'PerformanceModel' name=EString ('deployRef' deployRef=[deployment::
    DeploymentModel|QID])?
  '{'
    (cpuCores+=CPUCore)*
    (tasks+=TaskNode)*
    (dataFlows+=DataFlow)*
    (chains+=TaskChain)*
  '}';


QID:
  ID ('.'ID)*
;
EString returns ecore::EString:
  STRING | ID;

EDouble returns ecore::EDouble:
  '-'? INT? '.' INT (('E'|'e') '-'? INT)?;
EInt returns ecore::EInt:
  '-'? INT;


CPUCore returns CPUCore:
  'CPUCore' name=EString
  '{'
    'number' number=EInt
    ('CPU' cpu=[deployment::CPU|QID])?
  '}';

TaskNode returns TaskNode:
  {TaskNode}
  'TaskNode' name=EString (taskRealization=TaskRealization)?
  '{'
    ('inputs' '{' inputs+=InputNode ( "," inputs+=InputNode)* '}' )?
    (('affinity' affinity=[CPUCore|QID])? &
    ('activation' activation=ActivationSource)? &
    ('executionTime' executionTime=ExecutionTime)? &
    ('scheduler' scheduler=Scheduler)?)
  '}';

InputNode returns InputNode:
  RegisterInputNode | TriggerInputNode;

ActivationSource returns ActivationSource:
  DataTriggered | PeriodicTimer | Sporadic;

TaskChain returns TaskChain:
  'TaskChain' name=EString
  '{'
    'nodes' '{' nodes+=FirstNodeRef (nodes+=NodeRef)* '}'
    ('end2endSpecs' specs=End2EndSpecs)?
  '}';

DataFlow returns DataFlow:
  'DataFlow'
  '{'
    'source' source=[TaskNode|QID]
    'destination' destination=[InputNode|QID]
  '}';

FirstNodeRef returns NodeRef:
  ref=[TaskNode|QID]
;

NodeRef returns NodeRef:
  '->' ref=[TaskNode|QID]
;

End2EndSpecs returns End2EndSpecs:
  {End2EndSpecs}
  '{'
    (('MaxAge' maxAge=TimeValue)? &
    ('Reaction' Reaction=TimeValue)?)
  '}';

TimeValue returns TimeValue:
  value=EInt unit=TimeUnit
;

enum TimeUnit returns TimeUnit:
        SEC = 'sec' | MSEC = 'ms' | USEC = 'us';

ExecutionTime returns ExecutionTime:
  '{'
```

```
    (('minTime' minTime=TimeValue) &
    ('maxTime' maxTime=TimeValue))
  '}';

TaskRealization returns TaskRealization:
  'realizes' compArtifact=[deployment::ComponentArtifact] '.' task=[component
    ::Task]
;

enum SchedulerType returns SchedulerType:
  DEFAULT = 'DEFAULT' | FIFO = 'FIFO' | RR = 'RR';

Scheduler returns Scheduler:
  {Scheduler}
  '{'
    (('type' type=SchedulerType)? &
    ('priority' priority=EInt)?)
  '}';

RegisterInputNode returns RegisterInputNode:
  {RegisterInputNode}
  'RegisterInputNode' name=EString
  '{'
    ('inputLink' inputLink=[performExtension::InputLinkExtension|QID])?
  '}';

TriggerInputNode returns TriggerInputNode:
  'TriggerInputNode' name=EString
  '{'
    ('inputLink' inputLink=[performExtension::InputLinkExtension|QID])?
  '}';

DataTriggered returns DataTriggered:
  'DataTriggered' triggerRef=[TriggerInputNode|QID]
  '{'
    'prescale' prescale=EInt
  '}';

PeriodicTimer returns PeriodicTimer:
  'PeriodicTimer'
  '{'
    'periodicActFreq' periodicActFreq=EDouble 'Hz'
  '}';

Sporadic returns Sporadic:
  {Sporadic}
  'Sporadic'
  '{'
    (('minActFreq' minActFreq=EDouble 'Hz')? &
    ('maxActFreq' maxActFreq=EDouble 'Hz')?)
  '}';
```

Listing A.5: Performance Xtext grammar

Listings A.6 and A.7 present the implementation of two completion proposal providers, one for generating *TaskNode* elements with according *InputNodes* based on *ComponentArtefact* elements from the referenced *Deployment* model and the other for providing successive *NodeRef* elements in a task-chain.

```
override complete_TaskNode(EObject model, RuleCall ruleCall,
    ContentAssistContext context, ICompletionProposalAcceptor acceptor) {
  super.complete_TaskNode(model, ruleCall, context, acceptor)
  if(model instanceof PerformanceModel) {
    val perfomanceModel = (model as PerformanceModel)
    if(perfomanceModel.deployRef != null) {
      for(dev: perfomanceModel.deployRef.devices) {
        for(compArt: dev.artifacts) {
          for(task: compArt.componentInstance.compRef.tasks) {
            // check whether a TaskNode already is available
            if(!perfomanceModel.tasks.exists[it.taskRealization?.task==task])
            {
              // if not, create an according proposal provider
              var proposal = "TaskNode " + task.name + " realizes "
                + compArt.name + "." + task.name + " {\n";
              if(task.inputLinks.size > 0) {
                proposal = proposal + "\t\tinputs {\n"
                for(inLink: task.inputLinks) {
                  if(inLink!=task.inputLinks.head) {
                    proposal = proposal + ",\n"
                  }
                  if(inLink.optional) {
                    proposal = proposal + "\t\t\tRegisterInputNode "
                      + inLink.name + " { ";
                  } else {
                    proposal = proposal + "\t\t\tTriggerInputNode "
                      + inLink.name + " { prescale 1 ";
                  }
                  proposal = proposal + "inputLink " + inLink.name + " }";
                } // end for(inLink: task.inputLinks)
                proposal = proposal + "\n\t\t}";
              } // end if(task.inputLinks.size > 0)
              proposal = proposal + "\n\t}";
              // create completion proposal
              var proposalName = "generate TaskNode "+task.name+" body";
              val completionProposal =
                createCompletionProposal(proposal, proposalName,
                  imageHelper.getImage("smartTask.gif"), context
                );
              acceptor.accept(completionProposal);
            } // end if(!TaskNode already is available)
          } // end for(task: compArt.componentInstance.compRef.tasks)
        } // end for(compArt: dev.artifacts)
      } // end for(dev: perfomanceModel.deployRef.devices)
    } // end if(perfomanceModel.deployRef != null)
  } // end if(model instanceof PerformanceModel)
}
```

Listing A.6: TaskNode completion proposal provider

```
override completeNodeRef_Ref(EObject model, Assignment assignment,
    ContentAssistContext context, ICompletionProposalAcceptor acceptor) {
  //super.completeNodeRef_Ref(model, assignment, context, acceptor)
  if(model instanceof NodeRef) {
    val currNode = (model as NodeRef)
    val chain = (model.eContainer as TaskChain)
    val index = chain.nodes.indexOf(currNode)
    val prevNode = chain.nodes.get(index-1)
    if(prevNode != null) {
      val performanceModel = (chain.eContainer as PerformanceModel)
      val flows = performanceModel.dataFlows.filter[it.source==prevNode.ref]
      for(flow: flows) {
        val destinationTaskNode = (flow.destination.eContainer as TaskNode)
        acceptor.accept(
          createCompletionProposal(destinationTaskNode.name,
            destinationTaskNode.name,
            imageHelper.getImage("smartTask.gif"), context
          )
        );
      }
    }
  }
}
```

Listing A.7: NodeRef completion proposal provider

A.6. SymtaBase Xtext Grammar

Listing A.8 below presents the SymtaBase Xtext grammar.

```
grammar org.xtext.symtaBase.SymtaBase with org.eclipse.xtext.common.Terminals

import "http://www.ecore.org/symtaBase"
import "http://www.eclipse.org/emf/2002/Ecore" as ecore

SVWorkbench returns SVWorkbench:
  {SVWorkbench}
  'SVWorkbench' name=EString
  '{'
```

```
    ('system' system=SymtaSystemDef)?
  '}';

QID:
  ID ('.' ID)*
;

SymtaElementDef returns SymtaElementDef:
  CoreDef | PathDef | TaskDef | TriggerDef;


EString returns ecore::EString:
  STRING | ID;

SymtaSystemDef returns SymtaSystemDef:
  {SymtaSystemDef}
  'SymtaSystemDef' name=EString
  '{'
    'elements' '{' (elements+=SymtaElementDef)* '}'
  '}';

enum AnalysisResult returns AnalysisResult:
  ALL='All' | CUSTOM='Custom'
;


CoreDef returns CoreDef:
  {CoreDef}
  'CoreDef' name=EString
  '{'
    'analysisResult' analysisResult=AnalysisResult
    ('parentOf' '(' parentOf+=[TaskDef|EString] ( "," parentOf+=[TaskDef|
    EString])* ')' )?
  '}';

PathDef returns PathDef:
  'PathDef' name=EString
  '{'
    'semantics' semantics=PathSemantics
    ('parentOf' '(' parentOf+=[TaskDef|EString] ( "," parentOf+=[TaskDef|
    EString])* ')' )?
  '}';

TaskDef returns TaskDef:
  'TaskDef' name=EString ('trigger' parentOf=[TriggerDef|QID])?
  '{'
    (('minExecTime' minExecTimeMS=EDouble 'ms' ';') &
    ('maxExecTime' maxExecTimeMS=EDouble 'ms' ';') &
    ('activationType' activationType=ActivationType ';')? &
    ('activationTime' activationTimeMS=EDouble 'ms' ';')? &
    ('taskType' taskType=TaskType ';') &
    ('priority' priority=EInt ';') &
    ('analyse' analyse=EBoolean ';')?)
  '}';

TriggerDef returns TriggerDef:
  'TriggerDef' name=EString 'caller' caller=[TaskDef|QID]
  '{'
    'repetationFactor' repetationFactor=EInt
  '}';

enum PathSemantics returns PathSemantics:
        MAX_AGE = 'MaxAgeSemantic' | REACTION = 'ReactionSemantic';

EDouble returns ecore::EDouble:
  '-'? INT? '.' INT (('E'|'e') '-'? INT)?;

EBoolean returns ecore::EBoolean:
  'true' | 'false';

enum ActivationType returns ActivationType:
        PERIODIC = 'PERIODIC' | SPORADIC = 'SPORADIC' | DATA="DATA" |
    UNDEFINED = 'UNDEFINED';

enum TaskType returns TaskType:
        PREEMPTIVE = 'PREEMPTIVE' | NONPREEMPTIVE = 'NON_PREEMPTIVE' |
    UNDEFINED = 'UNDEFINED';

EInt returns ecore::EInt:
  '-'? INT;
```

Listing A.8: SymtaBase Xtext grammar

A.7. SymtaConfig Xtext Grammar

Listing A.9 below presents the `SymtaConfig` Xtext grammar.

```
grammar org.xtext.symtaConfig.SymtaConfig with org.eclipse.xtext.common.
    Terminals

import "http://www.ecore.org/symtaConfig"
import "http://www.ecore.org/symtaBase" as symtaBase
import "http://www.eclipse.org/emf/2002/Ecore" as ecore

SVWorkbenchConfig returns SVWorkbenchConfig:
  'SVWorkbenchConfig' ref=[symtaBase::SVWorkbench|EString]
  '{'
    systemConfig=SymtaSystemConfig
  '}';

QID:
  ID ('.'ID)*
;
SymtaElementConfig returns SymtaElementConfig:
  CoreConfig;


SymtaSystemConfig returns SymtaSystemConfig:
  'SymtaSystemConfig' ref=[symtaBase::SymtaSystemDef|QID]
  '{'
    (('numberOfTraces' numberOfTraces=EInt ';') &
    ('traceTime' traceTimeMS=EDouble 'ms' ';'))
    'elementConfigs' '{' (elementConfigs+=SymtaElementConfig)* '}'
  '}';

EInt returns ecore::EInt:
  '-'? INT;

EDouble returns ecore::EDouble:
  '-'? INT? '.' INT (('E'|'e') '-'? INT)?;

EString returns ecore::EString:
  STRING | ID;


CoreConfig returns CoreConfig:
  'CoreConfig' ref=[symtaBase::CoreDef|QID]
  '{'
    (('scheduler' scheduler=EString ';') &
    ('speedFactor' speedFactor=EInt ';') &
    ('kernelPrio' kernelPrio=EInt ';') &
    ('execBuffer' execBuffer=EInt ';'))
  '}';
```

Listing A.9: SymtaConfig Xtext grammar

A.8. Matlab Histogram Plots for the Execution- and End-to-End Times

Figures A.1 to A.4 show the plotted execution-times of the different tasks. Therefore, the real execution times of different task have been logged in a real robot system that has been executed for half an hour. After that, the logged raw data has been used to plot the presented histogram plots using Matlab.

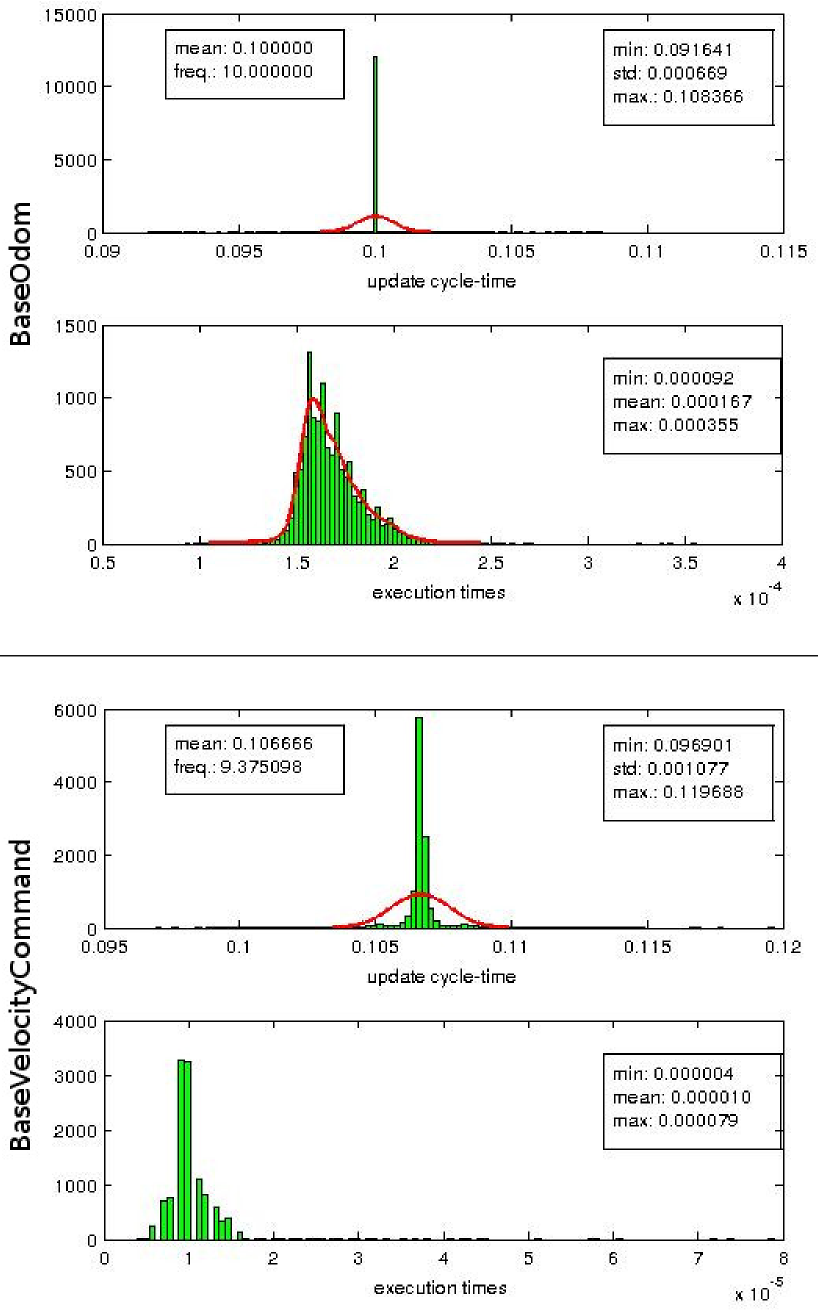


Figure A.1.: Histogram plots of the logged execution-times for the BaseOdometry task (the two plots on top) and the BaseVelocityCommand task (bottom plots)

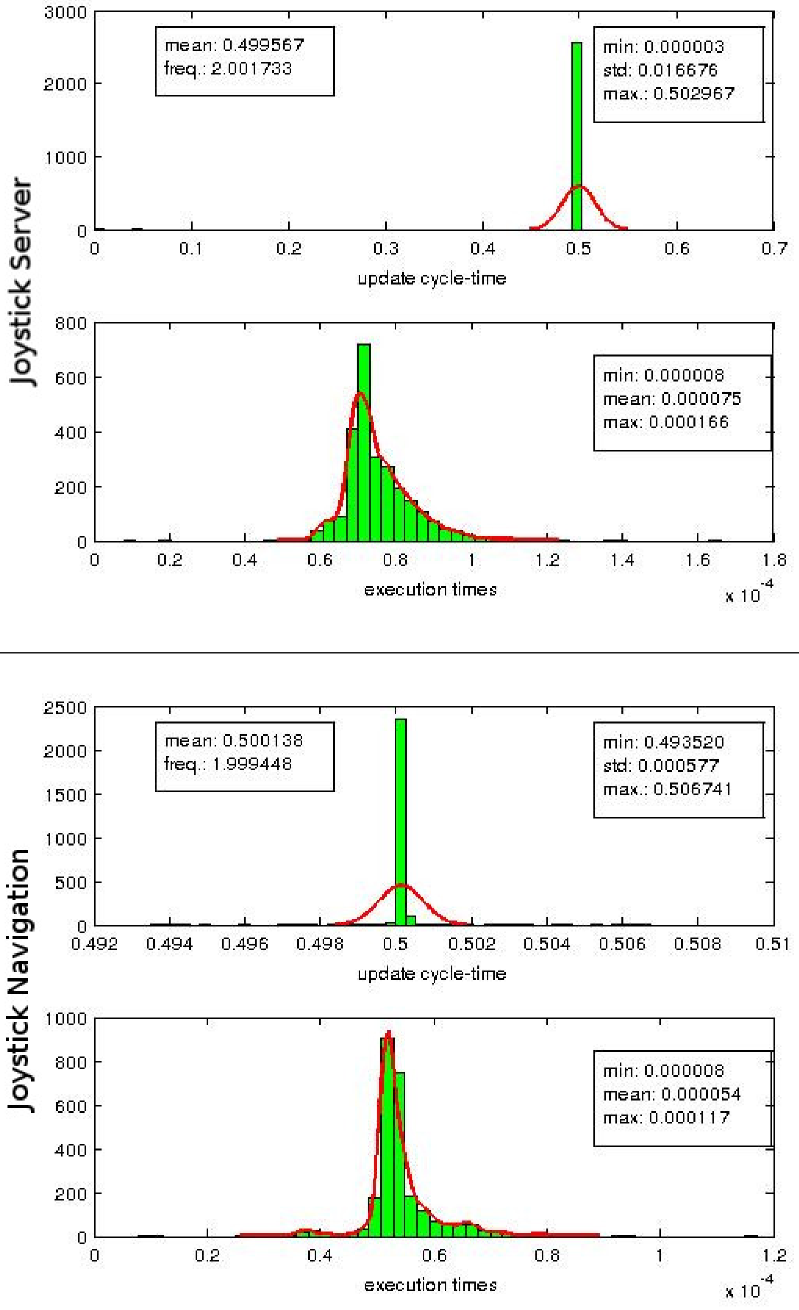


Figure A.2.: Histogram plots of the logged execution-times for the JoystickServer task (the two plots on top) and the JoystickNavigation task (bottom plots)

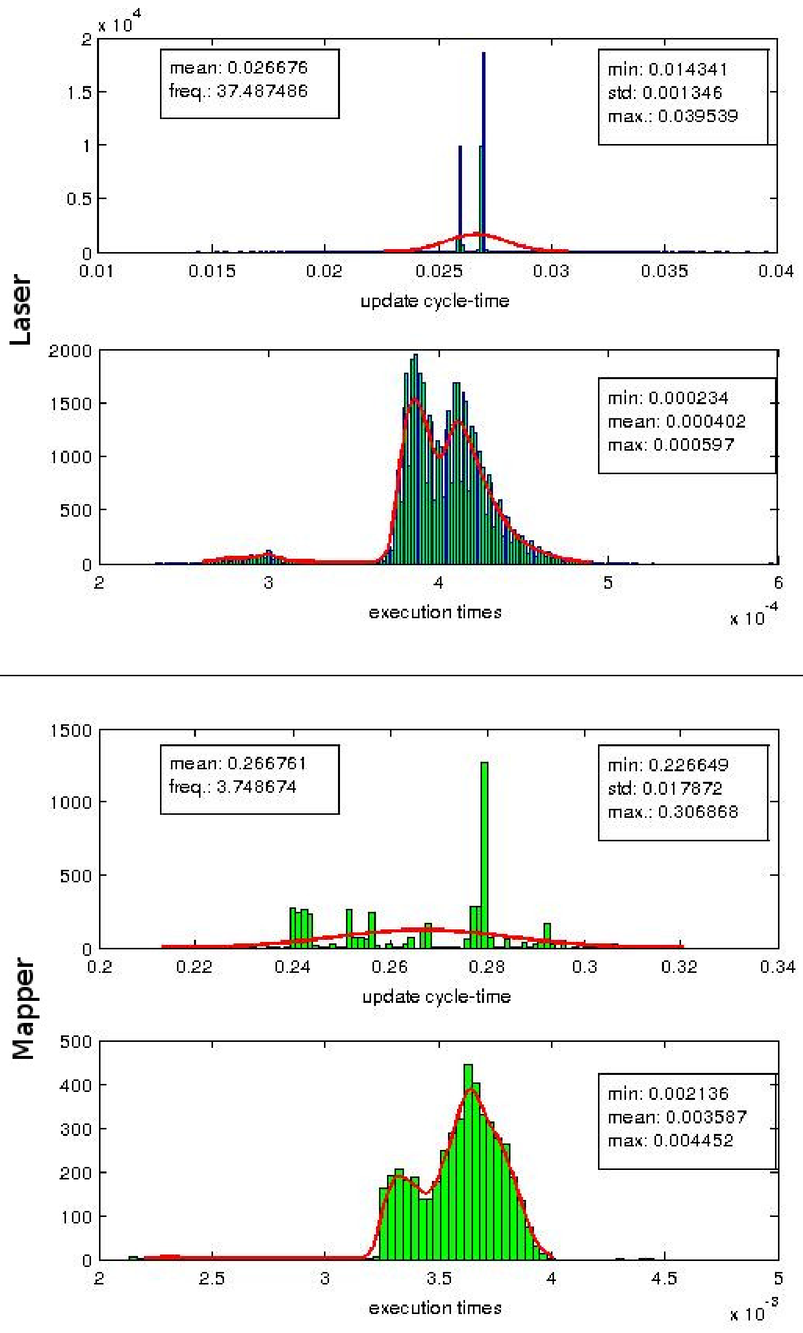


Figure A.3.: Histogram plots of the logged execution-times for the Laser task (the two plots on top) and the Mapper task (bottom plots)

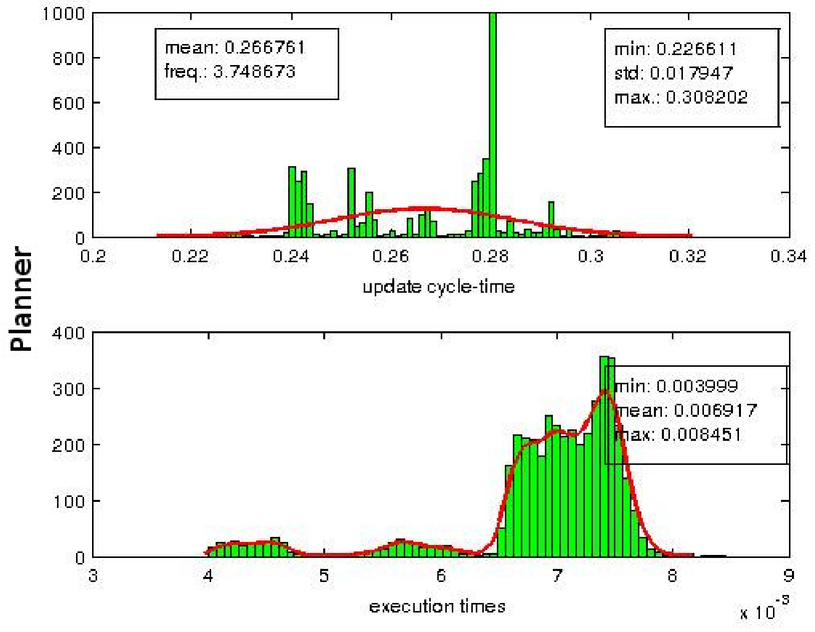


Figure A.4.: Histogram plots of the logged execution-times for the Planner task

Bibliography

[AAD04] AADL. *Architecture Analysis & Design Language (AADL)*. Society of Automotive Engineers (SAE) standard AS5506. 2004.

[AB01] Luca Abeni and Giorgio Buttazzo. “Stochastic analysis of a reservation based system”. In: *Proceedings 15th International Parallel and Distributed Processing Symposium. IPDPS 2001*. 2001, pp. 946–952.
DOI: 10.1109/IPDPS.2001.925049.

[AKS16] Sorin Adam, Marco Kuhrmann, and Ulrik Pagh Schultz. “Towards a virtual machine approach to resilient and safe mobile robots”. In: *IEEE 21st International Conference on Emerging Technologies and Factory Automation (ETFA)*. 2016, pp. 1–8.
DOI: 10.1109/ETFA.2016.7733545.

[And+05] Noriaki Ando, Takashi Suehiro, Kousei Kitagaki, Tetsuo Kotoku, and Woo-Keun Yoon. “RT-Component Object Model in RT-Middleware Distributed Component Middleware for RT (Robot Technology)”. In: *IEEE International Symposium on Computational Intelligence in Robotics and Automation (CIRA ’05)*. 2005, pp. 457 –462.
DOI: 10.1109/CIRA.2005.1554319.

[AC04] C. Angeli and A. Chatzinikolaou. “On-line Fault Detection Techniques for Technical Systems: A survey”. In: *International Journal of Computer Science & Applications* 1.1 (2004), pp. 12–30.

[Aut] *Automotive Open System Architecture*. Online.
URL: www.autosar.org.

[Awa+16] Ramez Awad, Georg Heppner, Arne Rönnau, and Mirko Bordignon. “ROS engineering workbench based on semantically enriched app models for improved reusability”. In: *2016 IEEE 21st International Conference on Emerging Technologies and Factory Automation (ETFA)*. 2016, pp. 1–9.
DOI: 10.1109/ETFA.2016.7733581.

[Bel04] Alex E. Bell. “Death by UML Fever”. In: *Queue* 2.1 (2004), pp. 72–80. ISSN: 1542-7730.
DOI: 10.1145/984458.984495.

[BHA12] Nelly Bencomo, Svein Hallsteinsen, and Eduardo Santana de Almeida. “A View of the Dynamic Software Product Line Landscape”. In: *IEEE Computer Society* 45.10 (2012), pp. 36–41. ISSN: 0018-9162.
DOI: 10.1109/MC.2012.292.

[BFA14] Geoffrey Biggs, Kiyoshi Fujiwara, and Keiju Anada. “Modelling and Analysis of a Redundant Mobile Robot Architecture Using AADL”. In: *Simulation, Modeling, and Programming for Autonomous Robots (SIMPAR 2014)*. Springer International Publishing, 2014, pp. 146–157.

[Boa90] IEEE Standards Board. "IEEE Standard Glossary of Software Engineering Terminology". In: *IEEE Std 610.12-1990* (1990), pp. 1–84.
DOI: 10.1109/IEEESTD.1990.101064.

[Bon+97] R. Peter Bonasso, R. James Firby, Erann Gat, David Kortenkamp, David P. Miller, and Mark G. Slack. "Experiences with an architecture for intelligent, reactive agents". In: *Journal of Experimental & Theoretical Artificial Intelligence* 9.2-3 (1997).
DOI: 10.1080/095281397147103.

[BC10] Jonathan Boren and Steve Cousins. "The SMACH High-Level Executive". In: *IEEE Robotics Automation Magazine* 17.4 (2010), pp. 18 –20. ISSN: 1070-9932.
DOI: 10.1109/MRA.2010.938836.

[Bos00] Jan Bosch. *Design and Use of Software Architectures: Adopting and Evolving a Product-line Approach*. New York, NY, USA: ACM Press/Addison-Wesley Publishing Co., 2000. ISBN: 0-201-67494-7.

[BFG17] Adnan Bouakaz, Pascal Fradet, and Alain Girault. "A Survey of Parametric Dataflow Models of Computation". In: *ACM Transactions on Design Automation of Electronic Systems (TODAES)* 22.2 (2017), 38:1–38:25. ISSN: 1084-4309.
DOI: 10.1145/2999539.

[BS05] Bruno Bouyssounouse and Joseph Sifakis. "Real-Time Scheduling". In: *Embedded Systems Design: The ARTIST Roadmap for Research and Development*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, pp. 242–257. ISBN: 978-3-540-31973-3.
DOI: 10.1007/978-3-540-31973-3_20.

[BCW12] Marco Brambilla, Jordi Cabot, and Manuel Wimmer. *Model-Driven Software Engineering in Practice*. 1st. Morgan & Claypool Publishers, 2012. ISBN: 9781608458820.

[Bro+14] Yury Brodskiy, Robert Wilterdink, Stefano Stramigioli, and Jan Broenink. "Fault Avoidance in Development of Robot Motion-Control Software by Modeling the Computation". In: *Proceedings of the 4th International Conference on Simulation, Modeling, and Programming for Autonomous Robots (SIMPAR 2014), Bergamo, Italy, October 20-23, 2014*. Ed. by Davide Brugali, Jan F. Broenink, Torsten Kroeger, and Bruce A. MacDonald. Springer International Publishing, 2014, pp. 158–169. ISBN: 978-3-319-11900-7.
DOI: 10.1007/978-3-319-11900-7_14.

[Bro86] Rodney A. Brooks. "A robust layered control system for a mobile robot". In: *IEEE Journal of Robotics and Automation* 2.1 (1986), pp. 14–23. ISSN: 0882-4967.
DOI: 10.1109/JRA.1986.1087032.

[BS09] Davide Brugali and Patrizia Scandurra. "Component-Based Robotic Engineering (Part I)". In: *IEEE Robotics and Automation Magazine* 16.4 (2009), pp. 84–96. ISSN: 1070-9932.
DOI: 10.1109/MRA.2009.934837.

[BS10] Davide Brugali and Azamat Shakhimardanov. "Component-Based Robotic Engineering (Part II)". In: *IEEE Robotics Automation Magazine* 17.1 (2010), pp. 100–112. ISSN: 1070-9932.
DOI: 10.1109/MRA.2010.935798.

[Bru01] Herman Bruyninckx. "Open robot control software: the OROCOS project". In: *IEEE International Conference on Robotics and Automation, ICRA 2001*. Vol. 3. 2001, pp. 2523 –2528.

[Bru+13] Herman Bruyninckx, Markus Klotzbücher, Nico Hochgeschwender, Gerhard Kraetzschmar, Luca Gherardi, and Davide Brugali. "The BRICS component model: a model-based development paradigm for complex robotics software systems". In: *Proc. of the 28th Annual ACM Symposium on Applied Computing*. SAC '13. Coimbra, Portugal: ACM, 2013. ISBN: 978-1-4503-1656-9.
DOI: 10.1145/2480362.2480693.

[Bus+96] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stal. *Pattern-oriented Software Architecture: A System of Patterns*. New York, NY, USA: John Wiley & Sons, Inc., 1996. ISBN: 0-471-95869-7.

[Cap+14] Rafael Capilla, Jan Bosch, Pablo Trinidad, Antonio Ruiz-Cortés, and Mike Hinchey. "An overview of Dynamic Software Product Line architectures and techniques: Observations from research and industry". In: *Journal of Systems and Software* 91 (2014), pp. 3 –23. ISSN: 0164-1212.
DOI: 10.1016/j.jss.2013.12.038.

[CCM06] CCM. *CORBA Component Model*. OMG Document formal/06-04-01. Version 4.0. 2006.

[Cle+10] Paul Clements, Felix Bachmann, Len Bass, David Garlan, James Ivers, Reed Little, Paulo Merson, Robert Nord, and Judith Stafford. "Documenting Software Architectures: Views and Beyond". In: Second. Addison-Wesley Professional, 2010. Chap. Prologue: Software Architectures and Documentation. ISBN: 0321552687.

[CN01] Paul C. Clements and Linda M. Northrop. *Software Product Lines: Practices and Patterns*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2001. ISBN: 0-201-70332-7.

[DDS07] DDS. *OMG Data Distribution Service for Real-time Systems*. OMG Document formal/07-01-01. Version 1.2. 2007.

[Der74] M. L. Dertouzos. "Control robotics: the procedural control of physical processes". In: *Proceedings of the IFIP Congress*. 1974, pp. 807–813.

[Dho+12] Saadia Dhouib, Selma Kchir, Serge Stinckwich, Tewfik Ziadi, and Mikal Ziane. “RobotML, a Domain-Specific Language to Design, Simulate and Deploy Robotic Applications”. In: *3rd International Conference on Simulation, Modeling and Programming for Autonomous Robots, SIMPAR 2012*. Ed. by Itsuki Noda, Noriaki Ando, Davide Brugali, and JamesJ. Kuffner. Vol. 7628. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2012, pp. 149–160. ISBN: 978-3-642-34326-1.

[Dij82] Edsger W. Dijkstra. “Selected Writings on Computing: A personal Perspective”. In: Monographs in Computer Science. Springer, 1982. Chap. On the Role of Scientific Thought, pp. 60–66. ISBN: 0-387-90652-5.

[Edw+09] George Edwards, Joshua Garcia, Hossein Tajalli, Daniel Popescu, Nenad Medvidovic, Gaurav Sukhatme, and Brad Petrus. “Architecture-driven self-adaptation and self-management in robotics systems”. In: *International Workshop on Software Engineering for Adaptive and Self-Managing Systems (SEAMS)*. Vancouver, BC, Canada, 2009, pp. 142–151. ISBN: 978-1-4244-3724-5.
DOI: `10.1109/SEAMS.2009.5069083`.

[ES12] Ayssam Elkady and Tarek Sobh. “Robotics Middleware: A Comprehensive Literature Survey and Attribute-Based Bibliography”. In: *Journal of Robotics* (2012), p. 15.
DOI: `doi:10.1155/2012/959013`.

[Erl07] Thomas Erl. *SOA Principles of Service Design*. The Prentice Hall Service-Oriented Computing Series from Thomas Erl. Upper Saddle River, NJ, USA: Prentice Hall PTR, 2007. ISBN: 0132344823.

[Fir89] Robert James Firby. “Adaptive Execution in Complex Dynamic Worlds”. PhD thesis. New Haven, CT, USA: Yale University, 1989.

[Fow11] Martin Fowler. *Domain-Specific Languages*. The Adisson-Wesley Signature Series. Addison-Wesley, 2011. ISBN: 978-0321712943.

[Gat92] Erann Gat. “Integrating Planning and Reacting in a Heterogeneous Asynchronous Architecture for Controlling Real-world Mobile Robots”. In: *Proceedings of the Tenth National Conference on Artificial Intelligence*. AAAI’92. San Jose, California: AAAI Press, 1992, pp. 809–815. ISBN: 0-262-51063-4.

[GVH03] B. Gerkey, R. T. Vaughan, and A. Howard. “The Player/Stage Project: Tools for Multi-Robot and Distributed Sensor Systems”. In: *Proc. of the 11th Int. Conf. on Advanced Robotics (ICAR 2003)*. Coimbra, Portugal, 2003, pp. 317–323.

[GB11] Luca Gherardi and Davide Brugali. “An eclipse-based Feature Models toolchain”. In: *6th Workshop of the Italian Eclipse Community (Eclipse-IT 2011)*. Milano, Italy, 2011.

[GB14] Luca Gherardi and Davide Brugali. “Modeling and reusing robotic software architectures: The HyperFlex toolchain”. In: *IEEE International Conference on Robotics and Automation (ICRA)*. 2014.
DOI: 10.1109/ICRA.2014.6907806.

[Gho+04] Arkadeb Ghosal, Thomas A. Henzinger, Christoph M. Kirsch, and Marco A. A. Sanvido. “Event-Driven Programming with Logical Execution Times”. In: *Proceedings of the 7th International Workshop on Hybrid Systems: Computation and Control (HSCC 2004)*. Philadelphia, PA, USA: Springer, 2004, pp. 357–371. ISBN: 978-3-540-24743-2.
DOI: 10.1007/978-3-540-24743-2_24.

[Gro09] Richard C. Gronback. *Eclipse Modeling Project: A Domain-Specific Language (DSL) Toolkit*. Upper Saddle River, NJ: Addison-Wesley, 2009. ISBN: 978-0-321-53407-1.

[HBK11] Martin Hägele, Nikolaus Blümlein, and Oliver Kleine. *EFFIROB-Studie - Wirtschaftlichkeitsanalysen neuartiger Servicerobotik-Anwendungen und ihre Bedeutung für die Robotik-Entwicklung*. Tech. rep. Fraunhofer IPA und ISI, 2011.
URL: http://www.ipa.fraunhofer.de/studien.

[Han07] Peter Feiler Jörgen Hansson. *Flow Latency Analysis with the Architecture Analysis and Design Language (AADL)*. TECHNICAL NOTE CMU/SEI-2007-TN-010. Carnegie Mellon University, 2007.

[HC01] George T. Heineman and William T. Councill. *Component-Based Software Engineering: Putting the Pieces Together*. Addison-Wesley Professional, 2001. ISBN: 0201704854.

[Hen+05] Rafik Henia, Arne Hamann, Marek Jersak, Razvan Racu, Kai Richter, and Rolf Ernst. “System level performance analysis - the SymTA/S approach”. In: *IEE Proceedings - Computers and Digital Techniques* 152.2 (2005), pp. 148 –166. ISSN: 1350-2387.
DOI: 10.1049/ip-cdt:20045088.

[Hic11] Rich Hickey. “Simple Made Easy”. In: *Strange Loop Conference*. Keynote, 2011.
URL: www.infoq.com/presentations/Simple-Made-Easy.

[HJS03a] Stephen D. Huston, James C. E. Johnson, and Umar Syyid. *The ACE Programmer’s Guide: Practical Design Patterns for Network and Systems Programming*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2003. ISBN: 0201699710.

[HJS03b] Stephen D. Huston, James C. E. Johnson, and Umar Syyid. “The ACE Programmer’s Guide: Practical Design Patterns for Network and Systems Programming”. In: Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2003. Chap. Using the ACE Logging Facility. ISBN: 0201699710.

[IBM06] IBM. *An architectural blueprint for autonomic computing*. White Paper. IBM, 2006.

[IW15] Didac Gil De La Iglesia and Danny Weyns. “MAPE-K Formal Templates to Rigorously Design Behaviors for Self-Adaptive Systems”. In: *ACM Trans. Auton. Adapt. Syst.* 10.3 (2015), 15:1–15:31. ISSN: 1556-4665.
DOI: 10.1145/2724719.
URL: http://doi.acm.org/10.1145/2724719.

[ISK15] Johann Thor Mogensen Ingibergsson, Ulrik Pagh Schultz, and Dirk Kraft. “Towards Declarative Safety Rules for Perception Specification Architectures”. In: *Sixth International Workshop on Domain-Specific Languages and Models for Robotic Systems (DSLRob’15)*. Hamburg (Germany), 2015.
URL: https://arxiv.org/abs/1601.02778.

[IR+12] Juan F. Inglés-Romero, Alex Lotz, Cristina Vicente-Chicote, and Christian Schlegel. “Dealing with Run-Time Variability in Service Robotics: Towards a DSL for Non-Functional Properties”. In: *3rd International Workshop on Domain-Specific Languages and models for ROBotic systems (DSLRob-12)*. Tsukuba (Japan), 2012.

[JA11] Sylvain Joyeux and Jan Albiez. “Robot development: from components to systems”. In: *6th National Conference on Control Architectures of Robots*. Grenoble, France, 2011.

[Kah74] Gilles Kahn. “The semantics of a simple language for parallel programming”. In: *Proceedings of IFIP Congress 74*. Ed. by Jack L. Rosenfeld. 1974.

[KPP95] Barbara Kitchenham, Lesley Pickard, and Shari Lawrence Pfleeger. “Case Studies for Method and Tool Evaluation”. In: *IEEE Software* 12.4 (1995), pp. 52–62. ISSN: 0740-7459.
DOI: 10.1109/52.391832.

[KB12] Markus Klotzbücher and Herman Bruyninckx. “Coordinating Robotic Tasks and Systems with rFSM Statecharts”. In: *Journal of Software Engineering for Robotics (JOSER)*. Vol. 3. 1. 2012, pp. 28–56.

[Kol+10] Steffen Kollman, Victor Pollex, Kilian Kempf, Frank Slomka, Matthias Traub, Torsten Bone, and Jurgen Becker. “Comparative Application of Real-Time Verification Methods to an Automotive Architecture”. In: *18th International Conference on Real-Time and Network Systems*. Toulouse, France, 2010.

[KS08a] David Kortenkamp and Reid Simmons. “Springer Handbook of Robotics”. In: ed. by Bruno Siciliano and Oussama Khatib. Berlin, Heidelberg: Springer, 2008. Chap. Robotic Systems Architectures and Programming. ISBN: 978-3-540-30301-5.
DOI: 10.1007/978-3-540-30301-5_9.

[KS08b] David Kortenkamp and Reid Simmons. “The Art of Robot Architectures”. In: *Springer Handbook of Robotics*. Ed. by Bruno Siciliano and Oussama Khatib. Springer, 2008. Chap. 8.5, pp. 202 –203. ISBN: 978-3-540-23957-4.

[Lee01] E.A. Lee. “Computing for embedded systems”. In: *Proceedings of the IEEE Instrumentation and Measurement Technology Conference* 3 (2001), pp. 1830–1837.

[Lee10] Edward A. Lee. “Disciplined Heterogeneous Modeling”. In: *MODELS 2010*. Invited Keynote Talk. Oslo, Norway, 2010.

[LM87] Edward A. Lee and David G. Messerschmitt. “Synchronous data flow”. In: *Proceedings of the IEEE* 75.9 (1987), pp. 1235–1245. ISSN: 0018-9219. DOI: 10.1109/PROC.1987.13876.

[Leh90] J. P. Lehoczky. “Fixed priority scheduling of periodic task sets with arbitrary deadlines”. In: *Proceedings of IEEE Real-Time Symposium (RTSS)*. 1990, pp. 201–209.

[LL73] C. L. Liu and J. W. Layland. “Scheduling algorithms for multiprogramming in a hard real-time environment”. In: *Journal of the ACM*. Vol. 20(1). 1973, pp. 46–61.

[LSS11] Alex Lotz, Andreas Steck, and Christian Schlegel. “Runtime Monitoring of Robotics Software Components: Increasing Robustness of Service Robotic Systems”. In: *15th International Conference on Advanced Robotics (ICAR)*. Tallinn (Estonia), 2011.

[LSS12] Alex Lotz, Andreas Steck, and Christian Schlegel. “Analysing solution quality of anytime Bag of Words object classification for a service robot”. In: *IEEE International Conference on Technologies for Practical Robot Applications (TePRA 2012)*. 2012, pp. 1–6. DOI: 10.1109/TePRA.2012.6215645.

[Lot+13] Alex Lotz, Juan F. Inglés-Romero, Cristina Vicente-Chicote, and Christian Schlegel. “Managing Run-Time Variability in Robotics Software by Modeling Functional and Non-functional Behavior”. In: *Enterprise, Business-Process and Information Systems Modeling: 14th International Conference, BPMDS 2013, 18th International Conference, EMMSAD 2013, Held at CAiSE 2013, Valencia, Spain, June 17-18, 2013*. Ed. by Selmin Nurcan, Henderik A. Proper, Pnina Soffer, John Krogstie, Rainer Schmidt, Terry Halpin, and Ilia Bider. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 441–455. ISBN: 978-3-642-38484-4. DOI: 10.1007/978-3-642-38484-4_31.

[Lot+14] Alex Lotz, Juan F. Inglés-Romero, Dennis Stampfer, Matthias Lutz, Cristina Vicente-Chicote, and Christian Schlegel. “Towards a Stepwise Variability Management Process for Complex Systems: A Robotics Perspective”. In: *International Journal of Information System Modeling and Design (IJISMD)*. Vol. 5. 3. IGI Global, 2014, pp. 55–74. DOI: 10.4018/ijismd.2014070103.

[Lut+14] Matthias Lutz, Dennis Stampfer, Alex Lotz, and Christian Schlegel. “Service Robot Control Architectures for Flexible and Robust Real-World Task Execution: Best Practices and Patterns”. In: *Informatik 2014, Workshop Roboter-Kontrollarchitekturen*. Stuttgart, Germany, Springer LNI der GI, 2014. ISBN: 978-3-88579-626-8.

[MAR11] MARTE. *A UML Profile for MARTE: Modeling and Analysis of Real-Time Embedded systems*. OMG Document formal/2011-06-02. Version 1.1. 2011.

[MAS] MAST. *Modeling and Analysis Suite for Real-Time Applications*. URL: http://mast.unican.es/.

[MFN06] Giorgio Metta, Paul Fitzpatrick, and Lorenzo Natale. “YARP: Yet Another Robot Platform”. In: *International Journal of Advanced Robotic Systems* 3.1 (2006), p. 8. DOI: 10.5772/5761.

[Met+10] Giorgio Metta et al. “The iCub humanoid robot: An open-systems platform for research in cognitive development”. In: *Neural Networks* 23.8–9 (2010), pp. 1125 – 1134. ISSN: 0893-6080. DOI: dx.doi.org/10.1016/j.neunet.2010.08.010.

[Mur89] T. Murata. “Petri nets: Properties, analysis and applications”. In: *Proceedings of the IEEE* 77.4 (1989), pp. 541–580. ISSN: 0018-9219. DOI: 10.1109/5.24143.

[Nil80] Nils J. Nilsson. *Principles of Artificial Intelligence*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1980. ISBN: 0-934613-10-9.

[Nor+16] Arne Nordmann, Nico Hochgeschwender, Dennis Wigand, and Sebastian Wrede. “A Survey on Domain-specific Modeling and Languages in Robotics”. In: *Special Issue on Domain-Specific Languages and Models in Robotics*. Vol. 7. Journal of Software Engineering for Robotics (JOSER) 1. 2016. URL: http://joser.unibg.it/index.php?journal=joser&page=article&op=view&path%5B%5D=100.

[Cor] *OMG Common Object Request Broker Architecture*. Online. URL: www.omg.org/spec/CORBA/.

[Mof] *OMG’s MetaObject Facility (MOF)*. Online. URL: www.omg.org/mof/.

[Ore+99] Peyman Oreizy, Michael M. Gorlick, Richard N. Taylor, Dennis Heimbigner, Gregory Johnson, Nenad Medvidovic, Alex Quilici, David S. Rosenblum, and Alexander L. Wolf. “An Architecture-Based Approach to Self-Adaptive Software”. In: *IEEE Intelligent Systems* 14.3 (May 1999), pp. 54–62. ISSN: 1541-1672. DOI: 10.1109/5254.769885. URL: http://dx.doi.org/10.1109/5254.769885.

[OSA] OSATE2. *Open Source AADL2 Tool Environment*. URL: http://osate.github.io/.

[Ple12] Johannes Pletzer. “Mapping of Timing Definition Language (TDL) Components to Distributed Platforms”. Dissertation. Department of Computer Sciences at the University of Salzburg, 2012.

[Qui+09] Morgan Quigley, Ken Conley, Brian P. Gerkey, Josh Faust, Tully Foote, Jeremy Leibs, Rob Wheeler, and Andrew Y. Ng. "ROS: an open-source Robot Operating System". In: *ICRA Workshop on Open Source Software* (2009).

[Rob] *RobMoSys: Composable Models and Systems for Robotics Systems-of-Systems*. Online. 2017.
URL: http://robmosys.eu/.

[Mar] *Robotics 2020 Multi-Annual Roadmap*. Version Rev B. Call 2 ICT24 – Horizon 2020. SPARC Robotics – The Partnership for Robotics in Europe, 2015.
URL: http://www.eu-robotics.net/cms/upload/Multi-Annual_Roadmap2020_ICT-24_Rev_B_full.pdf.

[Sch98] Christian Schlegel. "Fast local obstacle avoidance under kinematic and dynamic constraints for a mobile robot". In: *IEEE International Conference on Intelligent Robots and Systems (IROS)*. Victoria, Canada, 1998.
DOI: 10.1109/IROS.1998.724683.

[Sch04] Christian Schlegel. "Navigation and Execution for Mobile Robots in Dynamic Environments: An Integrated Approach". Dissertation. University of Ulm, 2004.

[SLS11a] Christian Schlegel, Alex Lotz, and Andreas Steck. *SmartSoft - The State Management of a Component*. Tech. rep. 01. Germany: University of Applied Sciences Ulm, 2011.

[SSL12a] Christian Schlegel, Andreas Steck, and Alex Lotz. "Model-Driven Software Development in Robotics: Communication Patterns as Key for a Robotics Component Model". In: *Introduction to Modern Robotics*. Ed. by Daisuke Chugo and Sho Yokota. iConcept Press, 2012, pp. 119–150. ISBN: 978-0980733068.

[SSL12b] Christian Schlegel, Andreas Steck, and Alex Lotz. "Robotic Software Systems: From Code-Driven to Model-Driven Software Development". In: *Robotic Systems - Applications, Control and Programming*. Ed. by Ashish Dutta. InTech, 2012, pp. 473–502. ISBN: 978-953-307-941-7.

[SW99] Christian Schlegel and Robert Worz. "The software framework SMARTSOFT for implementing sensorimotor systems". In: *Proceedings of IEEE International Conference on Intelligent Robots and Systems (IROS '99)*. Vol. 3. 1999, pp. 1610–1616.
DOI: 10.1109/IROS.1999.811709.

[Sch+09] Christian Schlegel, Thomas Hassler, Alex Lotz, and Andreas Steck. "Robotic software systems: From code-driven to model-driven designs". In: *International Conference on Advanced Robotics (ICAR)*. Munich, Germany, 2009.

[Sch+13] Christian Schlegel, Alex Lotz, Matthias Lutz, Dennis Stampfer, Juan F. Inglés-Romero, and Cristina Vicente-Chicote. "Model-Driven Software Systems Engineering in Robotics: Covering the Complete Life-Cycle of a Robot". In: *Informatik 2013, Workshop Roboter-Kontrollarchitekturen*. Koblenz, Germany, Springer LNI der GI, 2013.

[Sch+15] Christian Schlegel, Alex Lotz, Matthias Lutz, Dennis Stampfer, Juan F. Inglés-Romero, and Cristina Vicente-Chicote. "Model-driven software systems engineering in robotics: Covering the complete life-cycle of a robot". In: *Journal IT - Information Technology: Methods and Applications of Informatics and Information Technology*. Ed. by Paul Molitor. Vol. 57. DE GRUYTER, 2015, pp. 85 –98.
DOI: `10.1515/itit-2014-1069`.

[Sen12] Séverine Sentilles. "Managing Extra-Functional Properties in Component-Based Development of Embedded Systems". Dissertation. Mälardalen University, Västerås, Sweden, 2012.

[SPA] SPARC. *European SPARC Robotics Initiative*.
URL: `http://sparc-robotics.eu`.

[SS14] Dennis Stampfer and Christian Schlegel. "Dynamic State Charts: composition and coordination of complex robot behavior and reuse of action plots". In: *Intelligent Service Robotics* 7.2 (2014), pp. 53–65. ISSN: 1861-2784.
DOI: `10.1007/s11370-014-0145-y`.

[SLS11b] Andreas Steck, Alex Lotz, and Christian Schlegel. "Model-driven engineering and run-time model-usage in service robotics". In: *Proceedings of the 10th ACM International Conference on Generative Programming and Component Engineering (GPCE)*. Portland, Oregon, USA, 2011. ISBN: 978-1-4503-0689-8.
DOI: `10.1145/2047862.2047875`.

[SS10] Andreas Steck and Christian Schlegel. "SmartTCL: An Execution Language for Conditional Reactive Task Execution in a Three Layer Architecture for Service Robots". In: *International Workshop on Dynamic languages for RObotic and Sensors systems (DYROS), 2nd Intl. Conf. on Simulation, Modeling, and Programming for Autonomous Robots (SIMPAR)*. Darmstadt, 2010, pp. 274–277. ISBN: 978-3-00-032863-3.

[Ste+11] Dave Steinberg, Frank Budinsky, Marcelo Paternostro, and Ed Merks. *EMF: Eclipse Modeling Framework*. Second. Addison-Wesley, 2011. ISBN: 978-0-321-33188-5.

[Sys12] SysML. *OMG Systems Modeling Language*. OMG Document formal/2012-06-01. Version 1.3. 2012.

[TVS14] Tadele Shiferaw Tadele, Theo de Vries, and Stefano Stramigioli. "The Safety of Domestic Robotics: A Survey of Various Safety-Related Publications". In: *IEEE Robotics Automation Magazine* 21.3 (2014), pp. 134–142. ISSN: 1070-9932.
DOI: `10.1109/MRA.2014.2310151`.

[UCM] UCM. *Unified Component Model for Distributed, Real-Time and Embedded Systems*. OMG Document ptc/2016-07-04. www.omg.org/spec/UCM/1.0/Beta1/.

[UML15] UML. *OMG Unified Modeling Language*. OMG Document formal/15-03-01. Version 2.5. 2015.

[Voe13] Markus Voelter. *DSL Engineering: Designing, Implementing and Using Domain-Specific Languages*. dslbook.org, 2013. ISBN: 978-1481218580.

[WIS13] Danny Weyns, M. Usman Iftikhar, and Joakim Söderlund. “Do External Feedback Loops Improve the Design of Self-adaptive Systems? A Controlled Experiment”. In: *Proceedings of the 8th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*. SEAMS ’13. San Francisco, CA, USA: IEEE Press, 2013, pp. 3–12. ISBN: 978-1-4673-4401-2.
URL: http://dl.acm.org/citation.cfm?id=2487336.2487341.

[Wey+13] Danny Weyns, Bradley Schmerl, Vincenzo Grassi, Sam Malek, Raffaela Mirandola, Christian Prehofer, Jochen Wuttke, Jesper Andersson, Holger Giese, and Karl M. Göschka. “Software Engineering for Self-Adaptive Systems II: International Seminar, Dagstuhl Castle, Germany, October 24-29, 2010 Revised Selected and Invited Papers”. In: ed. by Rogério de Lemos, Holger Giese, Hausi A. Müller, and Mary Shaw. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013. Chap. On Patterns for Decentralized Control in Self-Adaptive Systems, pp. 76–107. ISBN: 978-3-642-35813-5.
DOI: 10.1007/978-3-642-35813-5_4.

[Wil+08] Reinhard Wilhelm et al. “The worst-case execution-time problem - overview of methods and survey of tools”. In: *ACM Trans. Embedded Comput. Syst.* 7.3 (2008).

[Zil93] Shlomo Zilberstein. “Operational Rationality through Compilation of Anytime Algorithms”. Dissertation. University of California at Berkeley, 1993.