Technische Universität München
Fakultät für Informatik
Lehrstuhl für Echtzeitsysteme und Robotik

Contributions to System Composition using a System Design Process driven by Service Definitions for Service Robotics

Dennis Stampfer

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. Alexander Pretschner

Prüfer der Dissertation:

1. Prof. Dr.-Ing. habil. Alois Knoll
2. Prof. Dr. Christian Schlegel

Die Dissertation wurde am 04.12.2017 bei der Technischen Universität München eingereicht und durch die Fakultät für Informatik am 18.03.2018 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

Zusammenfassung

Die bestehende Praxis in der Softwareentwicklung für Serviceroboter ist vergleichbar mit der Kunst des Handwerks: Sie erfordert viel Erfahrung, großes Systemwissen und Einfluss auf alle Schichten des Systems und seiner Teile. Die Teile bestehender Systeme können nur schwer herausgelöst und durch Dritte als Baustein wiederverwendet werden, da die ursprünglich getroffenen Annahmen und Eigenschaften in der neuen Umgebung nicht mehr zutreffen. Für ihre Wiederverwendung müssen sie angepasst oder neu entwickelt werden. Die entstehenden Aufwände und Kosten stehen jedoch im Widerspruch zu kundenspezifischen Systemen wie sie in der Servicerobotik notwendig sind. Als interdisziplinäres Fachgebiet ist es für die Servicerobotik deshalb unverzichtbar, Softwarebausteine von Dritten ohne Aufwand flexibel zusammensetzen zu können, um so die effektive Interaktion verschiedener Experten zu ermöglichen.

Ziel dieser Arbeit ist es, die Softwareentwicklung in der Servicerobotik zu verbessern, indem der Schritt von der manuellen Integration hin zum *systematischen Konstruieren von Software durch Zusammensetzen* erzielt wird (engl. "System Composition"). Das Zusammensetzen ermöglicht die Zusammenarbeit und die Interaktion von Experten in einem Software-Ökosystem für die Robotik. Dieses Ziel wird durch *drei* Beiträge erreicht. *Erstens* verbessert diese Arbeit die *Zusammensetzbarkeit* von Softwarekomponenten, damit diese als Bausteine genutzt werden können. Dazu werden relevante syntaktische und semantische Informationen systematisch modelliert. Bisher sind diese Informationen meist innerhalb der Bausteine versteckt. *Zweitens* führt die Arbeit einen *zugehörigen Entwicklungsprozess* ein. Er erlaubt es, Strukturen zu modellieren und in Bausteinen zu nutzen, damit die Teilnehmer des Ökosystems Bausteine austauschen und flexibel zu neuen Systemen kombinieren können. Dies wird durch eine strikte Rollentrennung erreicht, um die Übergänge zwischen den Beteiligten zu organisieren. *Drittens* unterstützt der Ansatz mit einer *integrierten Entwicklungsumgebung* die Beteiligten des Ökosystems in der Umsetzung der Strukturen und leitet sie durch die Entwicklungsschritte. Die Arbeit beschreibt die Organisation eines Software-Ökosystems in drei Ebenen. Sie führt dazu eine modellgetriebene Meta-Struktur ein, um mittels Service-Definitionen domänenspezifische Strukturen basierend auf serviceorientierten Softwarekomponenten zu beschreiben. Diese ermöglichen die Bereitstellung, die Auswahl und das flexible Zusammensetzen von Softwarekomponenten zu Systemen.

Der vorgestellte Ansatz wurde in einer Nutzerstudie ausgewertet und in diversen Projekten mit einem breiten Anwendungsspektrum eingesetzt. Erkenntnisse dieser Arbeit trugen zur EU Horizont 2020 Innovationsmaßnahme "RobMoSys" bei, welche an einer europäischen digitalen Industrieplattform für die Robotik arbeitet.

Abstract

In recent years, many service robotics applications have emerged. However, the way they are being built resembles the art of crafting. It requires a high level of system knowledge, control, and influence on all parts and all levels of the system. The parts of these existing systems are hard to separate and to use in another system—particularly by third parties. Assumptions are made for the original system that are not expressed and that do not hold true when used in a different environment. As a result, the development of service robots requires high effort in modifying and adopting reused parts—or even rebuilding from scratch. The resulting high effort and costs are in contradiction with the high demand for small batch sizes of custom service robots. The reuse of third party software and effective collaboration among specialized players is therefore a must for service robotics due to its interdisciplinary nature.

This thesis improves software development for service robotics. Its aim is to advance from handmade crafting and integration of software to *systematic engineering of software based on system composition* in order to enable collaboration among different experts in a service robotics software business ecosystem. This is addressed by *three* contributions. *First*, the approach improves the *composability* of software components as building blocks. The approach allows to explicate and use syntactic and semantic knowledge that is relevant for composability but that is typically hidden within building blocks. *Second*, the approach introduces a *composition workflow* for establishing and using structures that manage the supply and use of building blocks. It introduces the organization of an ecosystem in three composition tiers. The workflow applies separation of roles via freedom from choice to manage the interfaces between the involved ecosystem participants. *Third*, the approach provides an *integrated development environment (IDE)* that supports and guides users in adhering to the proposed structures and workflow. The approach introduces a model-driven meta-structure based on service-oriented software components to establish domain-specific structures. They are based on "service definitions" that enable supplying, selecting, and using software components for system composition in an ecosystem.

The approach was successfully applied in several projects and activities to develop real-world service robotics applications. It was evaluated in a user study as well. Insights of this thesis contributed to the RobMoSys European Horizon 2020 project that works towards an EU Digital Industrial Platform for Robotics.

Acknowledgements

Writing a thesis is a journey and many people have accompanied me during this journey.

I would like to thank Prof. Dr. Knoll at Technical University of Munich for giving me the opportunity to prepare this thesis and for supporting my work.

Many thanks to Prof. Dr. Christian Schlegel at the Service Robotics Research Center at the Ulm University of Applied Sciences (Hochschule Ulm). Working in his research group was inspiring and motivating from the very beginning, but it became a great pleasure to actively contribute and shape the overall vision that is now way beyond what began with the SmartSoft framework. Christian, many thanks for fruitful discussions, day and night. Thanks for the constant motivation that slowly turned into the necessary bugging. Thanks for your endless support during this thesis. Finally, thanks for all the good restaurant recommendations, no matter in which corner of the world we were ;).

Thanks to Sandra Frank, Timo Blender, and other former colleagues whom I had the pleasure to work with here at the Service Robotics Research Center. Special thanks deserve Matthias Lutz and Alex Lotz. Without Matthias and Alex, I would probably not have survived this thesis. They turned from former fellow students to colleagues and friends and we shared countless discussions, arguments, respect, support, critics, work load, days, nights, ups, downs, sweets, travel, meetings, dinners, papers, etc.

Thanks to Simone and Armin for providing a desk with a nice view to work and wrap up the thesis. Many thanks to Martin for accompanying me in cold nights, high and remote in the mountains during the final phase of this thesis. Hunting for new perspectives while getting a fresh mind in the cold provided the badly needed distraction and boost in motivation in the final phase. I always enjoyed waiting for the stars to shine or the sun to rise. Good luck with your worms.

So many thanks to my family. Britta, thank you for your constant support. Thank you for being part of my journey. Thank you for your patience. Ellie, thank you for being there. Thanks for blowing away the typical day's frustrations, that one experiences during the final sprint, just by smiling. And always remember "how the lions do" :)

Walter, this thesis is what your motivation in science and technology led to.

Contents

1 **Introduction** 1

1.1 Research Questions . . . 7

1.2 Approach and Contributions . . . 9

1.3 Thesis Outline . . . 12

1.4 Publications . . . 13

2 **Related Work** 19

2.1 Software Business Ecosystems . . . 19

2.2 Software Engineering and Integration . . . 22

2.2.1 Software Development Processes and Collaborative Development . 22

2.2.2 Software Product Lines . . . 23

2.2.3 Component-Based Software Engineering . . . 24

2.2.4 Service-Oriented Architecture . . . 25

2.3 System Design and Modeling . . . 27

2.3.1 General-Purpose Modeling Languages . . . 27

2.3.2 Ontologies . . . 28

2.4 Model-Driven Software Development . . . 30

2.5 Domain-Specific Approaches . . . 32

2.5.1 Automotive Industry . . . 32

2.5.2 Smartphone Domain . . . 32

2.5.3 Industrial Automation and Industry 4.0 . . . 33

2.6 Software Development in Robotics . . . 34

2.6.1 Robotics Libraries . . . 34

2.6.2 Robotics Frameworks 34
2.6.3 Development Approaches and Robotics Software Infrastructures . . 35
2.6.4 Tooling in Robotics 39
2.6.5 MDSD and DSLs in Robotics 40
2.7 Related Initiatives and Ongoing Projects 41
2.8 The SmartSoft World 42
2.8.1 A Brief Review of the SmartSoft Framework 42
2.8.2 The SmartSoft Framework in the Context of this Work 43
2.9 The 3T Architecture in Robotics 44

3 Towards System Composition 47
3.1 Vision and Goal 48
3.1.1 System Composition 48
3.1.2 A Software Business Ecosystem for Robotics 51
3.1.3 Use-Cases 54
3.2 The Need for Structures 57
3.2.1 Benefits of Structure 59
3.2.2 The Need for a Meta-Structure (Composition Structure) 59
3.2.3 Service-Level Composition in Robotics Software Architectures 60
3.3 Consequences on the Approach 61
3.3.1 Composability 61
3.3.2 Composition Workflow 65
3.3.3 Support via Tooling 71
3.3.4 Summary of Consequences 74
3.4 Considerations 78
3.4.1 Collaborate by Structure (C0) 79
3.4.2 Manage Interfaces (C1) 84
3.4.3 Express Offer and Needs (C2 and C3) 85
3.4.4 Matchmaking (C4) 86
3.4.5 Configuration of a Building Block (C6) 87
3.4.6 Verifying the Composition (C5) 89
3.5 Summary 90

4 An Approach for System Composition 91
4.1 The Approach in a Nutshell 91
4.1.1 Composition Tiers 91
4.1.2 Roles for System Composition 93
4.1.3 Service-Based Composition Workflow 94
4.1.4 Service Definitions to Support Separation of Roles 96
4.2 An Architectural View on the Approach 97
4.2.1 The Three Parts of the Approach 97
4.2.2 Structural Building Blocks 101
4.2.3 Functional Building Blocks: Software Components 103
4.2.4 Compositions 103
4.3 A Workflow View on the Approach 104
4.3.1 Workflow Steps 104
4.3.2 The Composition Workflow: Intra-Organizational Perspective 106
4.4 Summary 107

5 A Meta-Structure for System Composition 109
5.1 Meta-Model Overview 110
5.2 Structural Building Blocks 112
5.2.1 Service Definitions 112
5.2.2 Communication Data Structures 114
5.2.3 Communication Semantics 115
5.2.4 Service Properties 115
5.2.5 Reusable Variation Points: Parameter Sets 122
5.2.6 Documentation 124
5.3 Functional Building Blocks 125
5.3.1 Component Meta-Model 125
5.3.2 Service Endpoints 127
5.3.3 Instantiation of Service Properties 129
5.3.4 Service Endpoint Mapping 130
5.3.5 Parameterization 131
5.3.6 Component Documentation 133
5.4 Building Robot Applications: Compositions 135

5.4.1 Service Wish List . . . 136
5.4.2 Component Retrieval . . . 137
5.4.3 System Configuration . . . 142
5.4.4 Deployment . . . 148
5.5 Summary . . . 149

6 The SmartMDSD Toolchain 151
6.1 Considerations and Overall Setting . . . 151
6.1.1 Extension vs. Direct Realization . . . 152
6.1.2 Technologies used in the SmartMDSD Toolchain . . . 154
6.2 The Composition Workflow . . . 155
6.2.1 Overview on Workflow Steps and Projects . . . 155
6.2.2 Modeling Perspective . . . 156
6.2.3 Code-Generation and Artifacts . . . 158
6.3 SmartMDSD Toolchain Walkthrough . . . 161
6.3.1 Introduction and Overview . . . 161
6.3.2 System Design Step . . . 163
6.3.3 Component Development Step . . . 167
6.3.4 System Composition Step . . . 173
6.4 Summary . . . 179

7 Demos, Application, and Evaluation 181
7.1 Application Fields . . . 181
7.2 Maturity of the SmartMDSD Toolchain . . . 182
7.3 Demonstrators and Activities . . . 183
7.3.1 The (Collaborative) Robot Butlers . . . 183
7.3.2 Personal Mobile Navigation . . . 190
7.3.3 Industry 4.0 Robot Commissioning Fleet in Intra-Logistics . . . 195
7.3.4 Transportation Tasks in a Hospital . . . 196
7.3.5 Application in Education . . . 197
7.4 User Study . . . 198
7.5 Evaluation in Project-Context . . . 199
7.5.1 Benefits for Project Collaboration . . . 200

7.5.2 Benefits for Software Development 201
7.5.3 Benefits for System Composition 202
7.6 Discussion 206
7.7 Summary 210

8 Conclusion 211
8.1 Applicability and Relevance 213
8.2 Future Work 214

Glossary 217

Abbreviations 225

List of Figures 227

List of Tables 231

References 233

1 Introduction

Intelligent technology is becoming a part of our lives. The border between a mechatronic device and an intelligent service robot is vanishing. A service robot[1] is an autonomous system that performs useful tasks for humans [Int12] in a shared environment. We can soon expect more and more service robots to enter our lives—for example, as co-workers or as household assistants, as well as farming, delivery, and logistics robots.

So far, research in service robotics is a relatively young discipline that is focused primarily on solving particular technical challenges or problems. These are related to the basic functionalities of a robot, such as solving Simultaneous Localization and Mapping (SLAM), motion control, object recognition, and object manipulation, among others. Only as first solutions to basic functionalities began to emerge, the focus recently began to shift to demonstrating complete systems such as robots playing pool [NKH11] or fetching drinks [Boh+11]. These robots were no longer demonstrating a basic functionality; rather, they were demonstrating a relevant robot application in the real world. They were, however, still prototypes for technology demonstration.

As the focus shifts from individual technology demonstrations to the demonstration of complete robot applications, this comes with a shift from individual technologies to a combination of many different technologies in a full robot. Given the interdisciplinary nature of robotics, robotics is a science of integration (see [euR16]). One of the main efforts in building a robot is software development [HBK11, p. 339]. So, the combination and integration of individual technological contributions must be addressed particularly at the software level. This is necessary in order to develop methods and tools that push robotics to the next level via a step change [euR16]. Until recently, software development in robotics was considered a means to an end: It was carried out as a sideline piece of work while focusing attention and effort on solving the technical

[1] "Robotics" and "robot" in this thesis generally refer to "service robotics" and "service robot".

challenges of robot capabilities.

As a result, current practice in software development in service robotics resembles crafting (Fig. 1.1). When crafting robotics applications, the application is broken down into parts. These parts are solved individually at a lower level of complexity. There are implicit and explicit agreements between the parts, as well as between the parts and the whole to which they belong. There is high control and influence on the individual parts of the developed software, which allows managing these agreements. Reuse with existing software, such as past in-house or third party developments, is made possible by shaping and modifying the software to integrate into the system—for example, thanks to open source software. This is illustrated in Fig. 1.1. There is an analogy with the jigsaw puzzle (see also section 3.1.1): Sawing the wooden plate apart produces many pieces where only two of them will fit together (e.g. A and B, Fig. 1.1). To build a jigsaw puzzle, one literally first requires the overall picture to be painted on the wooden plate. Only then can it be sawed apart. Connecting two arbitrary pieces would require modification or adaptation (e.g. connecting X with C and B, Fig. 1.1).

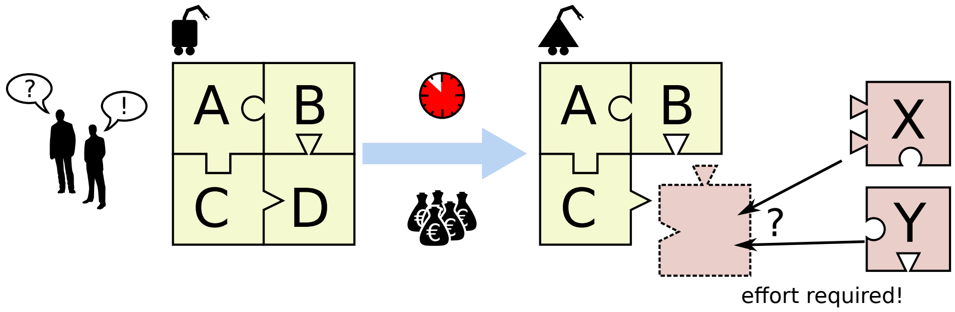


Figure 1.1: Current practice in software development in service robotics resembles crafting. A puzzle is an adequate analogy.

Handmade crafting has been sufficient so far for prototypes and technology demonstrations of a basic functionality. However, it is not sufficient in the long term, where more and more robotics applications shall be available as commercial products. At present, even slight changes in the application's requirements lead to huge development efforts when developing a robotics application (Fig. 1.2). This is because the parts are carefully woven together. Ripping the software apart for reuse in another robotics application—or even exchanging parts in the existing application when requirements change—results in extraordinarily high integration efforts since standardized interfaces are missing and agreements that were taken for granted no longer hold in the new environment. Furthermore, when exchanging pieces of the robot application, it is not clear what is a potential replacement (e.g. Fig. 1.1: choose X or Y as a replacement for D?). More effective and cost-efficient ways to build and modify the software of service robots are needed in order to lower the huge costs of service robots, which are sold in low quantities. Especially Small and Medium-Sized Enterprises (SMEs), which have a high potential for innovations, would benefit from that.

Exchange, collaboration, and the use of the knowledge and development results of others are mandatory for robotics as an interdisciplinary field. However, even robotics experts find it hard

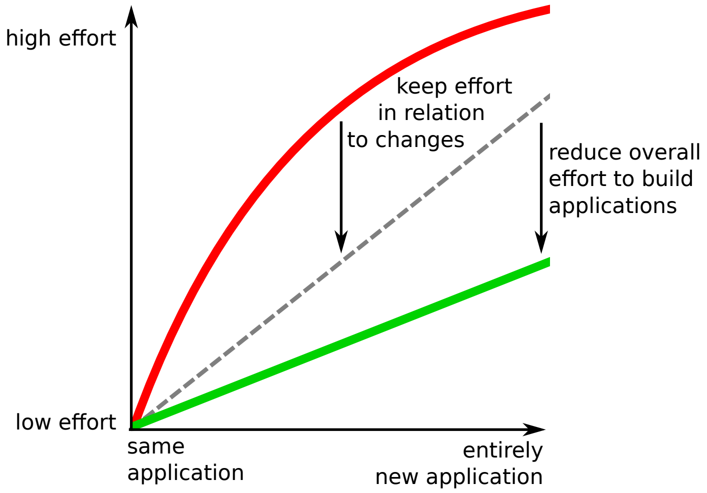


Figure 1.2: The relation of changes to development efforts. Even slight changes in the application's requirements lead to huge development efforts (red curve). However, the efforts and cost should be in relation to changes. It is even desirable to reduce the overall effort by composing from existing building blocks.

to select third party software such that it suits their needs and such that it integrates into the application without breaking system consistency. Given this situation, how should companies from non-robotics domains with promising robotics use-cases access and make use of robotics technology? Robotics would benefit from new domains that make use of robotics technology [euR13]. The limited access to robotics software solutions hinders the flexible combination of existing solutions. But being able to combine is an enabler for new innovations. Advancing from fixed value-chains to flexible value-networks— as is common in other domains—remains an unused potential for service robotics.

To cope with the complexity, the interdisciplinarity, the efforts and costs, the lack of access to robotics software, and the lack of means for collaboration, *robotics has to make the step change from handmade crafting and integration to a systematic engineering approach based on composition*. A compositional approach in the context of a software ecosystem allows a much better separation of roles and separation of concerns to address these issues.

In a composition-oriented approach (Fig. 1.3, right), software building blocks can be used in an "as-is" manner. They can be combined and re-combined to form new applications with low effort depending on the application's needs. This should be as easy as building with plastic Lego blocks for children. Any two pieces can be put upon each other at much simpler complexity: flexibly combining, re-combining, and exchanging the pieces as needed. Composition is a flexible approach for building applications from existing parts. These existing parts adhere to superordinate structures (e.g. standardized interfaces) and means for configuration from the outside but without modifying the building block itself.

An ecosystem is a collaboration model [BB10; IL04] that describes the many ways and advantages in which experts from various fields or companies can collaborate around a domain or

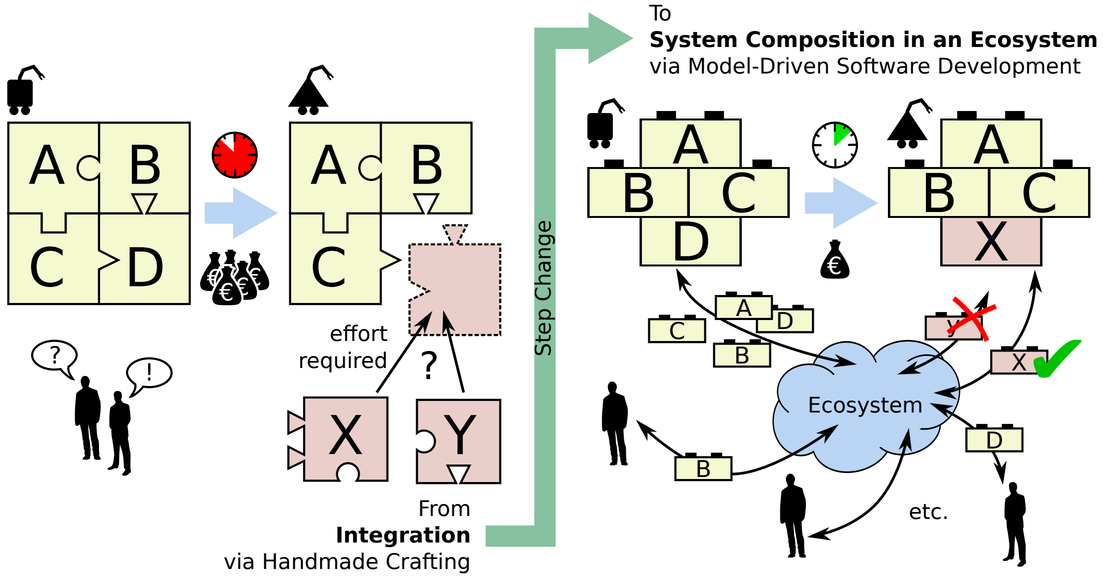


Figure 1.3: The thesis addresses the step change from integration by handmade crafting to systematic engineering of software based on system composition in an ecosystem.

product (Fig. 1.4). In an ecosystem, several participants collaborate, compete, and share efforts and costs by sharing building blocks (supplying and using building blocks, Fig. 1.4). The "collaboration" of participants in an ecosystem refers to complementing each other, and to sharing independent and self-contained development artifacts. Collaboration is not meant in the sense of close collaboration as in working in a team and collaborative editing.

There is no need for the participants of the ecosystem to know each other in order to negotiate technical agreements. There should also be no need for an ecosystem-wide software development process to manage these agreements. Instead of managing the collaboration through software development processes, the interfaces between the involved participants and artifacts should be managed by structures to which the contributions of the participants adhere. Introducing and adhering to such superordinate structures for system composition immediately makes accessible all parts (own or from a third party) that adhere to the same structure. The collaboration between participants does not require strong bilateral interaction: They can work in a distributed manner in time and space to supply or use a building block (Fig. 1.4). Such structures are still missing for robotics. The thesis contributes to this end.

The systematic engineering of software based on system composition in an ecosystem brings several benefits (Fig. 1.5):

- Composition reduces effort and costs for new applications or when changing existing applications. Components are instantly available and one can rely on parts that were developed by experts. These have a potentially better quality than components developed in-house [Frö02].

- Composition allows one to access and benefit from existing solutions. Knowledge of these

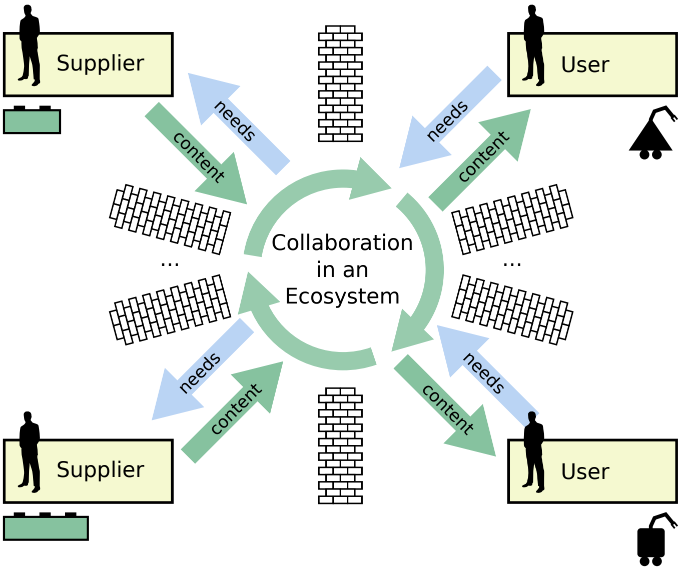


Figure 1.4: A robotics business ecosystem and the collaboration of its main participants. Suppliers provide building blocks for users to build robotics applications. The participants must not know one another but must still be able to collaborate—i.e. they must complement each other and share independent and self-contained development artifacts.

solutions immediately becomes available to others, opening up the possibilities to address new applications that companies otherwise would be unable to address on their own due to the lack of expertise. This is a basis for robotics to open up for new domains. Further, it forms the basis for advancing from a technology push to a use-case-driven technology pull which can make robotics advance in the relevant domains and applications.

- Composition allows moving from value-chains to value-networks in which participants can cooperate and collaborate flexibly. This is of special interest to SMEs since it creates the potential for specialization to fill up niches in the market (see [Jan12]).
- Since composition simplifies the combination of existing solutions and accessing them, it provides a setting for new innovations: One of the drivers of innovation is the combination of existing technologies [Dör13; Wit12; euR13] in new ways or to form new applications.
- Improving the software development of service robots lowers costs and thus the price of the end-product. Improving the way in which robots are built will also improve the quality and the capabilities of the robot itself [euR13, p. 70].

This thesis proposes an approach to contribute to a step change in robotics software engineering from integration and handmade crafting of software to systematic engineering of software for service robotics. This will enable the composition of building blocks to systems (thus "system composition") in a robotics business ecosystem. This is achieved (i) by addressing composability as the *ability* to flexibly combine and recombine a building block, (ii) by addressing

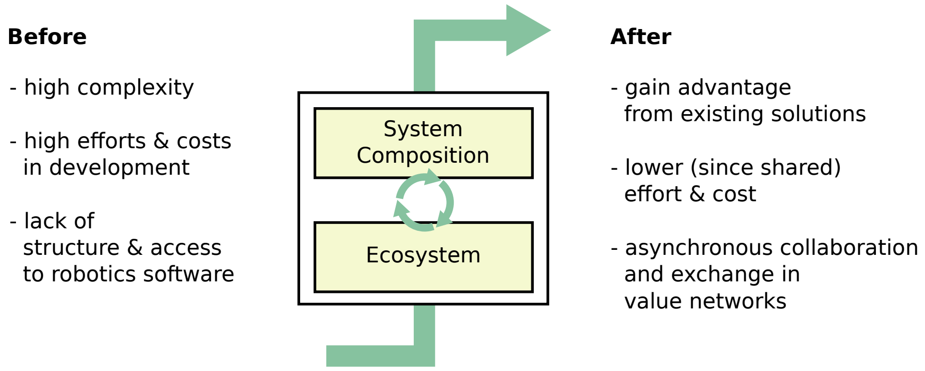

Before
- high complexity
- high efforts & costs in development
- lack of structure & access to robotics software
System Composition
Ecosystem
After
- gain advantage from existing solutions
- lower (since shared) effort & cost
- asynchronous collaboration and exchange in value networks

Figure 1.5: The current situation in robotics (left) triggers the need for system composition as the next step in software development for robotics (right) and the benefits of this step change.

the composition workflow to organize the *activity* of composition, and (iii) by providing *support* to users via model-driven tools which give access to the approach and which guide the user through the approach (Fig. 1.6). The thesis provides a Service Definition Language (SDL) (an Interface Definition Language (IDL) for services) to standardize the services of software components and to explicate information that is relevant for composition but usually remains hidden or implicit. It provides means for component selection and introduces a model-driven workflow for system composition based on the principles of Service-Oriented Architectures (SOAs) and Component-Based Software Engineering (CBSE). It is illustrated by the example of the SmartSoft framework [Scho4a]. An outcome are contributions to the integrated Smart-MDSD Toolchain that provides support for users and guides them through the composition workflow. The toolchain has been applied in several research projects and activities, and was used to develop the demonstrators that are presented in this thesis. The demonstrators show that it is possible to effectively build robotics applications in just a few hours. This is possible thanks to the composition of existing building blocks from different roles in an ecosystem approach.

Insights from this thesis contributed to the RobMoSys European Horizon 2020 project that is working towards an EU Digital Industrial Platform for Robotics. The funding of RobMoSys on EU-level underlines the relevance of a model-driven and composition-oriented approach that is addressed by this thesis.

The remaining parts of this chapter describe the research questions that build the foundation for this thesis. The chapter then summarizes the approach and the contributions, and provides the thesis outline. The chapter closes with a list of (own) related publications.

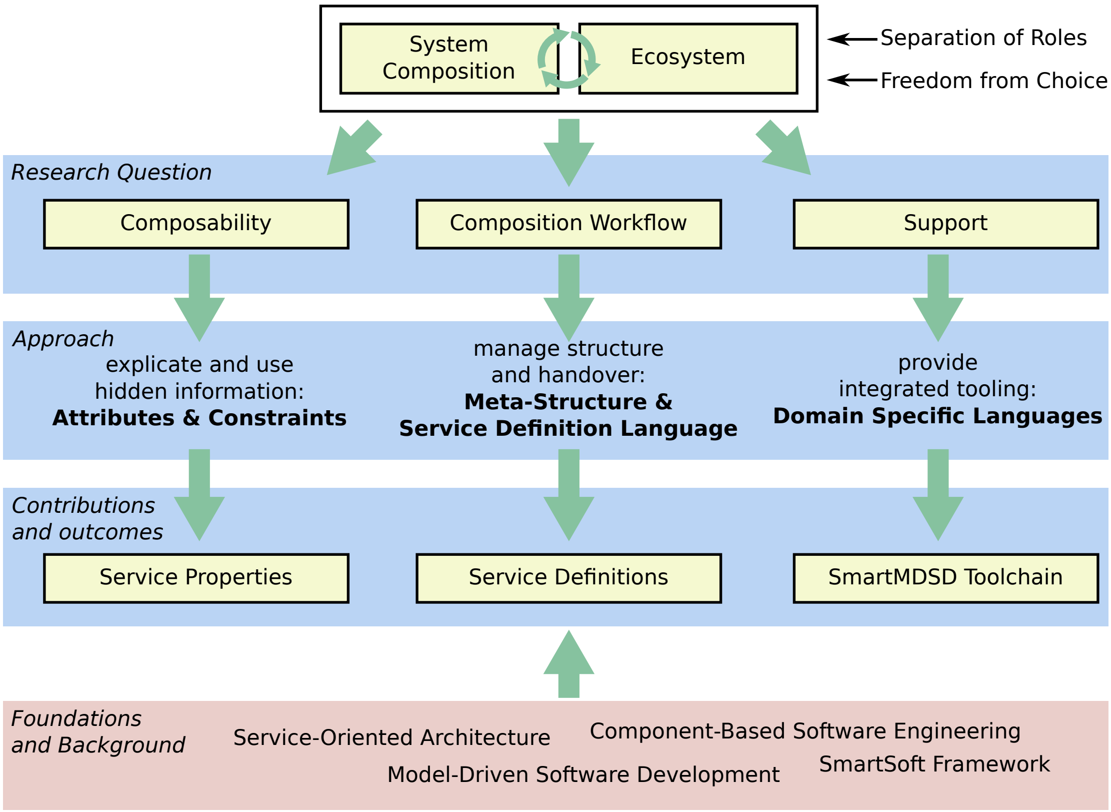

System Composition
Ecosystem
Separation of Roles
Freedom from Choice
Research Question
Composability
Composition Workflow
Support
Approach
explicate and use hidden information:
Attributes & Constraints
manage structure and handover:
Meta-Structure & Service Definition Language
provide integrated tooling:
Domain Specific Languages
Contributions and outcomes
Service Properties
Service Definitions
SmartMDSD Toolchain
Foundations and Background
Service-Oriented Architecture
Component-Based Software Engineering
Model-Driven Software Development
SmartSoft Framework

Figure 1.6: The research questions and approach of this thesis. Systematic engineering of software for service robotics based on composition in an ecosystem is achieved by addressing composability, a composition workflow, and support for the separated roles.

1.1 Research Questions

The overall research question of this thesis is provided below:

> *How can software development for service robotics be improved in order to advance from handmade integration and crafting to systematic engineering of software based on system composition in a software ecosystem?*

In the overall vision of a software ecosystem for service robotics, various experts can collaborate and compete to exchange building blocks. Separation of roles is a mandatory principle that supports finding adequate structures for system composition in an ecosystem (Fig. 1.6). Separation of roles enables experts to work independently (separation in time and space). Applying freedom *from* choice over freedom *of* choice provides support for realizing separation of roles: freedom *from* choice positively limits the set of available options to gain composable structures that allow bringing together (compose) the parts that are contributed by different roles.

Performing the step change from crafting and integration to system composition requires addressing composability as the *ability* to flexibly combine and recombine a building block (software component), to establish a composition workflow to organize the *activity* of composition, and to provide *support* to users via tools. The overall research question is refined for each of these three topics (Fig. 1.6):

> *1. How to improve the **composability** of building blocks (software components) such that they not only fit together technically, but also work together in a meaningful way in the overall application?*

Composability is the *ability* to combine and recombine building blocks into different systems for different purposes [PW03]. From the building block's perspective, it is the property that makes a "part" become a "building block". Thus, both the parts and the system that is formed by these parts need to be addressed as basic prerequisites for system composition. Aspects of composability exist among each of the components in a system. Aspects of composability also exist among the application's needs and the components meeting these needs. Composability comprises syntactic aspects to put parts together at all (e.g. data types) as well as semantic aspects (e.g. send or send–reply interactions) to put the parts together in a meaningful way (see [PW03]). Both must be considered on the technical level (e.g. interface for localization) and on the application-level (e.g. quality of localization). All these aspects need to be considered to ensure that systems composed from individual components work as intended. Further, system builders must be supported to identify the component from the market that fits their needs.

> *2. How to organize the building blocks (software components) in an overall **composition workflow** that decouples and manages both the stakeholders and the parts that they supply or use to collaborate in an ecosystem?*

Composition is the *activity* of putting together building blocks. This cannot happen by coincidence; rather, it must be carefully designed (cf. [Szy03]). Composability cannot be viewed as a self-contained property in isolation. It must be considered as a cross-cutting concern. Collaboration in an ecosystem brings special needs for composability. Thus, composability must be considered through the overall composition workflow. This requires developing a suitable structure that decouples the development of parts from the composition of parts and the according stakeholders (separation of roles). The parts must still maintain composability without the need to organize them in an overall development process[2], which is not possible [Bos09] in an ecosystem approach. Such a structure needs careful consideration of the necessary elements and suitable abstractions.

> *3. How to design an integrated tool that **supports users** in modeling and composition of models and corresponding software artifacts through the workflow for system composition?*

Support for users is not just a means to an end that simplifies development. It is an essential aspect for enabling composability and system composition. The stakeholders that develop

[2] A development process or methodology such as waterfall, Rational Unified Process, etc.

and compose components to systems need methods, structures, and implementations in frameworks. They also require supporting tools that make these methods and structures accessible in a consistent way. This enables the stakeholders to gain from the benefits of compositional approaches. The stakeholders must not be outrun by the effort and costs that result from applying a certain approach. Tools support users by guiding them through the composition workflow. They ensure consistent structures to maintain composability. There are several tasks and roles along the composition workflow that each require different languages, tools, or views. All of them need adequate interfaces to come together in an integrated tooling. An integrated tooling must enable structures, languages, and views to work together. This means that individual structures must be balanced and proper connections must exist. It goes beyond merely overcoming the isolation of dedicated tools.

1.2 Approach and Contributions

To address the research questions, the approach presented in this thesis can be summarized as follows:

> *The approach (Fig. 1.7) presented in this thesis* ***provides a meta-structure*** *(Tier 1, Fig. 1.7) for system composition based on Component-Based Software Engineering (CBSE) and Service-Oriented Architecture (SOA). It applies Model-Driven Software Development (MDSD) to allow domain experts to* ***create domain-specific structures*** *(Tier 2, Fig. 1.7). These structures support ecosystem users in supplying, selecting, and using software components to build robotics applications (Tier 3, Fig. 1.7). The approach allows to* ***explicate knowledge*** *that is relevant for composition but that usually remains hidden. The approach* ***manages the knowledge in an overall composition workflow****. An integrated* ***toolchain*** *supports the users in applying the approach to gain from its benefits.*

The approach uses the service-oriented component-based software framework SmartSoft [Sch04a] as a basis. Thanks to its service orientation and component-based nature, SmartSoft provides the basic technical prerequisites for system composition. This thesis applies Model-Driven Software Development (MDSD) on top of the SmartSoft framework to enable system composition. The thesis improves software development of service robots in the following aspects:

Composability. The approach improves composability by modeling information that is relevant for composability. This information is often not expressed or remains hidden within building blocks. Expressing information covers syntactic information as well as semantic properties of services of a software component. Explicating and using such information throughout the workflow ensures that the pieces not only fit together but also work together as intended.

Composition Workflow. The approach introduces a workflow and meta-structure to establish and use domain structures. These align the contributions from different ecosystem parti-

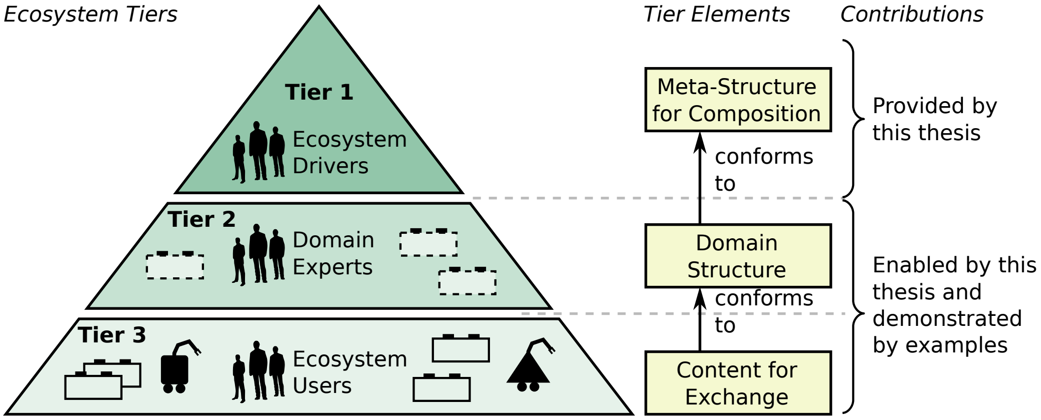


Figure 1.7: The organization of an ecosystem in three composition tiers. The approach provides a meta-structure for system composition in such a way that domain experts can create domain-specific and composable structures. The structures support ecosystem users in selecting and using software components to build robotics applications through composition.

cipants for system composition. The workflow uses strict separation of roles and enables using freedom *from* choice over freedom *of* choice to manage the involved stakeholders and the handover of artifacts throughout the composition workflow.

Support. To support users, the thesis provides an integrated development environment that provides graphical and textual modeling to create and use the domain structures.

Composability, a composition workflow, and support for users influence each other. Solutions covering all these aspects might even contradict each other. Therefore, it is necessary to address them in a holistic way to find the right balance between these topics in order to improve the systematic engineering of software in service robotics based on system composition in an ecosystem. This thesis carefully weighs the topics and contributes the following outcomes:

Service Definitions. Service definitions form the meta-structure for system composition and are modeled using a Service Definition Language (SDL) (an Interface Definition Language (IDL) for services) to define standardized services of software components. Service definitions are reusable elements in the composition workflow and are considered the basic architectural entities for decoupling component development from system composition. They are used and refined by all roles throughout the workflow to supply software components or use them for composition while ensuring composability. Service definitions are refined to describe the offer or need of software components. Service definitions thus improve the composability of software components and the collaboration of stakeholders within a composition workflow in an ecosystem.

Service Properties. Service properties are used within service definitions. They allow expressing the semantics of a service on the application-level, which otherwise may remain hidden within an implementation or within the documentation. Service properties improve the composability of software components, improve component selection, and ensure a valid composition by using constraints on properties.

Meta-Models and Workflow. The thesis provides meta-models that support a composition workflow to organize the ecosystem into three composition tiers. The workflow is based on service definitions. The meta-models allow the modeling of domain structures, the modeling of components, the composition of components, and deployment to the robot. They manage the handover between workflow steps and roles to make the structures accessible for tooling. Graphical and textual Domain-Specific Languages (DSLs) are provided to users so that they can create and use the models.

SmartMDSD Toolchain. This thesis contributes to the SmartMDSD Toolchain, an Integrated Development Environment (IDE) for engineering software for service robotics. The toolchain implements the meta-structure for system composition. It supports users in applying the composition workflow, starting with modeling of the domain structures, including the development of components, the selection and composition of components to applications, and their deployment to the robot. Matchmaking for component selection applies signature matching [Bac+02] of services and constraints evaluation to find a component that matches the needs of the application. The toolchain provides seamless integration of several DSLs to model the different elements by the different roles in the workflow.

Benefits of the Approach

The outcomes of this thesis achieve systematic engineering of software for service robotics based on system composition. The benefits of applying the outcomes are as listed below:

- They improve the quality, efficiency, and consistency of the software development process for service robotics.
- They simplify the reuse of existing solutions in new applications. Thus, they lower the effort for changing existing systems and developing new ones by composition.
- They improve the collaboration of participants in an ecosystem by improving the separation of roles and the separation of concerns.
- They ease access to the service robotics domain by structuring and simplifying component selection thanks to standardized services and expressed properties.
- They raise the level of abstraction in software development for service robotics from a code-driven or document-driven approach to a model-driven approach. This supports the transition from handmade crafting to systematic engineering.
- They reduce the cognitive load on the developer. The developer is supported to maintain an overview on the application thanks to checks for system consistency.

- They support in mastering the complexity of the overall system thanks to the improved collaboration of roles: better separation of roles and managing the interfaces to handover artifacts between roles.
- They improve the robustness and quality of the robot itself since improving the method of construction improves the performance of the outcome ("How a robot system is constructed determines how well it functions" [euR13, p. 70]).

The results of this research have been applied in research projects and other collaborations. They were used to develop functional demonstrators that showcase the capacity and benefits of the approach for system composition in an ecosystem. The approach is implemented in the SmartMDSD Toolchain, which is available under an open source license at http://www.servicerobotik-ulm.de.

1.3 Thesis Outline

Figure 1.8 illustrates the outline of this thesis, beginning with chapter 2.

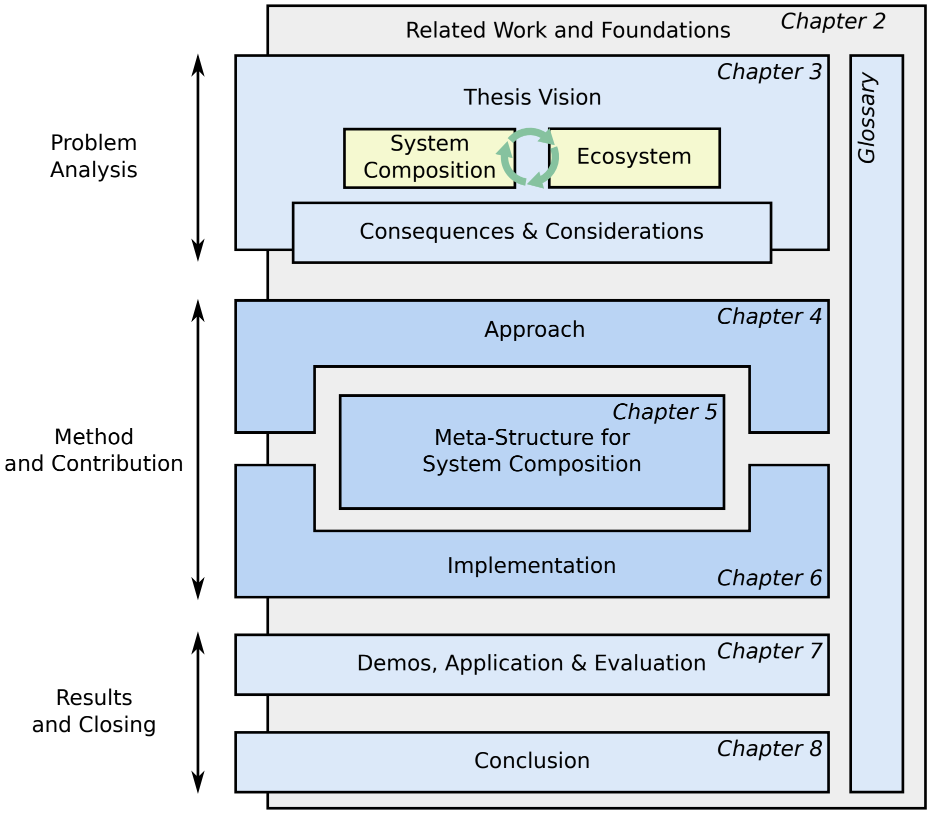


Figure 1.8: The thesis outline beginning with chapter 2.

Chapter 2 describes selected existing methods and approaches that are related to software development and system composition in a robotics software business ecosystem.

Chapter 3 describes the vision of composition and the vision of a robotics software business ecosystem. It will identify the need for structures in order to realize this for robotics. The chapter describes the consequences to such a structure and discusses the considerations to address them.

Chapter 4 outlines the main contributions of this thesis to provide the structures that enable system composition in a robotics software business ecosystem. It provides the overall setting and introduces the overall ecosystem organization in three composition tiers.

Chapter 5 contributes the meta-models and other necessary details that enable the vision of this thesis.

Chapter 6 presents the implementation of the approach and the contributions to the SmartMDSD Toolchain, an IDE for software development in service robotics.

Chapter 7 presents concrete robot systems and the initiatives that have built them. It presents a user study and evaluates the benefit of applying the thesis' contributions to systematic engineering of software based on system composition.

Chapter 8 concludes the thesis. It summarizes the contributions, and explains their applicability and relevance. The thesis ends with an outlook on future work.

Glossary describes the terms used in this thesis.

1.4 Publications

The following publications are related to this thesis and contain parts that are presented within this thesis.

Journal Publications

- **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: *Journal of Software Engineering for Robotics (JOSER): Special Issue on Domain-Specific Languages and Models in Robotics (DSLRob)* 7.1 (July 2016). ISSN 2035-3928, pp. 3–19.
  URL: https://joser.unibg.it/index.php?journal=joser&page=article&op=view&path[]=91.

- 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* 57.2 (Mar. 2015).

ISSN (Online) 2196-7032, ISSN (Print) 1611-2776, DE GRUYTER, pp. 85–98.
DOI: 10.1515/itit-2014-1069.

- 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". In: *International Journal of Information System Modeling and Design (IJISMD)* 5.3 (2014). IGI Global, ISSN 1947-8186, pp. 55–74.
  DOI: 10.4018/ijismd.2014070103.

- **Dennis Stampfer** and Christian Schlegel. "Dynamic State Charts: Composition and Coordination of Complex Robot Behavior and Reuse of Action Plots". In: *Journal of Intelligent Service Robotics* 7.2 (Mar. 2014). Springer Berlin Heidelberg, pp. 53–65. ISSN: 1861-2784.
  DOI: 10.1007/s11370-014-0145-y.

Conference Contributions

- 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: *IEEE International Conference on Simulation, Modeling, and Programming for Autonomous Robots (SIMPAR)*. San Francisco, CA, USA, Dec. 2016, pp. 170–176.
  DOI: 10.1109/SIMPAR.2016.7862392

- 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: *6th International Workshop on Domain-Specific Languages and models for ROBotic systems (DSLRob '15)*. Hamburg, Oct. 2015.
  URL: http://arxiv.org/abs/1601.02379.

- 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: *Workshop Roboter-Kontrollarchitekturen, co-located with Informatik 2014*. Ed. by E. Plödereder, L. Grunske, E. Schneider, and D. Ull. Vol. P-232. GI-Edition – Lecture Notes in Informatics (LNI). ISBN: 978-3-88579-626-8. Stuttgart: Bonner Köllen Verlag, Sept. 2014.
  URL: https://www.gi.de/service/publikationen/lni/gi-edition-proceedings-2014/gi-edition-lecture-notes-in-informatics-lni-p-232.html.

- 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: *Workshop Roboter-Kontrollarchitekturen, co-located with Informatik 2013*. Vol. P-220. GI-Edition – Lecture Notes in Informatics (LNI). ISBN: 978-3-88579-614-5. Koblenz: Bonner Köllen Verlag, Sept. 2013.

URL: https://www.gi.de/service/publikationen/lni/gi-edition-proceedings-2013/gi-edition-lecture-notes-in-informatics-lni-p-220.html.

- Matthias Lutz, **Dennis Stampfer**, and Christian Schlegel. "Probabilistic Object Recognition and Pose Estimation by Fusing Multiple Algorithms". In: *Proc. IEEE International Conference on Robotics and Automation 2013 (ICRA).* Karlsruhe, Germany, May 2013, pp. 4244–4249.
  DOI: 10.1109/ICRA.2013.6631177.

- **Dennis Stampfer** and Christian Schlegel. "Dynamic State Charts: Composition and Coordination of Complex Robot Behavior and Reuse of Action Plots". In: *Proc. IEEE International Conference on Proceedings of Technologies for Practical Robot Applications 2013 (TePRA).* ISBN: 978-1-4673-6224-5. Woburn, Massachusetts, USA, Apr. 2013, pp. 1–6.
  DOI: 10.1109/TePRA.2013.6556375.

- **Dennis Stampfer**, Matthias Lutz, and Christian Schlegel. "Informed Active Perception with an Eye-in-hand Camera for Multi Modal Object Recognition". In: *Proc. IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS), Workshop on Active Semantic Perception 2012 (ASP'12).* ISBN: 978-972-8822-26-2. Vilamoura, Algarve, Portugal, Oct. 2012.

- Matthias Lutz, **Dennis Stampfer**, Siegfried Hochdorfer, and Christian Schlegel. "Probabilistic Fusion of Multiple Algorithms for Object Recognition at Information Level". In: *Proc. IEEE International Conference on Technologies for Practical Robot Applications 2012 (TePRA).* ISBN: 978-1-4673-0854-0. Woburn, MA, USA, Apr. 2012, pp. 139–144.
  DOI: 10.1109/TePRA.2012.6215668.

- **Dennis Stampfer**, Matthias Lutz, and Christian Schlegel. "Information Driven Sensor Placement for Robust Active Object Recognition based on Multiple Views". In: *Proc. IEEE International Conference on Technologies for Practical Robot Applications 2012 (TePRA).* ISBN: 978-1-4673-0854-0. Woburn, MA, USA, Apr. 2012, pp. 133–138.
  DOI: 10.1109/TePRA.2012.6215667.

- **Dennis Stampfer** and Christian Schlegel. "Modellierung und Reuse komplexer Verhalten als Bausteine mit Dynamic State Charts am Beispiel der Servicerobotik". In: *Proc. Embedded Software-Engineering Kongress 2011 (ESE).* ISBN: 978-3-8343-2405-4. Sindelfingen: Vogel-Business-Media Würzburg, Dec. 2011, pp. 511–515.
  URL: http://d-nb.info/1082052493.

- Andreas Steck, **Dennis Stampfer**, and Christian Schlegel. "Software-Engineering in der Servicerobotik – Der Weg zum modellgetriebenen Softwareentwurf". In: *Proc. Embedded Software-Engineering Kongress 2009 (ESE).* ISBN: 978-3-8343-2402-3. Sindelfingen: Vogel-Business-Media Würzburg, 2009, pp. 513–516.
  URL: http://d-nb.info/999874837.

- Andreas Steck, **Dennis Stampfer**, and Christian Schlegel. “Modellgetriebene Softwareentwicklung für Robotiksysteme”. In: *Proc. 21. Fachgespräch Autonome Mobile Systeme 2009 (AMS)*. ed. by Rüdiger Dillmann, Jürgen Beyerer, Christoph Stiller, J. Marius Zöllner, and Tobias Gindele. Informatik Aktuell. Karlsruhe: Springer Berlin Heidelberg, Dec. 2009, pp. 241–248. ISBN: 978-3-642-10284-4.
  DOI: 10.1007/978-3-642-10284-4_31.

Talks

Besides accompanying talks to the above conference or workshop publications, the following notable talks were given to disseminate the research related to this thesis:

- Christian Schlegel and **Dennis Stampfer**. “The SmartMDSD Toolchain: Supporting dynamic reconfiguration by managing variability in robotics software development”. In: *Tutorial on Managing Software Variability in conjunction with the Robot Control Systems at Robotics: Science and Systems Conference (RSS 2014)*. Berkeley, CA, USA, July 2014.

- Christian Schlegel, Alex Lotz, Matthias Lutz, and **Dennis Stampfer**. “Supporting Separation of Roles in the SmartMDSD-Toolchain: Three Examples of Integrated DSLs”. In: *5th International Workshop on Domain-specific Languages and Models for Robotic Systems (DSLRob) in conjunction with the 2014 International Conference on Simulation, Modeling, and Programming for Autonomous Robots (SIMPAR 2014)*. Bergamo, Italy, Oct. 2014.

Other Publications

Other publications that disseminate the research conducted in this thesis:

- **Dennis Stampfer**, Alex Lotz, and Christian Schlegel. *FIONA Deliverable D2.2.1 State of the Art on Service-Oriented Software Component Models*. Project Deliverable. Mar. 2014.

- **Dennis Stampfer**, Christian Schlegel, Mathias Bürger, Christopher Brown, Wei Mao, Mitja Pugelj, Neda Petreska, Ali Golestani, Stefan Rueping, and Çağlar Akman. *FIONA Deliverable D2.3.1: FIONA Platform Architecture*. Project Deliverable. June 2015.

- **Dennis Stampfer**, Christian Schlegel, and Sandra Frank. *FIONA Deliverable D2.4.1 Handbook for Integrating Basic Services and Interface Components*. Project Deliverable. Mar. 2016.

- Mathias Bürger, Irene Süßemilch, **Dennis Stampfer**, Christian Schlegel, Florian Schreiner, Stefan Rueping, Christopher Zimmer, and Ali Golestani. *ITEA 2 Projekt: Framework for Indoor and Outdoor Navigation Assistance (FIONA): Abschlussbericht*. 2016.
  DOI: 10.2314/GBV:880711841.

- Alex Lotz, **Dennis Stampfer**, Christian Schlegel, Enea Scioni, Nico Huebel, Herman Bruyninckx, Matteo Morelli, Chokri Mraidha, and Sara Tucci. *RobMoSys D2.1: Modeling Foundation Guidelines and Meta-Meta-Model Structures*. Project Deliverable. June 2017.

- Alex Lotz, **Dennis Stampfer**, Christian Schlegel, Enea Scioni, Nico Huebel, Herman Bruyninckx, Matteo Morelli, Chokri Mraidha, Sara Tucci, Marie-Luise Neitz, and Daniel Meyer-Delius. *RobMoSys D2.2: Initial preparation of (meta-)models, prototypical DSLs, tools and implementation*. Project Deliverable. June 2017.
- **Dennis Stampfer**, Alex Lotz, Vineet Nagrath, Christian Schlegel, Mathias Lüdtke, Björn Kahl, Sebastian Friedl, Christian von Arnim, Johannes Baumgartl, and Simon Krais. *SeRoNet Deliverable D4.1 "Leitfaden"*. Project Deliverable. Aug. 2017.
- Various contributions to the RobMoSys ("Composable Models and Software for Robotics", an innovation action in the European Horizon 2020 research and innovation programme) approach and composition structures as published on http://www.robmosys.eu/wiki/.

2
Related Work

This chapter describes selected existing methods and approaches that are related to software development and system composition in a robotics software business ecosystem. We will come back to some of them in later chapters and consider them for the approach.

2.1 Software Business Ecosystems

The probably earliest description of a "business ecosystem" ("ecosystem" in short) was introduced by Moore [Moo93]. He describes it as the behavior of a network of tech-companies that cross a variety of industries where they co-evolve capabilities around new innovations by working cooperatively and competitively. A business ecosystem is often referred to as a collaboration model (cf. [BB10; IL04]), which describes the many ways and advantages in which experts in various fields or companies collaborate around a domain or product. Ecosystems can be based on a keystone-player [IL04] (e.g. the Apple ecosystem) or a complete domain or community [Jan12] (e.g. the Debian Operating System Community). The research of Bosch and Jansen describe an ecosystem approach as the next step of a software platform to open up and to extend [BB10; Jan12]. Peltoniemi and Vuori [PV05] merged and refined various definitions of an ecosystem. A generally applicable definition is:

> *A business ecosystem is "a dynamic structure which consists of an interconnected population of organisations. (...) A business ecosystem develops through self-organisation, emergence and co-evolution, which help it to acquire adaptability. In a business ecosystem, there is both competition and cooperation present simultaneously." [PV05]*

The work of *Bosch* [BB10; Bos09; Cap+14] addresses both the software engineering and the business perspective of ecosystems. He introduces the close relation between ecosystems and composition. Software engineering shifts from tightly coupled to loosely coupled interaction; Bosch argues in a similar way for software ecosystems and motivates loosely coupled and decentralized approaches for software engineering. Ecosystems support in shifting from an integration-centric to a composition-oriented approach [BB10]. An ecosystem, thus, provides the collaboration model that is needed when shifting from traditional value-chains to modern networks [MH13]. Bosch argues that an ecosystem approach is the next logical step after establishing a Software Product Line (SPL) [Bos09]. *Oster and Wade* [OW13] analyze software ecosystems with respect to composability in the context of the US defense industry in comparison to other domains such as electronics, computer, and automotive. They distinguish two kinds of ecosystems. Organic ecosystems are driven by market pressure and are guided by very loose rules. Planned ecosystems are carefully organized, their evolution is managed by some organization (e.g. Microsoft, Apple, Facebook). *Iansiti and Levien* [IL04] describe platform aspects of ecosystems around one driving player ("keystone-player"). *Peltoniemi and Vuori* [PV05] provide an overview and summary of business ecosystems research in the biological, industrial, economical, digital business and social fields. *Manikas and Hansen* [MH13] give a more recent overview on ecosystems from a software engineering perspective.

This thesis envisions the composition of building blocks that are shared and exchanged between participants in a robotics software business ecosystem. The kind of loosely coupled interaction as introduced by Bosch is the fundamental way of collaboration in an ecosystem. This kind of collaboration is not yet established or feasible in robotics: Robotics lacks the underlying structures that enable this kind of collaboration. This thesis proposes structures for service-based composition of software components. It thereby contributes in proposing such structures to close the gap between the vision of ecosystems and the state of practice in robotics. The remainder of this section describes the typical aspects of an ecosystem by the examples Eclipse, Debian GNU/Linux, and the Smartphone ecosystems for iOS and Android. Chapter 3 describes the vision and the needs of an ecosystem for robotics.

A well-known example of an ecosystem is the *Eclipse ecosystem*. There is, for example, a big community that gathers around the Eclipse Integrated Development Environment (IDE). It provides technical structures, for example the powerful plugin mechanism. These structures organize and enable the independent contributions and collaboration from different people, organizations, and domains. Most of the contributions in the Eclipse ecosystem use the Eclipse platform to build their own domain-specific solutions that then become part of the Eclipse ecosystem. For example, the Eclipse CDT for C++ development, Papyrus UML for modeling, and a whole collection of MDSD tooling in the Eclipse modeling project. The structures that enable such a huge ecosystem are yet missing in robotics. The basic organization of an ecosystem that is proposed in this thesis (composition tiers, Fig. 1.7 and section 4.1.1) can also be mapped to the Eclipse ecosystem. There is a core that develops and maintains the core structures of the Eclipse ecosystem composition Tier 1 (e.g. the Eclipse IDE). Based on this, there is a larger group of people and organizations that build upon these structures and develop tools and applications for specific purposes or domains at composition Tier 2. Finally, there is a huge group of "users" at composition Tier 3. They use the tools for their purpose and exchange development artifacts with others. In the end, ecosystem participants have different motivations to use or contribute, but all can benefit from the ecosystem. This also holds true for making business. Eclipse is a successful example which shows that open source, ecosystem, and community are not to be equated with hobbyists. Eclipse calls itself business-friendly and open source since there is an open platform that provides the core structures for ecosystem collaboration. They enable competition on products and services. Commercial examples, for example, are the SymTA/S tooling for timing analysis and the WindRiver VxWorks tooling suite.

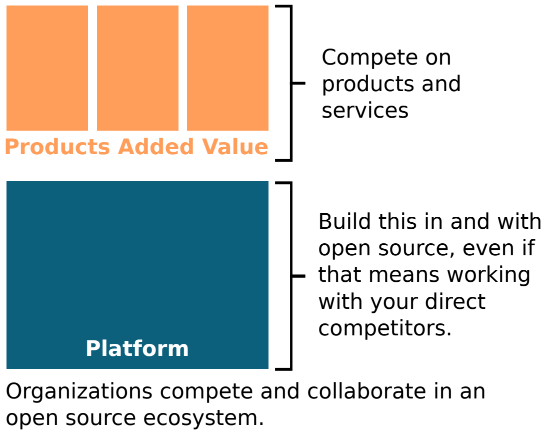

Products Added Value
Compete on products and services
Platform
Build this in and with open source, even if that means working with your direct competitors.
Organizations compete and collaborate in an open source ecosystem.

Figure 2.1: The eclipse ecosystem: The platform provides structures that enable collaboration but also competition at the level of products and services (from [Eclf]).

The ecosystem around the open source operating system *Debian GNU/Linux* is another good example of a successful ecosystem. Debian provides *technical structures* (and infrastructure) for package maintenance and organization. Hundreds of maintainers care for third party software packages, integrate them to the Debian packaging structure and keep them aligned. Debian is a good example of how a set of stable structures enable the loosely coupled collaboration of thousands of developers. The latter not only holds true for Debian, but also for all the other OS distributions that forked from Debian. Apart from technical structures, Debian is a good example of how an ecosystem may be run with respect to *organizational structures*. Debian is driven by a community (community-driven ecosystem, see [Jan12]), but uses clear organizational structures. There are elected persons in key-roles and committees, there are democratic ways of taking major decisions. Debian provides technical core structures and technical infrastructure that participants in the ecosystem use to contribute to and to benefit from. Debian's technical structures are open to use but with clear and democratic processes that decide how they evolve.

The *smartphone domain* probably runs the largest and most successful software ecosystems. *Apple* and *Google* as the two big keystone-player drive their technical and business platforms around their "market places" OS App Store and Google Play/Android Market. These markets contain self-standing and usable applications that mostly target for the end-user/consumer. Technical platforms and their structures for the smartphone domains are well established and supported, but are driven and controlled by its vendors (including the vendor lock-in). Robotics can learn from the smartphone ecosystems as an interesting example of a business model behind a platform and collaboration in an ecosystem with a benefit for all participants. Compared to the smartphone domain, the complexity to develop robotics systems is much higher because robots operate in the real world. The contributions made in this thesis with respect to system composition were also demonstrated in the smartphone domain within the FIONA research project [Fiona]. It showed that the concepts required for robotics system composition are also applicable and of benefit for the smartphone domain.

2.2 Software Engineering and Integration Approaches

2.2.1 Software Development Processes and Collaborative Development

Software development processes structure the collaboration of stakeholders that are involved in designing and implementing software. A whole range of approaches have already been developed. For example, sequential approaches such as the *waterfall-* or *V-model*, iterative approaches such as the *Unified Process* and its variants, and agile approaches such as *Scrum*. There are also activities that consider the applicability of software development processes to robotics (e.g. [Sal+17]).

Software development processes can be considered "management approaches" since they manage how software is being built. In line with Bosch [Bos09], this thesis argues that management approaches do not scale towards an ecosystem. Collaboration by structure is required rather than collaboration by management (section 3.2). This thesis proposes such a structure for robotics based on service-level composition of software components.

In the context of this thesis, software development processes can be used where management is possible. For example, by the individual roles or within the steps of the composition workflow: by the component supplier or system builder. However, there is a need to organize the handover between these roles by a superordinate structure such that two developers covering that role do not need to negotiate bilaterally. This is detailed in this thesis.

Tools for *collaborative software development* and also cloud-technologies for the same purpose have been offered and applied widely in the recent years. For example, revision control (e.g. *Subversion* and *Git*), (bug-)trackers (e.g. *Trac* and *Bugzilla*) and *Dropbox*. There is even collaborative code-editing available in the "Google-Docs-Way" [CodeCloud] and collaborative modeling in the Eclipse CDO Model Repository [Ecla]. A web-based integration platform for collaborative and distributed development of robotics systems with an according software development process is presented by Reiser [Rei14]. ROSMOD is a model-driven tool for ROS development that also supports collaborative "team" code development and collaborative model editing among developers for a particular robot system [Kum+16]. Both address the support of individual roles in a joint "team" development process for robotic systems. They contribute to separation of roles within a team of developers. For example, it enhances collaboration from different locations to support the integration of a robot system which is only physically available in one location.

Approaches for collaborative development enhance the collaboration and productivity in joint, team-based developments with close interaction of developers. However, such approaches must be applied in ecosystems with caution as it may contradict with the need for separation of roles (see section 3.3.2). Roles not only work in distributed in space (different locations), but also work distributed with respect to time. Structures that organize the interaction and separation of roles via managing the interfaces between roles are mandatory to be established in collaborative software development. Only then are they ready to be applied for ecosystem users. Once the clear separation of roles is established, these approaches will contribute to ecosystem collaboration. For example, via cloud-based IDEs where every ecosystem participant can use the same up-to-date version and easily share models or workspaces thanks to cloud-storage without version-conflicts (e.g. Eclipse Che [Eclb]).

2.2.2 Software Product Lines

Software Product Lines (SPLs) [Nor08; BCK12] are an approach for systematic reuse and management of similar products in a product family. They are extensively used in automotive industry [OW13], but are adopted by almost any other domain [Nor08]. A software product line is the collection of similar products that share a common and managed set of features [Nor08]. They are used to manage possible variants of product families. They are an approach for systematic reuse within the SPL [Nor08]. All products share a common (reference) architecture [BHA12; NOB11; Nor08]. Commonalities are typically managed and modeled by feature models (typically trees). Feature models capture the possible variants and valid combinations of a product. All valid variants must be designed into the SPL in advance. SPLs are not suitable for open-ended product variants [Voe13]: they are not flexible.

SPLs focus on *intra*-organizational reuse of components [HHJ08; RMM08; BB10] and they yet must evolve to the *inter*-organizational level as recognized in the ITEA Roadmap for Software-Intensive Systems and Services [ITE09]. Even though a SPL may span over organizations, the specific SPL must be established with the involvement of the organizations over which the SPL spans. An ecosystem approach is different, since it defines the core structures that enable interaction and collaboration without additional management and negotiation of the SPL. When spanning a SPL over organizations or applying it within an organization, reuse then is still made within the established SPL [HHJ08]. The products are thus limited to the SPL's specific architecture or feature structure. The characterizing features and the variability of a SPL depends on the product scope and intended variants which might be totally different for someone else. A SPL can be compared to configuration: A SPL is not as flexible and not as free in combination as is a compositional approach. Compositional approaches are the next step after establishing a SPL [Jan12; Bos09].

Most SPLs focus on functional features and model a set of alternatives that can be chosen (e.g. localization can be laser-based or visual); more advanced research also proposes to use non-functional properties in context of a SPL [BTR05] to cover measurable properties with attributes (e.g. the maximum speed that a robot is capable to drive).

The decision on which features are in or out of a product is made at design-time. That means, the feature selection cannot change once the software is deployed. Dynamic Software Product Lines (DSPLs) extend SPLs and defer the decision on features to run-time [Cap+14; BHA12]. This is useful to "dynamically" react and adopt the product to the current context at run-time. This means that a (sub)family of a product must be deployed—parts of the SPL infrastructure must be deployed as well. While there is a focus on run-time adaptation, DSPLs are, generally spoken, a way to cope with adaptation of products that originate from third party (e.g. from a component market place). For example, a component that was provided by someone else might be downloaded, composed into a robot system and configured via a DSPL.

A SPL is not in contrast to Component-Based Software Engineering (CBSE) or a Service-Oriented Architecture (SOA), as both are approaches to realize systems that are modeled in a SPL [Nor08; Cap+14; Omm02; HHJ08]. An overview on applying SOA in the context of a SPL is provided in [Cap+14], further examples can be found in [SA08] for SOAs and [Omm02] for CBSE. Model-driven techniques are quite useful to manage SPLs [BHA12; Nor08; GB14], for example to use Domain-Specific Languages (DSLs) to model feature models [Voe13]. An example of a model-driven approach for a SPL for robotics is presented in [GB11; GB14] for architecture modeling and in [Bau+13] for modeling of manipulation and grasping.

SPLs can be applied in the context of this work in the same way as any other development process, as long as the SPL remains limited for a particular role or step in the composition workflow. Component suppliers, for example, can apply a SPL to manage variants of components, thereby reducing their effort to come up with a bunch of components with individual features that are all build upon the same baseline. System builders can use a SPL to manage variants of the robot systems they build. For the envisioned ecosystem, it is not possible to span a SPL over all roles and the whole workflow as the SPL bases on a concrete architecture and on explicated product variants. It is not possible to foresee and define the variants of robot systems that are supposed to be built in an ecosystem. This is only possible within a narrow scope of an ecosystem; an ecosystem participant that covers a certain role is such a narrow scope.

We will come back to SPLs in section 3.4.1 when considering them for collaboration by structure and in section 3.4.5 when considering them for configuration of a building block.

2.2.3 Component-Based Software Engineering

Component-Based Software Engineering (CBSE) is an approach to structure software systems [Szy02; Bro+98; CSS11; Frö02]. It applies separation of concerns and identifies dedicated functional blocks with specified interfaces in order to achieve encapsulation between components. It enables flexible, off-the-shelf reuse of software components. In CBSE, components shape the architecture of a system [HKF08]. In this thesis, the main benefit of components is seen in them being suitable as units of composition and exchange in the ecosystem where components come with SOA services. The granularity of a component (e.g. a component with three services vs. three components with one service each) is not the factor that shapes the architecture, but the services are. Service-oriented components enable service-based composition of software components (see section 3.4.1).

A software component is defined through a component model as a generalization of concepts of a component. A component model that provides suitable abstractions in a stable structure is mandatory to apply Model-Driven Software Development (MDSD). Several general-purpose component approaches exist such as *Enterprise JavaBeans (EJB)* and as the component model of the *Common Object Request Broker Architecture (CORBA)*. Since they base on *freedom of choice*, they are well suited for robotics application development, but cannot serve as a structure for system composition. The *Unified Component Model (UCM)* [OMG13] is an initiative by the Object Management Group (OMG) to build a next-generation and state of the art component model for distributed, real-time and embedded systems. Since *freedom from choice* seems to be the design principle, the developments behind UCM are worth considering for robotics, once it is more advanced. It mentions *SmartSoft* [Sch04a] (section 2.8) as one of the potential approaches that may shape the UCM; this underlines the relevance of SmartSoft as a component model not only tied to the SmartSoft framework.

CBSE can be considered as state of the art in robotics software development. An overview is given by Brugali and Scandurra [BS09] and Brugali and Shakhimardanov [BS10]. Even if software is developed in form of components, it is often developed without an explicit component model (e.g. ROS). While there are robotics-specific component models (e.g. Rock [JA11], RTC [OMG12a], RobotML [Dho+12]), they are either shaped towards their single target framework or they are the result of reducing the component model to the least common denominator of the target frameworks. The latter then results in losing coverage of relevant aspects (e.g. BCM [Bru+13]). Providing a component model is mandatory for system composition. The existing robotics component models, however, improve the development within the particular supporting frameworks. They do not address the overall ecosystem vision in terms of the workflow. Instead of focusing on a particular framework and making models its base, the focus should be put on the way how robots should be built in such an overall ecosystem in general. The result will be structures that then, expressed as (meta-)models, can serve as basis to come up with transformations into the specific robotics frameworks. This thesis contributes such a model-driven structure and workflow for service-based composition of software components. It realizes the mappings to the SmartSoft Framework.

2.2.4 Service-Oriented Architecture

A *Service-Oriented Architecture (SOA)* [Erl08; SW04] is often seen as a synonym for technical realizations of remote software (e.g. WSDL, SOAP, REST) [Man09], but it is much more than that. It is a way of thinking about architectures, loose coupling, abstractions, and reuse. A definition is given by Sprott and Wilkes [SW04]:

> *A service-oriented architecture is "the policies, practices, frameworks that enable application functionality to be provided and consumed as sets of services published at a granularity relevant to the service consumer. Services can be invoked, published and discovered, and are abstracted away from the implementation using a single, standards-based form of interface." [SW04]*

SOAs are about entities that provide or require activities or data through services. These services have a certain value to the consumer at a granularity and abstraction that is adequate and reusable for someone else. Services should be self-contained, loosely coupled and usable as black-boxes with known interfaces that is more than an Application Programming Interface (API).

A SOA puts a focus on the flexible combination of a multitude of services to provide the expected functionality of an application. A service-oriented approach is considered mandatory for system composition as is also argued in the ITEA Roadmap for Software-Intensive Systems and Services [ITE09]. Service-oriented thinking helps to come up with structures for composition supporting separation of roles for robotics.

An example realization of SOA is *Representational State Transfer (REST)*, a style to transfer messages via HTTP using various formats (e.g. JSON, XML). *Simple Object Access Protocol (SOAP)* [W3C07a] is a standard and technical realization of SOA for exchange of structured messages via HTTP.

An important element for the technical realization of SOAs is the definition of a service and publishing this information. The *Web Services Description Language (WSDL)* [W3C07b] is a well-known Interface Definition Language (IDL) for web services to describe how clients interact with a (remote) service. It focuses on the syntactic expression of an interface and does not consider its semantics and (non-functional) properties. This is addressed by the *Web Ontology Language for Web Services (OWL-S)* [Mar+04], an ontology-approach for web services. It enables the semantic description of web services. In the context of this thesis, ontologies can be used to describe the semantic relations between individual models of the composition workflow, e.g. service descriptions, to enhance composability (see section 2.3.2). Thus, OWL-S may be suitable to organize the domain structures.

The *Service Oriented Architecture Modeling Language (SoaML)* [OMG12b] is an extension (profile) to the Unified Modeling Language (UML) to design service-oriented architectures. It focuses on the reusable modeling of services. Like interfaces in UML, it models a fine-grained API-style definition and use of interfaces. In extension to UML interfaces, SoaML allows to describe the collaboration of services through "protocols" to define how multiple operations are used within a service interface [Ams12]. Service "contracts" describe the interaction of multiple services independent of a particular provider or consumer. SoaML is targeted at *holistic modeling* (one big model captures everything) that is different from *distributed modeling and composition* supporting separation of roles as addressed in this thesis.

Most robotics approaches apply SOA as technical realization of remote software, i.e. by using SOA implementations for message exchange. For example, making use of existing (external) web services for robotics [Bru+14; MBF11], i.e. to connect robots to the web. Koubaa [Kou15] implements web services for ROS to "expose ROS resources as SOAP and REST Web services". They do not consider SOA as a philosophy and principle to enhance the development and architecture of a robot system. SmartSoft is one of the outstanding examples that does so. SmartSoft (section 2.8) uses service-oriented software components for software development in robotics. This thesis widens its scope to apply it in an ecosystem for system composition by introducing structures that are required therefore. The need of SOA and CBSE in relation to this thesis is described in more detail in section 3.4.1.

2.3 System Design and Modeling

2.3.1 General-Purpose Modeling Languages

The *Unified Modeling Language (UML)* [OMG15b] is a well-known general-purpose modeling language in computer science and software development. Thanks to profiling, it allows for domain-specific extensions and adaptations. There are, for example, profiles to UML for modeling service-oriented architectures: The *Service Oriented Architecture Modeling Language (SoaML)* [OMG12b] focuses on service modeling. The work of Wada et al. [WSO06] models non-functional properties for SOA communication. Other notable profiles with relation to this work are the *Systems Modeling Language (SysML)* [OMG15a] to provide a general-purpose language for systems engineering and *RobotML* [Dho+12] to provide a modeling language for robotics. *Papyrus* [Papyrus] is one of the most advanced open source graphical modelers that also supports UML in the Eclipse Project. This thesis utilizes UML profiles and Papyrus to implement the SmartMDSD Toolchain (see chapter 6). The composition structures, however, are independent of any specific implementation. For a discussion to use UML for composition structures, see section 3.4.1.

The *Object Constraint Language (OCL)* [OMG14] is included in UML [OMG15b] to specify expressions and conditions on UML models to ensure their consistency more as it can be expressed in UML. It enables to describe constraints about the models which cannot be done by UML. It is not intended for structural modeling and expressing or constraining relations as this is covered by UML. It addresses functional aspects and is not intended for architectural use. In context of the presented work, OCL may be applied within dedicated models, in a communication data structure or parameter set to ensure their consistency (for possible uses of OCL therein, see sections 3.4.3, 5.2.2 and 5.2.5).

Capella is an Eclipse-based and model-based engineering tooling for graphical modeling to support architecture design with a focus on architecture viewpoints. It is of interest to this thesis because of two similarities. (i) The Capella-approach is inspired by UML/SysML [Bon+16] and (ii) the position of the Capella-approach with respect to building tooling via profiling is very similar. As within this thesis, the Capella-approach does not strictly rely on the general-purpose modeling standard, but uses it where appropriate. The authors argue [Bon+16] that applying UML/SysML structures are hard to use for domain-specific modeling by users that do not have UML/SysML background and that tooling based on profiles alone often becomes unnecessarily complex. Capella directly adopts the structures and elements from UML/SysML that are suitable, extends the ones that are reasonably close, leaves the ones out that are not adequate, and creates new ones where they are missing.

The *Architecture Analysis and Design Language (AADL)* [FGH06] is a modeling language for hardware and software architectures of safety-critical embedded and real-time systems. AADL is used to model systems in a holistic and single, "all-in-one" model at a rather low level of abstraction and very close to the later implementation (see also [Del+08]). It exposes all parts of the system with no guiding structures: They support freedom of choice. While this level of modeling is adequate to do in-depth analysis (e.g. timing, network-latency) as needed in AADL's safety-critical domain, this hinders reuse, separation of roles and composition. Many community-driven or commercial tools are available for textual and graphical modeling, as well as analysis and code-generation for AADL. For example, the model-driven tool *Osate*. It is based on Eclipse and Xtext.

The above modeling languages are suitable to model and develop custom robotics applications. To apply them for system composition in an ecosystem, they provide way too less guidance as they offer freedom of choice (section 3.3.1). If using these general-purpose languages for ecosystem collaboration, it requires additional structures such as presented in this thesis. The above modeling languages can be used to model systems that support separation of roles during development, but this depends on discipline and is not enforced by structure in the modeling language itself. It can be done rather by policies that the users follow while using it. Even the robotics-specific approaches such as RobotML foster the all-in-one model, not supporting separation of roles to enable composition of building blocks that were supplied by different participants in an ecosystem. They do not base on service-level granularity as required for system composition. RobotML and BRICS contributed component models, but component models alone are not sufficient to address system composition in an ecosystem. Modeling services, for example via SoaML alone is not sufficient, too. It needs careful considerations of component models, services, and other aspects with respect to the overall workflow and involved roles. These considerations influence the superordinate structures. The structures must include a service-oriented component model and particular aspects of the mentioned approaches might be worth to be included as long as they contribute to the overall vision of composition.

2.3.2 Ontologies

According to Gruber [Gru09], "an *ontology* defines (specifies) the concepts, relationships, and other distinctions that are relevant for modeling a domain. The specification takes the form of the definitions of representational vocabulary (classes, relations, and so forth), which provide meanings for the vocabulary and formal constraints on its coherent use" [Gru09]. Ontologies express the knowledge of a domain and can be queried. Based on an ontology, new knowledge that was not explicitly expressed, can be reasoned [Gru09]. With the use of ontologies, one can build common definitions and use them in a distributed manner: The semantic web, to name one prominent example, uses the *Web Ontology Language (OWL)* [W3C12] to describe and process the information that is provided by websites. The *Web Ontology Language for Web Services (OWL-S)* [Mar+04] is a DSL on OWL [W3C12] to describe web services.

In the context of the approach that is presented in this thesis, ontologies can be considered for use in three directions. Ontologies can be considered for (i) modeling the composition structures at composition Tier 1 (section 3.4.1) as an improvement of meta-modeling since ontologies support integrated semantics and consistency checks. Ontologies can be considered to (ii) improve composability, for example to manage the interfaces between components as expressed in domain structures at composition Tier 2 (see section 3.4.1). Since ontologies are descriptive in their nature [AZ06], they can express relations between domain structures and eventually describe the differences. Finally, ontologies (iii) can enhance the matchmaking mechanisms for component selection that is needed at composition Tier 3 thanks to classification and reasoning (see section 3.4.4). The remaining section first reports on the use of ontologies for software development in general and then reports on the use of ontologies in robotics. We will come back to ontologies to address each of the above considerations in sections 3.4.1 and 3.4.4.

Ontologies can be used for modeling in software development in general. The European FP7 project *MOST (Marrying Ontology and Software Technology)* has shown how to use synergies between ontologies and software development and researched the integration of ontology technologies and meta-modeling/Model-Driven Software Development (MDSD) as *Ontology-Driven Software Development*. The *TwoUse Toolkit* [Sta+10] is a general-purpose Eclipse-based implementation that bridges between UML, OWL, and Ecore. Aßmann and Zschaler [AZ06] describe the benefit of "ontology-aware meta-modeling".

Ontologies are descriptive models, while meta-modeling enables prescriptive/specification models [AZ06]. Both provide complementary expressiveness to meta-modeling and combining them improves software modeling [Sil+10; AZ06]. Semantics and constraints, for example, are not part of meta-models and only exist implicitly in tools or natural language descriptions. Using ontologies, one can express semantic restrictions and constraints in the same language in one place [Sta+10; Mik+13]. Ontologies can simplify consistency checks of models that cannot be covered by meta-models. While this can partially be addressed by the Object Constraint Language (OCL) [OMG14], the use of ontology technology allows to use more advanced features such as expressing semantic constraints, classification, and reasoning abilities.

Most use of ontologies in robotics is made at *run-time* to represent, share, and interpret knowledge for decision making [ZAF16]. The *Knowledge Processing for Autonomous Personal Robots (KnowRob)* initiative and related EU research projects RoboEarth and RoboHow are good examples. They collect knowledge of household objects and everyday manipulation tasks to share them among robots [TB09]. An overview on the use of ontologies in robotics was given by the *IEEE RAS Ontologies for Robotics and Automation Working Group* [Pau+12]; a more recent survey for knowledge representation in robotics can be found in [ZAF16].

There are few use-cases for ontologies at *design-time* for engineering of robotics software. One notable approach is the *ReApp* project (see section 2.6.3). It applies ontologies for MDSD in robotics to simplify the reuse of Robot Operating System (ROS) components and means to discover them [Zan+15]. The ReApp Workbench interfaces between OWL ontologies and Ecore models [Wen+16]. ReApp uses the classification mechanism of ontologies to search components that provide a certain functionality or capability. This supports system integrators in finding a component with a certain capability. However, providing a certain capability does not yet mean that the component that provides this capability is composable with others. As a result, adapters and integration might be required.

2.4 Model-Driven Software Development

Model-Driven Software Development (MDSD) is an approach to software development that puts a focus on modeling certain aspects of software to gain an abstract representation. That is, models follow a specific purpose and capture the relevant details of software which are important for this purpose. Thanks to the explication and accessibility of these relevant details, MDSD supports in lowering the complexity of the system being developed [BCW12]. The purpose and value of using MDSD has long been seen in consistent documentation and in lowering the effort through code-generation. But MDSD is much more than that.

MDSD is becoming a matured technology and is seen as the next evolutional step in software engineering [Bar16]. MDSD has been identified as a key technique for robotics "in order to achieve a separation of roles in the robotics domain while also improving composability, system integration and addressing non-functional properties" as stated in the "Robotics 2020 Multi-Annual Roadmap (MAR)" [euR16]. It is also seen as "challenge to be addressed in future software engineering" [ITE09, p. 279] in general as stated in the "ITEA Roadmap for Software-Intensive Systems and Services".

MDSD brings many benefits for providing an approach for system composition and for using it. Its benefits in relation to this thesis are:

- MDSD significantly increases productivity, quality and lowers development efforts, thus saves time and money [Völ11]. For example, thanks to processible models for model-checking, model transformation and expert knowledge encoded in code-generators.

- MDSD brings explicated structure into the approach. Designing meta-models to apply MDSD means to think about the elements and properties that are necessary to be represented. Using the meta-models then means using only these elements for modeling, enabling well-thought and explicated freedom from choice (instead of freedom of choice).

- MDSD makes the solutions that are covered by meta-models accessible to the user. The meta-models abstract away deep technical details and do not require the user to understand their full details. They are thus simpler to understand [Völ11]. As a result, they raise the discussion about models for composition to an appropriate structural level, keeping it away from unnecessary low-level details: discuss with others only what is relevant to others. This helps in collaboration of roles.

- Meta-model abstractions allow for creating specific views or models that separate the overall problem into parts of specific concerns for specific roles. MDSD therefore helps to support separation of roles and separation of concerns for system composition [euR16]. MDSD allows to separate the domain knowledge (the structure and architecture) from its implementation. For example, the structure of the solution is modeled independently from the particular technical implementation of these structures. MDSD allows for separation of the different steps and roles in the workflow (e.g. supplying and using a component) and between stakeholders (e.g. component developer A and B) to manage their contributions to bring them together consistently to an application.

The concept of a *Domain-Specific Language (DSL)* is of particular importance to realize MDSD tooling. A DSL is "a computer programming language of limited expressiveness focused on a particular domain" [Fow11]. Compared to MDSD as a general approach, a DSL is a "specific" way to describe and use a model that is motivated by a certain problem or domain; it is tailored towards the user who is using it [Voe13, p. 31f.]. The use of DSLs simplifies providing adequate graphical or textual modeling to support ecosystem participants. Using a DSL provides a way of modeling that is tailored to the role or problem in a task or domain. In contrast, using general-purpose modeling tools such as UML, is not always the best choice for every modeling-task (see [SG99; BS02; Bon+16]). General-purpose modeling is often too heavy and too general to use, allowing too much freedom of choice.

By using MDSD, users can directly gain advantage from expert knowledge that is encoded in generators. Thanks to DSLs, the way to create and work with models can be adjusted to the task at hand. This can include, for example, vocabularies or representations that are typically used in the target domain and thus help to understand and work with the model. There is no cryptic syntax that is usually introduced when using general-purpose programming languages.

There are powerful environments available to apply MDSD and to develop DSLs. Very well-known is the tool collection of the *Eclipse Modeling Project* [Eclc] with *Ecore* [Ste+08] and *Ecore-Tools* [Ecle] for meta-modeling, *Sirius* [Eclg] for graphical modeling, *Papyrus* [Papyrus] for UML and general graphical modeling, *Xtext* [Ecli] for DSL engineering, and *Xtend* [Eclh] for code-generation. The *JetBrains Meta Programming System (MPS)* [Jet16] and *MontiCore Language Workbench* [Sof] are other notable approaches for MDSD-suites. The existing MDSD-suites are now matured enough to apply them to build productive tools and to realize the structures presented in this thesis.

Several industry-grade domain-specific model-driven tools exist. For example the *Yakindu State Chart Tool (SCT)* [ite] for modeling and implementing state-chart-based applications. The MPS-powered *mbeddr* [Voe+13] is an example for a collection of integrated languages and is used for embedded systems development.

2.5 Domain-Specific Approaches

Each domain has different requirements and many domains have established specific approaches to software development, for example, the automotive, avionics, and smartphone domains. Before going into detail on software development in robotics, we briefly strive software development in the automotive, smartphone, and industry 4.0 domains.

2.5.1 Automotive Industry

The *automotive industry* is an interesting domain for system composition. It has a high demand in software development and it has established an ecosystem with its suppliers. The *Automotive Open System Architecture (AUTOSAR)* [Autosar] is one example for software engineering from the automotive domain. AUTOSAR is an established and structured engineering approach in a worldwide effort of car manufacturers, suppliers, and other companies in the electronics and software industry. AUTOSAR uses a model-driven approach and aims at commercial off-the-shelf components. Heinecke, Rudorfer, Hoser, et al. [Hei+08] present an Eclipse-based MDSD-tooling for AUTOSAR.

AUTOSAR builds on standards that include a formal component model. Even though the standard is large, it does not define all relevant aspects and leaves room for interpretation. This ends up in components that are not composable due to inconsistencies, even though they follow the standard (e.g. "complex drivers" as universal tool, no standardized message exchange). On the other hand, AUTOSAR standardizes an architecture that tends to be restrictive when people want to apply another architecture. AUTOSAR on one hand underlines the need for structures as basis for collaboration. On the other hand, it is a negative example that underlines the need for carefully balancing these structures such that they neither restrict too much nor provide too much freedom. Even though there is a need in automotive for composition of components to keep pace with technology development and market expectations, flexible composition is not possible. In turn, there is a need for deep knowledge and for mechanisms to manage non-functional properties that enable safety analysis and predictability. Thanks to "all-in" models with access to all levels of detail, automotive is very advanced in these areas. The price to pay is low support for separation of roles and composability.

2.5.2 Smartphone Domain

The *smartphone domain* is of interest to this thesis because of its successful ecosystems and market places (see section 2.1). The complexity of applications for smartphones is rather low in comparison to robotics software systems. Many important issues are covered by the platform providers (e.g. Apple and Google) which then only expose limited complexity to app developers, for example with respect to hardware and platform. The number of available hardware in the smartphone domain is huge, but it is still rather low compared to the variety of hardware (and also software) in robotics: All smartphones have a touchscreen, microphone, GPS, and wireless on board whereas the variety of different sensors and actuators is much higher among robots. There are no component models in widespread use and tool support focuses on rapid

GUI development and on abstraction between platforms to overcome vendor lock-in (write once, deploy to Android, iOS, etc.). Reuse in the smartphone domain is made on the library- and code-level (e.g. *Xamarin component store* [Xama]). An overview on open source frameworks for app development is given in [Wyl12]. When looking at the success of smartphones and the number of available apps, the way of dealing with complexity with libraries seems sufficient. This kind of source-level and library-level collaboration, however, will not scale for a robotics ecosystem. The smartphone domain can benefit from a composition-oriented approach, as has been demonstrated in the FIONA research project [Fiona] (see sections 7.3.2 and 7.5).

2.5.3 Industrial Automation and Industry 4.0

The challenges in *industrial automation* are raising with today's needs for flexible adaptation of the production flow, intra-logistics, human-robot collaboration, and robot co-workers. Traditional industrial automation is on the move to *Industry 4.0*. The *OPC Unified Architecture (OPC-UA)* [MLD09] is a promising approach for these domains as there is huge commitment and drive of machine and device suppliers towards being conformant to OPC-UA. OPC-UA is a communication infrastructure for machine-to-machine communication. OPC-UA works in an object oriented fashion and each machine or device grants access to specific variables in the OPC-UA "address space". Clients can access these data via predefined services (read/write, subscribe, method calls).

OPC-UA aims at integration and interoperability. In comparison to other approaches that addressed this in the past (e.g. CORBA), OPC-UA foresees the concept of companion specifications. Companion specifications define information models (variables/data types, methods, events) as a standard for a particular domain. In the context of the composition tiers for ecosystem organization in this thesis, the general OPC-UA standard would be on Tier 1, individual companion specifications on Tier 2 and concrete OPC-UA servers and clients on Tier 3. Since OPC-UA has means to apply freedom from choice via companion specifications, it comes with the basic structures that are necessary for use within an ecosystem.

OPC-UA eases device integration thanks to an overall methodology (Tier 1) and domain-specific standards (composition Tier 2). Device suppliers now can adopt the Tier 2 standards and gain compatibility with users that expect these standards. OPC-UA, however, does not specifically aim for composition and is, in fact, less suitable for composition of software components. It misses adequate abstractions: Even though it claims to be service-oriented, it is still too much focused on object-orientation. Using OPC-UA soon leads to fine-grained interfaces and method calls as known from object-orientation. Integrating such systems results in spaghetti systems, breaking composability. It defines the information models and data access independently. When a data structure or variable is offered, the client is free to choose the way to access it. This separation hinders composability. Adding structures to OPC-UA to express data structures and ways to access them as a stable pair, would enhance composability via OPC-UA. This is the approach that is applied in this thesis via "service definitions".

2.6 Software Development in Robotics

Robotics is different from many other domains with respect to the broad diversity of systems that combine more heterogeneous fields than systems do in other domains. The need to interact with the unknown, open, and dynamic physical world and with humans brings additional complexity. This complexity exists in a way that is not present in other domains. But the solutions that address this complexity in robotics are very valuable to these domains. This does not allow to directly apply approaches established in other domains. It requires their adaptation or even establishment of new approaches that are specific to robotics.

2.6.1 Robotics Libraries

The *OpenCV* library [Its] and the *Point Cloud Library (PCL)* [RC11] for (depth-)image processing, *OpenRave* [Dia10] and *MoveIt* [SC] for manipulation planning and more general the *Mobile Robot Programming Toolkit (MRPT)* [Bla] are very successful robotics programming libraries that simplify applying established robotics algorithms for specific purposes. Their existence, active maintenance and support is a great boost in advancing robotics technology without reinventing the wheel. These and other libraries enhance reuse at the code-level and library-level, but system integration is still a challenge in robotics [euR13]. The named libraries are great candidates to easy functional development by integrating libraries in components by component suppliers. These developments then become composable when following the approach that is described in this thesis. To benefit from the power of composition on the functional level within component implementations, however, it again requires superordinate structures. These structures are the challenge that still needs to be addressed.

2.6.2 Robotics Frameworks

Player [GVH03] was aimed at supporting the implementation of device drivers. It turned out to be a very successful robotics framework since it allowed to reuse third party software for robotics; and because of the popular simulator *Stage*. *Orca* [Bro+07] is one of the early CBSE approaches to robotics that pushed the awareness and need for reuse for robotics, supporting API-style interface definition based on CORBA IDL and a graphical tool to wire components. The *Orocos Real-Time Toolkit (Orocos-RTT)* [OroRTT] is a framework supporting real-time capabilities for industrial robots and manipulation. It collects a basic set of typically used robotics applications in the Orocos Component Library. There is no dedicated IDE for Orocos, but some activities provide mappings and code-generation for Orocos, e.g. BRIDE [BRIa], oroGen (Rock [JA11]) and RobotML [Dho+12]. *Yet Another Robot Platform (YARP)* is a robotics middleware “for plumbing for robot software” [MFN06]. It is targeted for research and academia and mostly used with humanoid robots requiring fast and reactive control loops. *Fawkes* [Nie+10] is a component-based approach for robotics software development around a Lua-based engine for robotics behavior. It supports blackboard messaging components in a monolithic plugin architecture. *Rock* [JA11] is a more recent framework for component-based software development for robotics. It focuses on error detection and error handling for long-living systems.

Rock comes with the oroGen component development tool and its internal component model is based on Orocos-RTT. Rock offers drivers and modules in a library. *SmartSoft* [SW99b] in its original form is a framework to support modularized implementation of robotics systems. It uses components and introduced the concept of communication mechanisms ("communication patterns"). Even before Orca was introduced, it applied CBSE and the idea of what today is called Service-Oriented Architecture (SOA). SmartSoft nowadays is a holistic approach to software development in robotics: the "SmartSoft World" (section 2.8).

A still relevant overview on robotics frameworks and middlewares can be found in [ES12].

2.6.3 Development Approaches and Robotics Software Infrastructures

Robot Technology Component

Robot Technology Component (RTC) [OMG12a] is a standard by the OMG. Its CORBA-based reference implementation OpenRTM-aist / RT-Middleware [And+05] comes with an Eclipse IDE. RTC was one of the early initiatives to specify a component model in robotics and thereby pushed the awareness and need for structures in robotics. It only provides data-flow communication and does not clearly separate between the reference implementation and the component model. While RTC is suitable for MDSD, its Eclipse-based tooling is not state of the art anymore in MDSD technology. RTC in general is out of date compared to more recent approaches such as BRICS, RobotML, and SmartSoft for robotics or UCM in general.

Robot Modelling Language

The *Robot Modelling Language (RobotML)* [Dho+12] is an outcome of the French PROTEUS activity to provide a common platform for collaboration in robotics research and development in France. RobotML is a Domain-Specific Language (DSL) for robotics modeling and a profile to UML. It can be seen as abstraction of common ground of the ROS and Orocos platforms. It does not strictly separate between robotics modeling and its robotics target platforms. It thus uses and directly maps to the terminology and elements provided by these platforms. This direct mapping is a close coupling that hinders composition in the way it is envisioned in this thesis. PROTEUS foresees an online platform to exchange elements of modeling, for example, models at different abstraction levels and granularities but also algorithms or libraries. The distinction between providers and users is a very basic separation of roles in terms of exchange through a platform. It is not comparable to the separation of roles in this thesis, which is in terms of dedicated modeling perspectives and enabling experts to work loosely coupled. The model-driven tooling of RobotML uses Papyrus [Papyrus] and Eclipse [Eclc]. The overall approach is like the SmartSoft World and the SmartSoft MDSD Toolchain. The consortium of the ongoing EU H2020 RobMoSys [RobMoSys] Innovation Action includes developers of the SmartSoft World and RobotML. It is expected that both concepts, among others, join in an approach for an EU Digital Industrial Platform for Robotics (see section 2.7).

Robot Operating System

The *Robot Operating System (ROS)* [Qui+09] is, on one hand, a communication framework and a set of guidelines to implement robotics software, and on the other hand, a huge set of software libraries, tools, and downloadable software components (nodes). ROS is the most widely used robotics software platform to date and the availability of ROS definitely pushed robotics thanks to its "content" (availability of downloadable software packages and infrastructure to run them).

ROS lacks separation of concerns at several levels and it lacks structures to apply separation of roles. For example, there is no support for freedom from choice and design and implementation decisions are in the responsibility of the developer and integrator. The existing freedom and availability of modifiable source code is well appreciated by the community (e.g. [Del14]) but results in assumptions of components that only hold true for a certain environment. These assumptions must be understood—hopefully with good component documentation—before using the component in a new environment (see [Cou+10]). There is no explicit component (meta-)model for ROS which makes it hard to apply model-driven approaches. While ROS is very successful and in widespread use, these shortcomings hinder system composition. There is no dedicated IDE for ROS.

While ROS is mainly used by research for service robots, ROS-industrial [RosInd] aims to apply ROS in factory automation and manufacturing. In this context, the ongoing H2020 ROSIN project [Rosin] aims to improve the quality of existing ROS-Industrial components and tools using model-driven approaches, testing and code-checking to make it better and more business-friendly. The contributions of this thesis could be applied to ROS to improve software development and enabling system composition using ROS.

Robot Operating System Model-Driven Development

The *Robot Operating System Model-driven development tool suite (ROSMOD)* [Kum+16; Kum+15] is a relatively young approach. It provides a graphical model-driven component-based development tool to support rapid prototyping for ROS.

ROSMOD comes with its own component model that also addresses the support of analysis and verification [Kum+15]. In this respect, there is a high focus on task/thread level modeling in the component model. ROSMOD is an adequate solution to provide structure and tools for ROS. ROSMOD does not address composability and the modeled deployments and components are highly interweaved. This makes it hard to use it for separation of roles as required for a composition workflow in an ecosystem. Its web-based development tool and service-oriented design [Kum+16] of the tooling infrastructure enables collaborative "team" code development and collaborative model editing among developers for a particular robot system. Collaborative development easily gets in conflict with separation of roles (see section 2.2.1).

Best Practice in Robotics

Best Practice in Robotics (BRICS) was a European FP7 research project for component-based development of robotics software. It aimed at using MDSD to make existing robotics best practices usable. BRICS fostered separation of concerns for robotics.

The main outcome was the *BRICS Component Model (BCM)* [Bru+13], a robotics component model. The BCM aimed at providing an abstraction for common robotics frameworks. This led to a simplified component model that covers only the common structures. Thus, it does not allow modeling some of the import structures. The communication between components, for example, cannot be defined at the level of a component. In consequence, the communication can be modified at component implementation time, thus making it hard to reuse the component or use it for composition.

BRICS developed the *Integrated Development Environment (BRIDE)* [BRIa] as a model-driven tool for component creation applying the BCM. The *BRICS Open Code Repository (BROCRE)* [BRIb] is an online platform to search and install software. It is basically a convenient way to access and retrieve software. At the level of libraries and source code, it supports with package management and versioning. The method of choice to use the libraries seems "integration" rather than composition (see integration vs. composition, section 3.1.1) which makes it extremely hard to predict the behavior of components and to compose them.

ReApp

The recently completed *ReApp* project [ReApp] is a notable example of a complete environment for enhancing the reuse in robotics software development. ReApp and the approach presented in this work can complement each other. ReApp ontologies can be used to semantically describe and relate domain structures at composition Tier 2. The contributions of this thesis provide adequate structures to realize the underlying workflow that enables composability.

ReApp uses ontology technology at design-time for model-driven engineering to enhance reusability and discoverability of components (called "Apps") for the Robot Operating System (ROS). It basically enriches ROS with meta-data and tooling [Zan+15] and is an important benefit for the ROS world to give it more structure. ReApp provides means for semantic description of components via capabilities that describe the intended purpose of the software component (e.g. "image_processing"). Capabilities are modeled as enumerations [Awa+16]. The approach comes with the ReApp Workbench [Awa+16; Wen+16] and the ReApp Store [Bas+14]. Its underlying component model corresponds to the BRICS Component Model (BCM) [Bru+13]. ReApp models capabilities that are associated with interfaces and thus defines a collection of "component templates". Component suppliers model components by searching for and then selecting capabilities. This is supported by an ontology. The interfaces that are associated to the capability, i.e. the interfaces that the selected capability is expected to have, are automatically added to the component model (see [Zan+15]). System builders can discover components via a rough description of what the component will be used for. The ReApp Store will use the ontology and capabilities to propose adequate components that fit the criteria. The ontology classification mechanism allows to infere the capabilities of components based on their interfaces in case capabilities are not explicitly modeled. This is a typical ontology-based approach in contrast to a class-based modeling in which the capabilities of a component would be expressed explicitly. ReApp offers a very basic interface modeling, basically by referencing ROS .msg-files. The ReApp Workbench is so far described as a prototype for demonstrational purposes [Awa+16], no productive use has been reported.

ReApp is one notable example of a development suite for robotics to support the complete workflow from component modeling, over discovery, and system assembly to deployment. ReApp is a reasonable contribution to re-using, structuring, and classifying software in the robotics domain for new and non-expert users.

ReApp supports system integrators in finding a component with a certain capability. However, finding a certain capability (e.g. object recognition) does not yet mean that the found component is composable with other components that require/provide this capability. As a result, adapters and integration might be required. The semantic description of components, thanks to the descriptive nature of ontologies, is a strong part in ReApp and may be adopted on top of the approach presented in this thesis. It would help to semantically relate and organize domain-specific structures (see sections 2.3.2 and 3.4.1). But before doing so, one needs adequate underlying structures.

The underlying structures of ReApp are not sufficiently rich to be applied for system composition. For example, ReApp internally uses the BCM. It misses adequate structures in its component model as it is a generalization of too many frameworks (see section 2.6.3). ReApp builds on ROS and thus is structured in the ROS-way. Due to the freedom of choice philosophy, it misses clear structures that can be mapped to other approaches. Freedom of choice is needed for composability.

The parts describing components semantically are closely connected to the underlying component structures. Speaking in terms of composition tiers as presented in this thesis (Fig. 1.7 and section 4.1.1), Tiers 1 and Tiers 2 are combined. This leads to low flexibility of domain structures since they are part of the main composition structures. It also limits the eventual co-existence of domain structures that can contribute to finding a de-facto standard. ReApp builds on a tight coupling between capabilities and interfaces that explicitly define the input and output interfaces. This supports new users in finding adequate interfaces, but it ties the capability to the defined interfaces and requires foreseeing all possible combinations. As a result, this may lead to fine-grained capabilities and fine-grained modeling instead of hitting the sweet spot between freedom from choice and freedom of choice. This is a typical effect that often can be observed in ontologies. Providing a kind of templating mechanism and to assign a pair of interfaces to components also indirectly specifies parts of the architecture. It pre-defines the granularity of components and makes components the architectural unit. Speaking in terms of the composition tiers in this thesis, in ReApp this architectural definition of components happens at a high level and is thus too restrictive. An architecture should not be shaped by components, but by services that are provided/required (by components). Service definitions as the entities that shape the architecture, as presented in this thesis, do not restrict the architecture around them. Not restricting the architecture is important for composition, as the architecture cannot be foreseen when structuring a domain (composition Tier 2) or providing a building block (composition Tier 3).

The developers of SmartSoft and ReApp are partners in the SeRoNet [Bun17] activity and it is expected that SeRoNet reuses contributions from both approaches in a joint approach. This underlines that both approaches can complement each other.

SmartSoft

SmartSoft is a component-based and service-oriented approach for robotics software development [SW99b; Sch04a]. It early applied freedom from choice via providing only a small but sufficient set of communication patterns. The "SmartSoft Framework" originally was a set of concepts and guidelines to implement component-based systems that were supported by a reference implementation. The "SmartSoft World" nowadays provides a complete set of development methods, concepts, implementations and model-driven tools. This thesis builds on the "SmartSoft Framework" and contributes to the "SmartSoft World". Both is described in more detail in section 2.8.

2.6.4 Tooling in Robotics

This section focuses on tooling that supports the development of robot applications with respect to the overall workflow.

Most tooling dedicated to robotics probably exists in the context of ROS. Even though, there is no dedicated IDE for ROS that supports the design, development and integration of ROS nodes. The development environment mainly consists of dedicated tools for specific purposes (e.g. dedicated tools for visualization and a set of CMake- and shell-script based tools, cf. [Kum+16]). Using general-purpose IDEs (e.g. QtCreator) [RosIde] seems to be adequate for the ROS community, since the community is familiar with programming environments. There are some user-driven initiatives to come up with development environments for ROS. For example, *RIDE* [Ride] simplifies connecting and launching of ROS nodes. *rxDeveloper* [MHB12] is a graphical environment for parameterization of running ROS nodes. Other larger scale activities also contributed to tooling for ROS. The *BRICS Integrated Development Environment (BRIDE)* [BRIa] is a model-driven tool for component creation applying the BCM. BRICS heavily builds on ROS; BRIDE therefore mainly allows to develop ROS packages, nodes, and associated launch files. It also provided mappings to Orocos. As such, BRIDE contributes to the systems integration challenge for ROS. The *Hyperflex Toolchain* [GB11; GB14] is an extension of BRIDE that applies MDSD and the concept of SPLs to robotics and supports building and managing reference architectures using feature models. ReApp comes with the *ReApp Workbench* [Awa+16; Wen+16] (see also: section 2.6.3). It uses the OWL for ontologies and component modeling [Zan+15]. The underlying component model corresponds to the BRICS Component Model (BCM) [Bru+13] and the BRIDE [BRIa] environment is used for code-generation [Awa+16]. Component assembly and deployment is modeled in the solution editor; it uses Ecore for modeling and Xpand for code-generation [Wen+16]. *Robot Technology Component (RTC)* [OMG12a] comes with an Eclipse-based IDE, and also *RobotML* uses Papyrus [Papyrus] and Eclipse for modeling with an UML profile. The Robot Operating System Model-driven development tool suite (ROSMOD) [Kum+16; Kum+15] is a relatively young approach. It provides a graphical model-driven component-based development tool to support rapid prototyping for ROS and comes with its own component model.

While there is a lot of tooling around, it is typically dedicated for one purpose, i.e. one task that may be covered by a DSL (see section 2.6.5). It, however, is required to hand over the output of one workflow step to the next that must be supported by tools: Tools are not integrated with respect to an overall workflow (no need to "leave" one IDE) and not integrated with respect to superordinate structures as it is necessary for composition. Tooling is very often tied to one target system and each environment comes with its own set of tooling and/or component model. The contributions of this thesis to the *SmartMDSD Toolchain* realize the proposed composition structures in a single Integrated Development Environment (IDE). The structures and the toolchain enable the collaboration of the different roles in the composition workflow such that they can complement each other. In other words: The toolchain guides the involved roles to apply and benefit from the composition structures. The composition structure is independent of the used target framework.

2.6.5 MDSD and DSLs in Robotics

The need for domain-specific research on MDSD for robotics has been identified by the *Robotics 2020 Multi-Annual Roadmap (MAR)* [euR16]. It is being addressed by the robotics community, for example in the *Domain-Specific Languages and Models for Robotic Systems (DSLRob)* workshop series [DSLRob] which specifically focuses on the development of DSLs for robotics. MDSD topics are more frequently being featured in specific journals, e.g. *Journal of Software Engineering for Robotics (JOSER)* [Joser]. The European Horizon 2020 Innovation Action *Composable Models and Software for Robotics Systems (RobMoSys)* [RobMoSys] specifically addresses the combination of existing modeling approaches for robotics. Notable model-driven approaches in robotics are *RTC*, *RobotML*, *ROSMOD*, *BRICS*, and *ReApp*. They are described in more detail in section 2.6.3. An overview on MDSD in robotics can be found in [Bru15] and [RMT14].

Most activities in MDSD on robotics are focused on providing DSLs. For example, in [SS11b] and [KB12] for task coordination, in [Hoc+13] for modeling and code-generation for deployment of ROS nodes, in [Bau+13] for modeling manipulation and grasping using a Software Product Line (SPL), and in [GB11] to model reference architectures using a SPL. An extensive overview on the state of the art on DSLs in robotics can be found in [Nor+16]. These DSLs have in common that they are dedicated for one purpose, as would one expect, but are isolated tools with respect to overall system design or workflow. In contrast to that, this thesis proposes a harmonic / balanced structure and integrates several dedicated DSLs into a consistent approach and development environment.

2.7 Related Initiatives and Ongoing Projects

Most of the activities in robotics aim at advancing in technical challenges. This can be observed via the two top robotics conferences IEEE ICRA and IEEE/RSJ IROS. The need for systematic software engineering in robotics has been recognized lately and it started getting the attention it deserves. However, there is still a long way to go in comparison with other domains such as software engineering in automotive.

A rather small community is active in developing and promoting means for (model-driven) software engineering for robotics. For example, the *Domain-Specific Languages and Models for Robotic Systems (DSLRob)* workshop series [DSLRob] and the *Journal of Software Engineering for Robotics (JOSER)* [Joser] advance on software engineering for robotics. Initiatives such as the *RAS Technical Committee on Software Engineering for Robotics and Automation (TC-SOFT)* [Tc-Soft] and most important the euRobotics AISBL topic group on *Software Engineering, System Integration, Systems Engineering* [euR] promote the relevance of software engineering for robotics and the need for domain-specific adaptation of software engineering both towards the general robotics community and to funding activities in Europe.

Two ongoing Innovation Actions in the European Horizon 2020 research and innovation programme are of interest to this thesis. *ROSIN (ROS-Industrial quality-assured robot software components)* [Rosin] aims at enhancing the quality of ROS-Industrial, both the framework and its content, via continuous integration and code-testing in order to push robotics by broad availability of high-quality software. *RobMoSys (Composable Models and Software for Robotics)* [RobMoSys] focuses on a model-driven and composition-oriented approach to system integration to establish a European robotics software ecosystem. ROSIN aims to enhance the quality of existing software through software quality assurance. RobMoSys addresses better quality of robotics software right from the beginning via providing adequate and sound structures as a fundament that improve the quality and composability of building blocks.

The RobMoSys Horizon 2020 project is part of the European efforts to create an EU Digital Industrial Platform for Robotics. Its funding on EU-level underlines the relevance of a model-driven and composition-oriented approach. This thesis is closely related to RobMoSys and the research in this thesis therefore also contributes to these highly relevant topics. Many insights of this thesis contributed to the overall vision of RobMoSys and contributions of this thesis are part of the RobMoSys composition structures.

On a German national level, the BMWi PAiCE project *SeRoNet (Plattform zur arbeitsteiligen Entwicklung von Serviceroboter-Lösungen)* [Bun17] has a similar aim as RobMoSys and it is expected that this project will reuse structures from this thesis as part of the SmartSoft World and ReApp.

2.8 The SmartSoft World

The term "SmartSoft" was originally introduced with the SmartSoft Framework [SW99b; Sch04a]. It addresses the complexity of software development for mobile robots. SmartSoft nowadays is used as an umbrella term for concepts, principles, tools, and content that are developed at the Service Robotics Research Center (Service Robotics Ulm, [Ser]) to extend and carry on the core motivation of the SmartSoft Framework. The "SmartSoft World" includes, amongst others, two main reference implementations of the SmartSoft Framework (based on the ACE and CORBA middlewares), model-driven concepts implemented in the Eclipse-based SmartMDSD Toolchain, concepts and implementations for task sequencing, and a set of reusable software components. The results of this thesis are part of the SmartSoft World. Methods, concepts or implementations in relation to the SmartSoft World are often prefixed with "Smart" for identification.

2.8.1 A Brief Review of the SmartSoft Framework

The SmartSoft Framework [SW99b; Sch04a] applies a component-based and service-oriented approach. Loosely coupled components provide execution containers for the implementation of algorithms that communicate with other components through services. The only exchange of information between any two components is through *services*. There is a clear separation between the outside view of a component and its inside view. A service (Fig. 2.2) in the SmartSoft Framework is formed by a communication pattern (communication semantics) and by a set of communication objects (data structures)

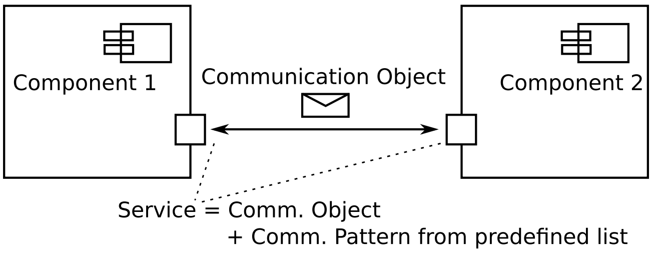


Figure 2.2: Components in SmartSoft communicate through services. A service is a combination of a communication object and a communication pattern.

The SmartSoft Framework defines "a small but sufficient set of carefully chosen communication patterns for component interaction" [Sch04a]. They support maintaining a stable interface towards the user implementation inside the component and a stable interface towards other components. Communication patterns limit the set of possible interaction mechanisms for the sake of composability. They provide a fixed set of few but sufficient communication mechanisms for robotics such as one-way "send", two-way "request-response", and "publish/subscribe" mechanisms on a timely basis (push timed) or based on the most recent update (push newest). The patterns also contain coordination patterns [Lut+14] to orchestrate a robot's components for task sequencing, e.g. discrete event notification, run-time parameterization and lifecycle management in the state-pattern (see section 2.9).

Communication objects [Sch04a; Lut+14] define the data structure that a service uses to exchange information. Communication objects are C++-like objects that implement framework internal access methods and can implement additional user methods (getter and setter) for convenient data access. Communication objects are always transmitted by value for decoupling. It is a best-practice that communication objects are self-contained entities [Lut+14]: Communication objects include all information that is needed to process them; no additional information retrieval should be necessary to interpret or use it. For example, a laser scan is tagged with the position where it was recorded.

2.8.2 The SmartSoft Framework in the Context of this Work

The clear separation of the SmartSoft Framework between the component internal and the component external view enhances component composability via communication patterns. This separation provides basic means that enable a model-driven approach, separation of roles and system composition. While these are basic structures that also enable composition, it requires structures that consider them in context of the overall composition workflow and the involved roles. Such structures are not provided by the SmartSoft Framework.

The approach presented in this thesis uses the concept of SmartSoft services (communication patterns and the concept of communication objects). This thesis formalizes them as "service definitions" to enable creating and using a domain structure in a composition workflow. The workflow is realized in a model-driven Integrated Development Environment (IDE) that supports users in applying the approach. Service definitions increase the separation of roles and the composability of components. Service definitions introduce a structure that components adhere to and can rely on. This prevents the manual organization by policies and replaces the communication between roles in an ecosystem. In SmartSoft, there is too few guidance and yet freedom of choice with respect to managing communication objects and services of components and their component alternatives. As a result, this limits composability and system composition with SmartSoft can only be achieved via following documented policies and guidelines. Service definitions, as presented in this thesis, contribute at this end to explicate the necessary information and manages them in the overall model-driven composition workflow.

The communication patterns and the communication objects are an important contribution of the SmartSoft Framework to structure systems and to come up with composable components. To achieve system composition with separation of roles in an ecosystem, semantics on the application-level also must be considered. Service properties address this aspect and support communication patterns and communication objects in service definitions in order to improve composability and separation of roles.

The approach presented in this thesis provides a structure to address the activity of putting together building blocks in a holistic model-driven composition workflow. It enables using SmartSoft in an ecosystem. It introduces service definitions with explicit properties to organize interfaces between components, component modeling, component (instance) parameterization, component selection, composition of component, and deployment. The thesis realizes this in a model-driven structure and applies model-driven tooling to raise SmartSoft from the code-level to the model-level.

2.9 The 3T Architecture in Robotics

Bass, Clements, and Kazman [BCK12] define a software architecture as "the structure or structures of the system, which comprise software elements, the externally visible properties of those elements, and the relationships among them". A high-level system architecture that is in widespread use in service robotics is the 3T architecture[Bon+97; Fir89] which introduces "three interacting layers or tiers" [Bon+97]. "Tier" as in the "3T / three tier architecture" is not to be mixed with "tier" in this thesis' "composition tiers" to structure the ecosystem. The 3T architecture was refined and put in context of SmartSoft in [Sch04a; SW99a]; more recent views and descriptions in relation to SmartSoft are covered in [Lut+14; SS14b]. The work presented in this thesis is based on the three tier robot architecture. The architecture is divided in a skill, sequencing, and deliberative tier (Fig. 2.3).

A skill is a basic and encapsulated capability or function of a robot. The skill tier is a reactive tier with lots of data communication. Communication in this tier is on a subsymbolic level. There are components running low-level algorithms, continuous control loops such as collision free driving. For example, a skill can realize speech input, speech output, or object recognition. The skill tier also runs components providing hardware access, for example a laser ranger or RGB camera. This thesis uses the term skill for all components that are coordinated by the sequencer. In the context of SmartSoft robotics behavior, a "skill" is defined in a more narrow scope; skill here refers to the link between robotics behavior tasks (or task plots) and services of components.

The sequencing tier works on a symbolic level and consists of a single and central sequencer component. It is responsible for coordinating and orchestrating the skill components. The interaction between sequencer and the components it coordinates is event-driven on a symbolic level. The sequencer is in control of the whole system and executes an action plot (e.g. make coffee, modeled as state chart [SS14b] or task tree [SS10]) by coordinating the skill components in a certain sequence as encoded in the action plot (e.g. recognize objects, grasp cup, place cup, etc. to make coffee). The sequencing tier thus covers both the knowledge about the task (how to make coffee) and execution of that task by orchestrating the single components (object recognition to recognize the coffee machine). The sequencer triggers actions on the component and receives discrete event notifications; both by using the SmartSoft coordination patterns *parameter*, *state* and *event*. All problems encountered in the skill that cannot be handled or recovered by the skill itself must be reported [Nor90]; the sequencer then can resolve the problem as encoded in the action plot.

A knowledge base is used to store persistent data to build and use a world model (e.g. locations of persons and objects, names of persons). The deliberative tier runs advisors for the sequencer. For example, an expert component for topological path planning through a building or finding the optimal sequence of stacking cups and plates.

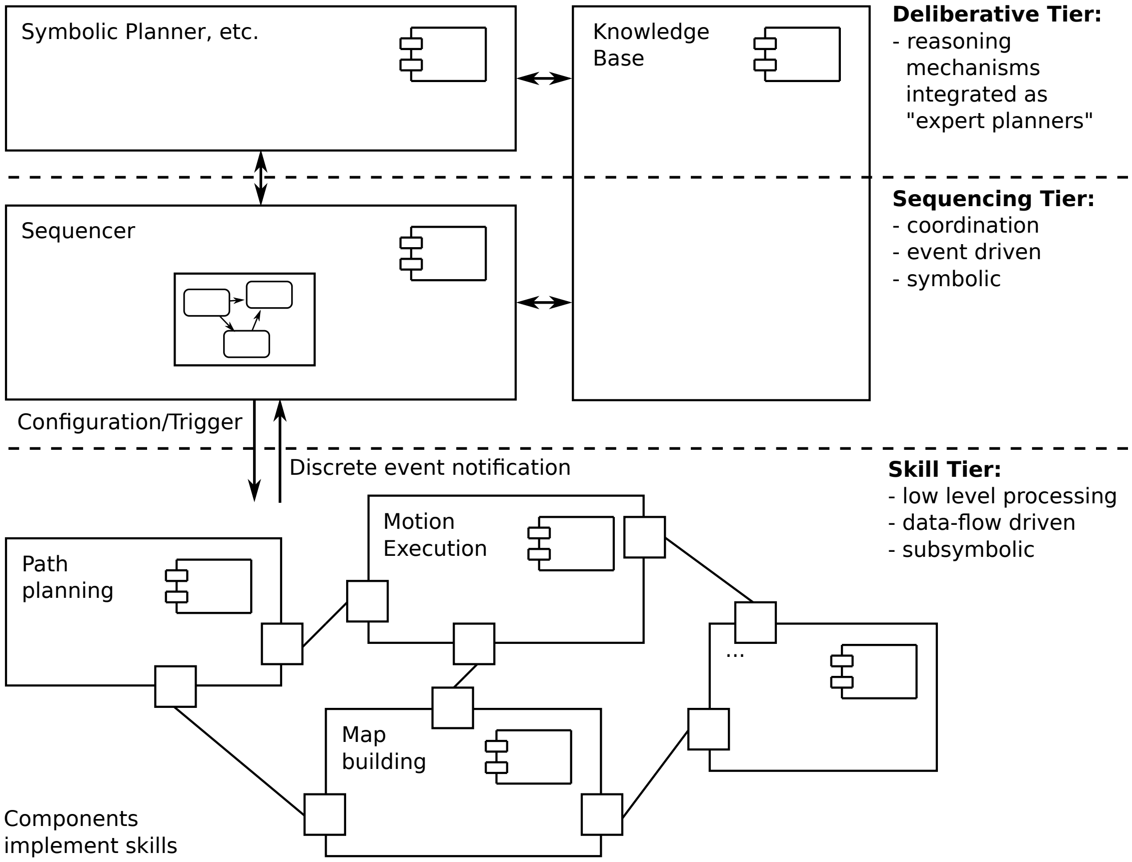


Figure 2.3: The 3T architecture that is commonly applied in service robotics. (Figure adopted from [SW99a])

3 Towards System Composition

This thesis envisions the composition of building blocks that are shared and exchanged among the participants of a robotics software business ecosystem (Fig. 3.1).

This chapter describes the *vision of composition* and the *vision of an ecosystem* (section 3.1). It shows that *structures are required* to realize such an approach for robotics (section 3.2). The chapter then describes the consequences to such a structure and such an approach (section 3.3). It also looks at the considerations to address these consequences (section 3.4).

As the core contribution of this thesis, the following chapters explain the approach that realizes the technical structures for system composition in an ecosystem for robotics.

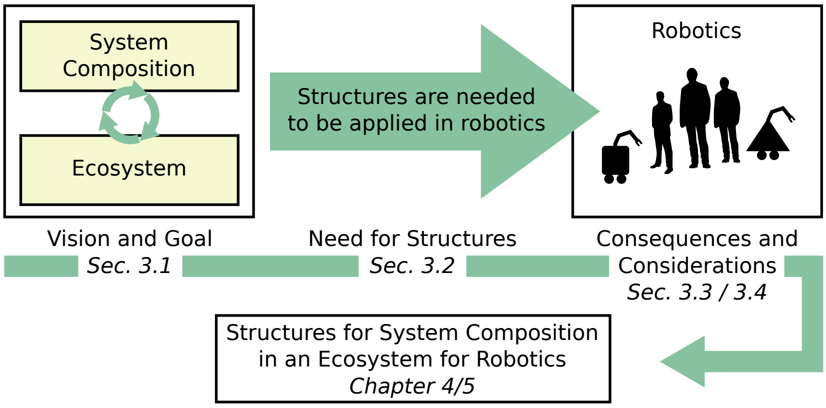

System Composition
Ecosystem
Structures are needed to be applied in robotics
Robotics
Vision and Goal
Sec. 3.1
Need for Structures
Sec. 3.2
Consequences and Considerations
Sec. 3.3 / 3.4
Structures for System Composition in an Ecosystem for Robotics
Chapter 4/5

Figure 3.1: Structures are necessary to enable system composition in a software business ecosystem for robotics. This thesis identifies the necessary technical structures for building blocks to support system composition in an ecosystem for robotics.

3.1 Vision and Goal

To describe the vision and goals of this thesis, the definition and meaning of "system composition" is discussed to distinguish it from an integration approach. After describing a robotics business ecosystem, five inter-related use-cases are presented to form the overall picture.

3.1.1 System Composition

System composition is considered the activity of bringing parts together. "Integration" is widely used in software engineering and describes the process of combining or assembling components into an overall/whole system [ISO15; ISO10]. Composition and integration, however, are not synonyms (see [SW02]).

System Integration

Most applications in service robotics are "created" from scratch, e.g. by "blank-sheet design" through only breaking down requirements (cf. [OW13]), then meeting the requirements, and finally bringing together the parts in the end. Bosch and Bosch-Sijtsema [BB10] call this an integration-centric approach where "significant effort is placed on the last stage of software development, where independently developed parts are manually integrated and validated" [BB10]. It is the approach that is applied in most companies [BB10].

System Composition

Composition is "the action of putting things together" [Oxf, "composition"]. In context of this thesis, it is thus the action or activity of putting together a service robotics application from existing building blocks (here: software components) as is, in a meaningful way, flexibly combining and re-combining them depending on the application's needs. Composition-oriented approaches [BB10] follow the idea of a building blocks approach. Instead of a problem breakdown, they create something new from something existing. In such an approach, systems are built by combining and putting together parts in a flexible way [Jan12, Part IV]. This is in contrast to building each part mostly from scratch again and again. It is different from the substantial effort to write glue-code and adapters to integrate existing software.

System Integration vs. System Composition

The word "integration" stresses the need to enable parts to work together. It implies that the parts are different in some way (i.e. not compatible), that it requires effort to overcome this by modifications. This is supported by Petty and Weisel [PW03], who report on integration in the context of computer science being the effort to modify components to make them interoperable and to work with others (see also [Erl08, p. 92ff]). Stafford and Wallnau [SW02] even go further and compare integration to "the mechanical task of 'wiring' components together" [SW02] while modifying their interfaces to match. Writing glue-code and adapters is a typical activity to make parts and their interfaces compatible to integrate. For example, integrating functions of a library into an existing software project. Integration of parts often does not come with the intention to use the component beyond that particular purpose [PW03].

In contrast to integration, composition puts a focus on the new whole that is created from existing parts rather than on enabling parts to work together just by gluing them together: The resulting whole still consists of its parts [SW02], they are still visible and are still isolated as entities and are thus still exchangeable. In contrast to composition, a part that is integrated amalgamates into the whole: i.e. becomes one part; or mixes, as red and green water will mix. It is thus hard to separate the parts again or to exchange individual parts[1]. Even if that part is removed, new adapters are required. Integration is inefficient and expensive.

Due to the required in-depth knowledge, integration creates a tight coupling between the parts of the system, making it hard to manage parts, reuse parts, distribute work or even outsource work [BB10]. The reason is that the parts of the system each come with assumptions about the environment they were developed for and in which they are supposed to be used in: they are shaped by their requirements [GAO95]. Thus, integration requires knowledge and access to all parts of the system on all abstraction levels.

System composition is about adhering to a composition structure. It is about putting in effort once to comply to the composition structure and gain immediate composability with other building blocks. In contrast, integration is about building individual adapters between mutual parts or even modifying the part itself. High effort comes through the infinite number of combinations between building blocks that each require a new adapter each time they are integrated. Software components that are subject to composition can be taken "as-is", only being configured within modeled boundaries without requiring adapters. They thus must be built with this intention right from the beginning. The context in which they will later be composed is unknown, which puts special requirements on their composability and on the overall workflow.

In software engineering in general, "integration" is the process or step of combining or assembling components into an overall/whole system [ISO15; ISO10]. In this sense, the effort for system integration is lowered by applying a compositional approach.

[1] "Integration" is very closely connected to "to incorporate", which means "to take in or contain something as a whole; to include" [Ame11]

An Analogy: Jigsaw Puzzle and Lego Building Blocks

There is an analogy between integration and a jigsaw puzzle (see also: Fig. 1.3): Sawing a wooden plate apart produces many pieces where only two of them will fit together. To build a jigsaw puzzle, one literally first requires the overall picture to be painted on the wooden plate. Only then can it be sawed apart. It is not possible to produce the jigsaw puzzle in the reverse order. One cannot saw apart the wooden plate, then mix the puzzle parts and start painting them. Once the puzzle is produced, one has a very hard time to put together the pieces by their shape alone without knowing the picture of the puzzle. Connecting two arbitrary pieces after the wooden plate is sawn apart will require modification or adapters to make them fit. This will most probably also modify the overall appearance.

The jigsaw puzzle is in contrast to composition of (classic) Lego plastic building blocks for kids which can serve as an analogy for composition. Any two pieces can be built upon each other because all adhere to a certain structure. Pieces can be exchanged according to the needs (e.g. uniform color of all blocks) and known constraints (one 4x4 piece can be replaced by another 4x4 or by two 1x2). Lego pieces can be put together in a much more flexible way compared to puzzle pieces. Lego pieces can be combined and re-combined as required. When exchanging one Lego block in an existing composition, enough information is available about the missing block to find an alternative: the size and dimension as well as the color. With a jigsaw puzzle, one must find the one that fits, most likely through trial and error. In case the puzzle piece got lost, one must produce a new piece that exactly fits.

The Need for System Composition

Integration-centric approaches are suitable for robotic prototypes that demonstrate a single ability in a laboratory or for development of a single robot application. Software development for service robotics have to overcome the limitations of integration-centric approaches, and need to move to composition-oriented approaches because (i) it is not possible or not desirable to address each part of the system in all depth during third party reuse and/or to come up with adapters, (ii) because of the complex and interdisciplinary nature of robotics, and (iii) because this will hinder the emergence of a robotics business ecosystem with component suppliers and users. The need for composability and composition of software has also been identified in the "ITEA Roadmap for Software-Intensive Systems and Services" for software engineering in general [ITE09] and in the "Robotics 2020 Multi-Annual Roadmap (MAR)" [euR16] for robotics.

A composition-oriented approach brings many benefits. It helps to manage the overall complexity, since the overall system consists of parts and these parts are solved individually (cf. divide-and-conquer approaches). The problem complexity of such a part is lower than that of the overall system and it can be addressed by an expert person or an expert company to distribute and decouple work. Reuse of already developed parts can be made on a much better suited level of granularity both within a company and externally. The building blocks for composition ideally exist in-house within the institution or from third party suppliers. The huge benefit of composition, however, is that it supports the emergence of a software ecosystem [Bos09] in which organizations contribute to and take parts from to collaborate and compete. Ecosystem participants collaborate based on stable structures and supply building blocks (as in Lego, see section 3.1.1). They do not need to work together as a big team in the sense of collaboratively developing the whole system (as in a jigsaw puzzle, see section 3.1.1).

System composition in this thesis focuses on composing software components at a certain level of abstraction which is of interest to the robot (system) and which is in close relation to its abilities (section 3.2.3). The granularity of system composition is therefore driven by the needs of the application (e.g. object recognition), while the building block that offers such an ability itself is technically-driven or problem-driven as it solves a particular problem (e.g. the fusion of several algorithms for the purpose of object recognition). The "service" that the building block provides is therefore of interest to the application along with additional properties, for example the quality with respect to service execution. The details of how that service is realized (e.g. algorithms, implementations, and methods) might not be of first interest to the application.

The goal is to build the software of a service robot by composing software components as building blocks. To build a coffee delivery robot, for example, one would compose software components that provide needed basic abilities that are needed such as object recognition, speech input, speech output, collision avoidance, path planning, mapping, etc.

Building Blocks Alternatives

System Composition is the activity of putting together an application from building blocks such that they meet the application's needs. Since there might be different needs in different applications, this requires not only one building block that satisfies a need but several *alternatives* to choose from. Without alternatives, system composition would resemble a jigsaw puzzle (see section 3.1.1) with exactly one combination.

An "alternative" to a building block is another building block that satisfies the same need. From the consumer point of view, it provides an equivalent service. For example, a histogram-based object recognition and a feature-based object recognition building block are alternatives since both provide object recognition. Alternatives can differ in quality, implementation standards, maturity level, performance, or functionality. We call this *diversity of performance*.

For service robotics, we envision a system composition approach that allows for the identification of alternatives and expressing their diversity. It comes with the need for systematic support for the developers to select the one that matches the needs of the application best.

3.1.2 A Software Business Ecosystem for Robotics

This thesis envisions a robotics business ecosystem in which various stakeholders can network and collaborate (Fig. 3.2). An ecosystem is a collaboration model (cf. [BB10; IL04]), which describes the many ways and advantages in which stakeholders (e.g. experts in various fields, organizations) network, collaborate, share efforts and costs around a domain or product [IL04; IL04; Moo93; PV05; ITE09]. System Composition assumes the existence of building blocks. These building blocks can be built and shared in such an ecosystem by the ecosystem's participants.

There are several key points that form an ecosystem [PV05] (see section 2.1) which we can put in relation to this thesis. They will shape the robotics business ecosystem as envisioned in this thesis:

Population; interconnected; cooperation. There are several stakeholders (organizations, partners, persons) working in such an ecosystem. Each stakeholder contributes expertise or products or benefits from such. Therefore, there is a need to manage this cooperation, giving each stakeholder sufficient freedom to work. Separation of these roles must at the same time ensure that their contributions, that were built in isolation, can come together in the end without the extra effort of integration and adapters. The participants of the ecosystem might be cooperating in the sense of working closely together ("team work" or "custom development") or might not know each other at all (supplier does not know the individual customer in a retail store).

Competition. Competition is about finding the best one in terms of my needs from a set of parties. In relation to this work, this can be the competition to find the one from several building block alternatives that matches the needs of the application.

Structure; technological platform. From a technological point of view, such an ecosystem needs a proper structure that makes all contributions fit together. Such a structure must draw boundaries between the individual contributions. There must be the appropriate tools to work with this structure and to provide a platform to actually "meet" each other, thus finding and delivering outcomes/products.

Considering an ecosystem perspective will contribute to push forward software development in the interdisciplinary domain of service robotics. Establishing an ecosystem for robotics brings many benefits. It allows to outsource parts of development or use components that are already available [ITE09, p. 36]. It allows to share existing solutions or adapt them for new markets [ITE09, p. 36]. According to [Bos09], applying an ecosystem increases the value of existing products since they can be offered to new users, accelerates innovation through opening up a complete domain and shares the cost of innovation with partners.

As robotics is a diverse field, most contributors will have dedicated experience and expertise. They should be able to contribute software building blocks reflecting their focus and expertise. This expertise is made available to others who can benefit from this expertise just by using the provided software building blocks, applying system composition methods to combine them to new applications. "Collaboration" of participants in an ecosystem refers to complementing each other and sharing independent and self-contained development artifacts. Collaboration is not meant in the sense of close collaboration as in working in a team and collaborative editing (see section 2.2.1). This thesis identifies the necessary technical structures for building blocks (software components as unit of composition and unit of exchange) to support the concept of an ecosystem for robotics.

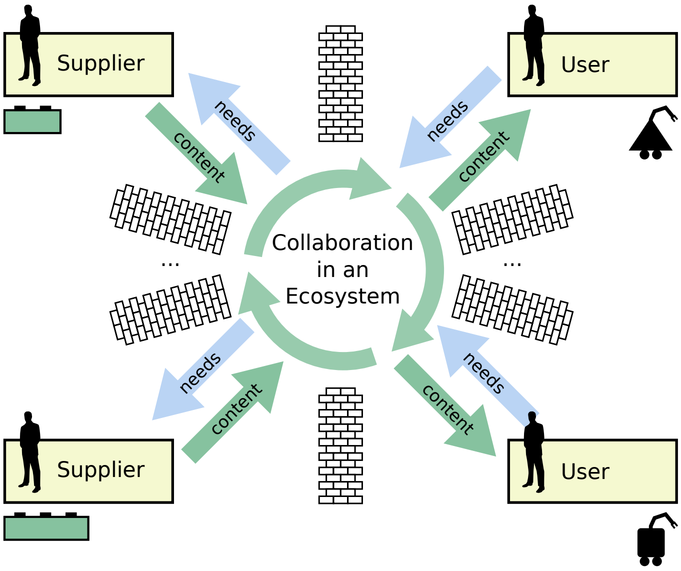


Figure 3.2: A robotics business ecosystem and the collaboration of its main participants. Suppliers provide building blocks for users to build robotic applications. The participants must not know one-another but still must be able to collaborate, i.e. complementing each other and sharing independent and self-contained development artifacts.

There are several ways to scale a robotics business ecosystem. One can think of a global ecosystem with high standards which manage the collaboration between companies or even complete branches of sub-domains in robotics. On a smaller scale, one can think of establishing such an ecosystem within a company, thus collaborating with several departments of that company, collecting a set of building blocks and using them for composing different applications according to projects with the customer.

An ecosystem for service robotics needs to be decentralized (see [Jan12]), i.e. not in control of a single company only. This would be too narrow and would limit the evolution that is necessary to find the right structures which needs to be done in collaboration with developers and community (see [BB10])—especially in a young field as service robotics. An ecosystem for robotics must thus be driven and shaped by the community or representatives of the community itself.

Establishing an ecosystem will establish one or more places to exchange building blocks. We call the market of components the technical "meeting-platform" to collect and exchange building blocks. The success of such a market depends on components to be used "as-is" [SW02].

3.1.3 Use-Cases

Consider a small tech-company with knowledge in the field of human machine interaction and in human behavior. The company plans to build a robot to support people in navigating through unknown buildings. The robot shall greet people at the entrance and shall guide them to their meeting point. This can be useful, for example in public administration buildings. The company does not have deep robotics knowledge and therefore, must concentrate on their main expertise.

The envisioned robot application has special requirements towards path planning: There is a human that shall follow the robot, but typical available path planning components are made for robots. These do not take into account the way humans typically move through buildings.

The Company Perspective

Based on the previously described setting, we can describe the use-cases that the company will face while developing and maintaining the robot application. We will look at it from the perspective of the company first (Fig. 3.3), before taking an ecosystem-perspective in the next section.

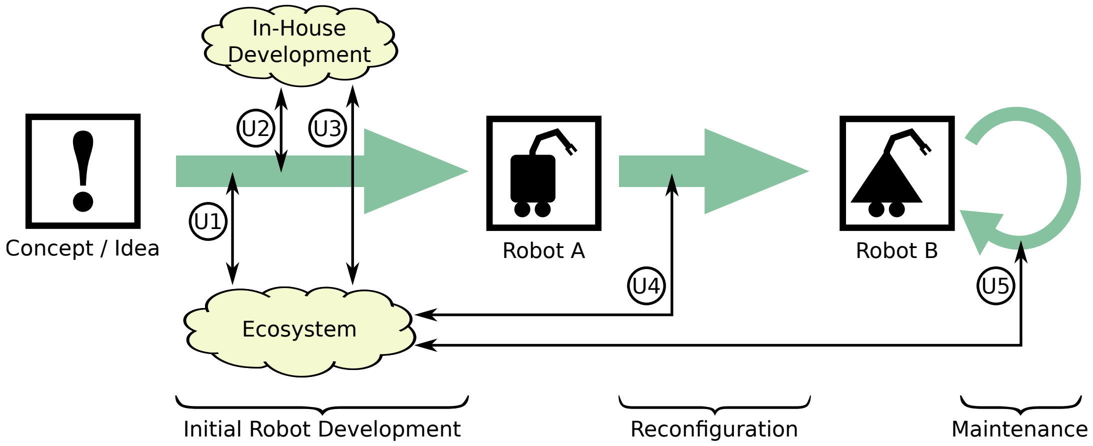


Figure 3.3: Use-cases while developing and maintaining a service robot application from the perspective of a single "example-company" that is part of the ecosystem.

(U1) **Use Existing Building Blocks.** Because of its limited expertise in robotics, the company wants to rely as much as possible on already existing third party building blocks. These shall already provide a certain level of granularity and functionality that is of use for the company. For example, they have no insights into Simultaneous Localization and Mapping (SLAM). Thus, they want to use a complete localization solution instead of just a SLAM library or even only a robotics toolkit which would require them to build SLAM on their own.

Motivation: Lack of robotics knowledge; reduce development time and time to market; increase quality (since existing solutions come from experts) and thus robustness of the robot; participate in the ecosystem (here: use existing solutions for everything that is not within your own expertise and thus focus on your own core expertise).

Challenges: How to express the needs of the application and how to identify and select existing and composable solutions that match these needs? How to ensure that the building blocks will work with others?

U2 **Develop Custom Solution.** Because of its high expertise in the use-case, the company wants to develop a custom solution for a particular sub-problem of the application: e.g. a component that makes the robot follow other people or a path planning component that considers special requirements in navigating people. The company wants to use this in-house development as an "existing solution" when developing new applications in the future as described in U1.

Motivation: No or no satisfying and composable solution exists for a given problem; reuse in other applications within the company, the custom solution is becoming an "existing solution" for the company; eventually call for someone developing a custom solution defined by its needs (call for tenders).

Challenges: How to develop an own solution to a certain problem such that it is composable in the robot application?

U3 **Supply Custom Solution.** After the company has finished developing the custom component, they plan to supply and sell it to others. This use-case is the counterpart of U1, thus acts as its enabler in the perspective of third parties.

Motivation: Fill market niche; revenue by multiple users to make money even when higher development efforts; unique selling point / feature in development of the custom solution; participate in the ecosystem (here: supply/sell component for use by others).

Challenges: How to build the component such that it is composable by other, third party users? How to express what the component provides? How to express its needs towards other components that it requires to function?

U4 **Reconfiguration.** As the product is very successful, the company wants to extend market share. It wants to provide a new variant of the product with less performance at much lower costs. The company thus wants to reconfigure the existing system and, for example, exchange hardware with less expensive hardware, accepting lower performance. This use-case can also be applied to custom applications towards adapting to new requirements or a low number of units down to batch size one. Slight changes in the application's requirements should not lead to huge development efforts.

This use-case is closely related to U1, but focuses on bringing a new product or variant to the market by modifying an existing product (possibly with third party software as in U1) rather than building it anew.

Motivation: Design-time reconfiguration; reduction of development effort and costs; reconfiguration necessary through changed requirements (quality, cost target).

Challenges: Managing the consequences that come through the exchange of components: How to identify and select existing and composable solutions that match the new needs of the application? How to detect whether the new component still maintains system consistency or breaks it, for example because the components of the application have needs that are not satisfied by the new one—or the new component has certain assumptions that no longer hold true in its new environment?

U5 **Maintenance.** When the application broke down and needs maintenance, it should be brought back into operation as soon as possible even though identical spare parts might not be available. Alternative replacements must be considered.

This use-case is closely related to U4, but focuses on fixing a broken system after it has been set into operation in contrast to building a new variant at design-time. While U4 wants to reconfigure the system to cover the need of the new application as close as possible while reducing costs, this use-case aims at getting the system back into operation with a performance that is the same as the original one.

Motivation: Reduction of maintenance effort and costs; maintenance-time reconfiguration.

Challenges: Same as U4.

The Ecosystem Perspective

The company is a participant in the ecosystem (Fig. 3.4). While the development seems to be linear from the company's point of view, there is no order in which participants in the ecosystem interact: they are separated in time and space, working in parallel.

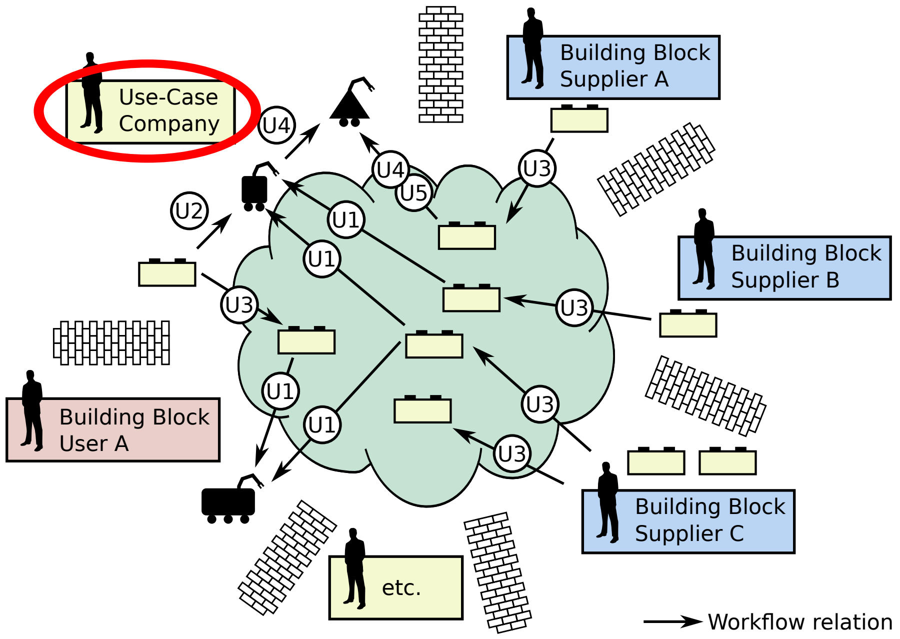


Figure 3.4: The ecosystem perspective on the use-cases. It includes the company and other ecosystem participants. This figure is a detailed version of Fig. 1.3 (right-hand side) and Fig. 1.4.

There is no direct communication between building blocks suppliers and system builders as it would be the case in a value-chain. Any two participants do not interact in a bilateral way. They do not negotiate technical interfaces and other agreements. Building blocks just must fit and work. All participants form a network of collaborations: produce once, sell many times.

A participant in the ecosystem is not limited to a single role. For example, the company acts as a building blocks user in use-case U1 and as a building blocks supplier in U3. Use-cases U1 and U3 may be considered the same use-case, but their different point of view makes them two individual use-cases. They are important for the overall vision of composition in an ecosystem, since both perspectives need to be considered when designing appropriate ecosystem structures.

3.2 The Need for Structures

Today's service robotic applications are designed and developed with access and influence on all levels of a system. All details and properties that build upon each other across the whole application are carefully designed, agreed, and built such that they work.

These agreements are managed in project management and software development methodologies between the involved people and companies along the value chain. In an ecosystem, however, the separation of roles is natural. The stakeholders that want to collaborate act independently. They are distributed and are separated in time and location. They do not necessarily know each other. In an ecosystem, there is no way to manage the overall development in terms of management and software processes [Bos09]. Agreements cannot and should not rely on such a process. Management processes do not scale with ecosystems. The natural separation of stakeholders and thus the lack of common coordination is a central challenge when pursuing an ecosystem and the strength of a compositional approach [Bos09].

A compositional approach in an ecosystem with naturally separated roles thus needs to organize agreements and collaborate by structure rather than by management (Fig. 3.5). This does not mean that there is no management and that there are no processes anymore. These processes still exist within each single participant or step. The process of a single participant might be different to the process of another participant. This particularly holds true for distributed organizations. It is where traditional approaches for collaborative development fall short. The approaches that are applied must address the way in which participants shall collaborate. They collaborate in a loosely coupled way and, thus, the approach applied must support loosely coupled collaboration of participants. Composition and composability is about agreements to fulfill. Composition in an ecosystem requires to find and hold on agreements without negotiating them bilaterally between the participants. Participants must adhere to an overall structure once to gain composability of their building blocks with all other participants (see section 3.1.1).

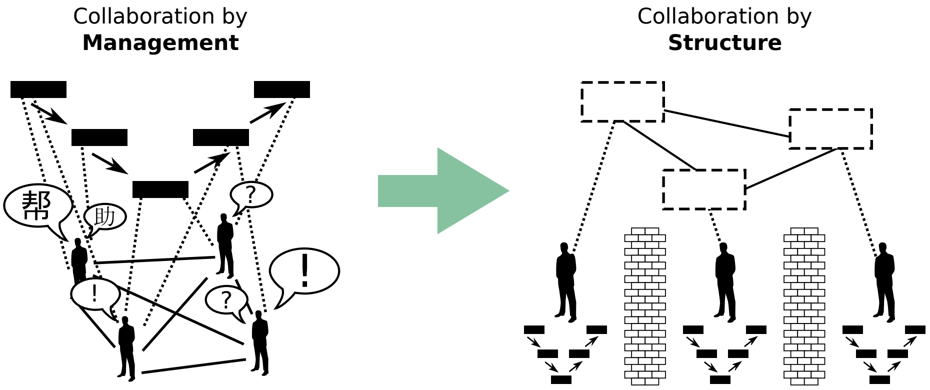


Figure 3.5: Collaboration by management is not possible within an ecosystem approach: system composition needs collaboration by structure.

A composition structure defines a set of reference elements that can be taken over by users of the structure. It serves as a framework to organize and structure parts that then will be part of a composition. It provides means to explicate information that is necessary for composability in a structured and processible way (in contrast to a free structure) such that the parts can be managed and handed over between stakeholders that supply or use them.

The agreements and negotiations are no longer between two partners, thus gaining only compatibility of these two. They are towards a global structure, thus gaining immediate access to all parts that adhere to the same structure: All parts that follow the structure are composable (Fig. 3.6). As a result, there is no need for ecosystem-wide management processes.

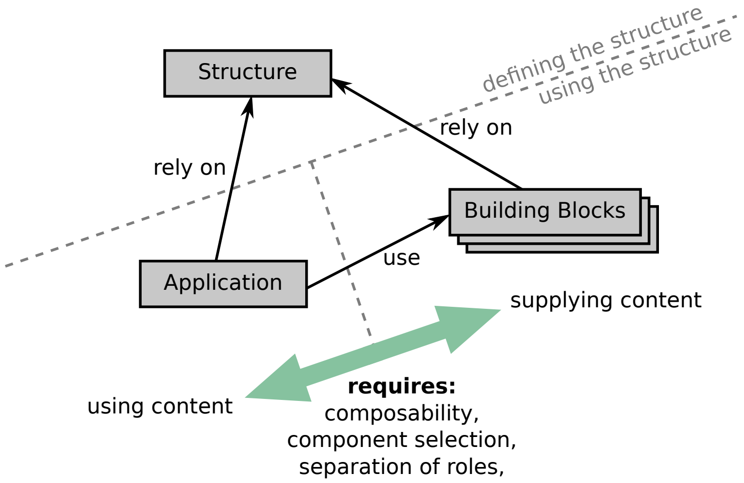


Figure 3.6: Superordinate structures for system composition. The definition of structures in an ecosystem makes contributions composable.

3.2.1 Benefits of Structure

Pursuing system composition within a structure brings many benefits:

- Direct interaction between two stakeholders is no longer necessary. This is covered by the structure; both can rely on their common elements in that structure. Conformity to the structure brings immediate access to all others that conform to the same structure. This supports the transition from value-chains with tight bilateral agreements to value-networks that align to the given structure to collaborate.

- Defining a structure means limiting the number of options in a positive way to only relevant options (freedom from choice). This minimizes the number of different interfaces for the same purpose and thus increases the number of replacable building blocks one can choose from. Structure thus is required for composability.

- The definition of structures can support the transition from technology push to market pull. Structures that represent what is really needed will be taken up by participants that will provide contributions according to these structures, thus meeting exactly these needs.

- Defining and adhering to structures also eases access to the domain. Elements of the structure can be compared to a vocabulary of that domain. In the best case, such structure will converge towards a (de-facto) standard.

3.2.2 The Need for a Meta-Structure (Composition Structure)

Structures need stability to work [Moo93] but they will also undergo evolution [BB10] to keep up with technology changes. There is, thus, no final structure. An adequate structure for ecosystem collaboration cannot be provided by a single activity or organization (or even this thesis) to cover the broad range of domains within robotics. Instead, the goal should be to provide a superordinate structure (a "meta-structure") that enables composition per-se and that allows the definition of structures for the particular domains in robotics. Such a "composition structure" should only organize the basic composition, collaboration, and technical framework for the individual robotics (sub)domains to fill in details. For example, it should organize the exchange at component and service level. The definition of individual services and data representations (what is a laser scan?) should be up to organizations within particular domains in order to ensure that domain needs are being matched. Of course, this must not happen in complete isolation. In case several domain structures for the same purpose co-exist, these structures can evolve, and ultimately, the "right" structure can evolve to a standard: Addressing system composition in an ecosystem is not necessarily about finding the best structure, but about finding an accepted structure that works well.

3.2.3 Service-Level Composition in Robotics Software Architectures

The three tier architecture is a common architecture for building robots (section 2.9). In this control architecture, a sequencer coordinates the execution of software components (called "skill" in context of this architecture) according to a task plot. The sequencer is responsible for the high-level task execution (e.g. first "recognize objects on table", then "grasp the cup") while the software components on the skill layer encapsulate the functionality for these tasks (e.g. object recognition, motion planning). The skill layer is thus a major concern in this robotics architecture and it is desirable to assemble this layer using an effective compositional approach. At the same time, components and their interfaces on this layer require a certain level of granularity and abstraction such that they bring an immediate benefit to the robot application in which they are composed. Thus, this thesis focuses on composition at the level of services using components as the unit of composition and unit exchange in an ecosystem.

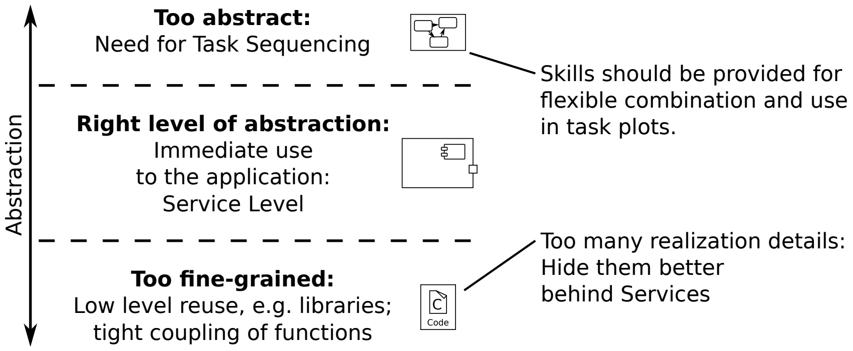

Too abstract:
Need for Task Sequencing
Skills should be provided for flexible combination and use in task plots.
Abstraction
Right level of abstraction:
Immediate use to the application: Service Level
Too many realization details: Hide them better behind Services
Too fine-grained:
Low level reuse, e.g. libraries; tight coupling of functions
C
Code

Figure 3.7: The granularity of components. The granularity and abstraction of components must neither be too specific nor too generic but must be of immediate use to the application.

Skills in robotics are particularly complex. Tight coupling of fine-grained operations and control loops that are required to provide them should not be exposed to the whole system: Fine-grained interfaces (that tend to be Application Programming Interfaces (APIs), e.g. at the functional level such as in MRPT [Bla], OpenCV [Its]) should stay within components and should stay hidden behind component interfaces. Exchanging components with too fine-grained components in an ecosystem will lead to component interweaving (see [Scho4a]) hindering system composition and separation of roles. An approach for system composition in robotics thus requires skills to provide high level interfaces on the service-level (Fig. 3.7).

There is no clear boundary of what is the right granularity. Asking the question "Will this component and its interface bring an *immediate* use to an application?" will help to evaluate (see [SW04]). A skill component, for example, that provides a low level of abstraction and provides certain functions (e.g. a blob-detector) will bring no immediate use when being composed for a delivery robot application. An object recognition capability, however, will bring immediate use to the delivery robot application since it is at the service-level (Fig. 3.7).

Very abstract capabilities that require task knowledge or sequencing of sub-tasks (e.g. make coffee) should not be provided as components, but should be covered by dedicated technologies making use of components and services (e.g. services for navigation, object recognition and manipulation). For example, SmartTCL [SS11b] and Dynamic State Charts [SS14b] are approaches to encode task knowledge and manage its execution; they are in line with the approach presented in this thesis.

The demand for composition is the highest at the level of services, since this level touches many heterogeneous domains: The skill level of an application requires services for many heterogeneous technologies (for example object recognition, manipulation, navigation) which each require deep expert knowledge.

3.3 Consequences on the Approach

System composition in an ecosystem requires structures. This section explains the requirements on such a structure for service robotics from three perspectives (Fig. 3.8): *composability* as the ability of building blocks to be combined and recombined into different compositions. Since composability is a cross-cutting concern, it needs consideration through the whole *composition workflow* that involves all steps, stakeholders and elements. Finally, the workflow must be applied by stakeholders who need proper *support* via tooling.

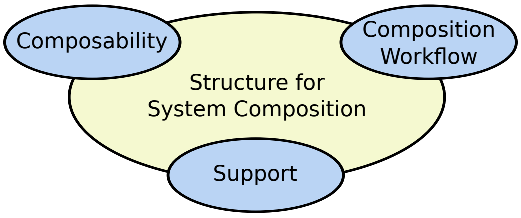


Figure 3.8: A structure for system composition has requirements originating from composability, composition workflow, and support via tooling.

3.3.1 Composability

This section defines composability in context of this thesis and discusses its aspects to consider.

Definition of Composability

Composability is the ability behind system composition that enables to put together parts in a meaningful way. It comes with composability as the property of parts that makes them become “building blocks”. Composability puts a focus on the new whole (system) that is created from existing parts. It is not just about making the individual parts work together just by amalgamating and uniting pieces that then become inseparable (see “system integration”, section 3.1.1). Petty and Weisel [PW03] provide a merged definition of “composability”. Even though their definition is targeted for simulation, it is still applicable in a broader context and taken as a basis for the definition in this thesis:

> *“Composability is the capability to* ***select*** *and* ***assemble*** *simulation components in*

> ***various combinations** into valid simulation systems to satisfy **specific user requirements**." [PW03]*

Further, Petty and Weisel [PW03] state that components are readily available to combine and recombine "as-is" from a repository or library to different applications that meet different needs. Open source components are white-boxes, giving full access to modifications and not requiring to accept them as they are. This is often seen as an advantage of open source [Del14; Frö02; BA99], since they can be adapted or can be fitted. But the additional effort, the required knowledge, and the eventually decreasing quality of the component require better approaches. Modification of source code is no synonym for composability.

Composability does not happen by coincidence, but must be carefully designed right from the beginning [Szy03]. From Service-Oriented Architectures (SOAs), we know that composability relies on collective and balanced application of several principles such as loose coupling, abstraction, reusability, autonomy, statelessness, discoverability, and contracts [Erl08].

Occurrence of Composability

With respect to system composition, composability must be addressed on three axes from two perspectives (Fig. 3.9): *between different components* Ⓐ, between *alternatives of components* Ⓑ (section 3.3.2) and *between components and the application needs* Ⓒ. The relations on all three axes need to be satisfied with respect to (i) syntax and semantic plus (ii) application and technical level perspectives to enable composability for system composition.

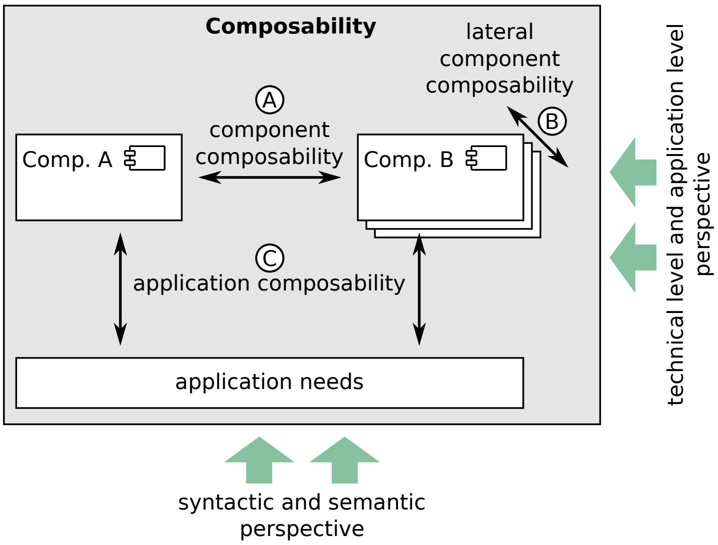

Composability
lateral component composability
Ⓐ
component composability
Ⓑ
Comp. A
Comp. B
Ⓒ
application composability
application needs
technical level and application level perspective
syntactic and semantic perspective

Figure 3.9: Occurrence of composability: Composability for system composition must satisfy the relations *between different components* Ⓐ, between *alternatives of components* Ⓑ and *between components and the application needs* Ⓒ.

Two components that are part of an application (component composability, Fig. 3.9) are not composable when they cannot interact meaningfully—for example when their interfaces mismatch or have different assumptions about their environment. To qualify for a component alternative (see lateral component composability, Fig. 3.9), neither the new component nor the system must need modification. The relation between components and application (application composability, Fig. 3.9) considers the needs of the application versus what the components provide: The application will not perform as intended if the components are not suitable for the application. This, for example, is the case when the component's quality or performance, e.g. localization quality, does not match what is needed by the application.

Syntactic and Semantic Composability

Composability can be viewed from a syntactic and a semantic perspective (cf. [PW03]). Syntactic composability refers to the ability that components can be connected and exchange data. As semantics refers to the meaning of something, semantic composability also refers to components that are composed meaningfully. For example, two components can be connected and exchange velocities via an integer over a specified protocol that both understand. This composition, however, might not be meaningful if both components have different assumptions about the velocity (e.g. speed in mm/s or m/s). Composability might look valid during design time at first, but issues might become visible during run-time as Staffort and Wallau [SW02] distinguish: They call it component mismatch when interfaces do not match and behavioral mismatch when the components plug together, but do not work as expected. Semantic composability also comprises the communication semantics as argued by Schlegel [Sch06], which is one additional important aspect in system composition.

Composability on the Technical Level and Application-Level

Technical composability is located at the level of components. It is about their interaction and making them communicate. This gives access to functionality as is partitioned by components. Most importantly, it includes the specification of interfaces. There are interaction mechanisms between building blocks such as *send* or *push* and *blackboard-based* or *message-based*. There is data that is exchanged between components and care must be taken of endianness and data types. There are agreements that go further, for example one building block involved in navigation might use cartesian coordinates in x/y/z versus another one using GPS WGS84.

System composition shifts the focus from parts to the whole that is being composed: the application. Composing systems thus happens at the level of applications. It is about providing support for component selection via matching the application's needs with the components that are available. It is not only about restricting the set of available components only to the components that perfectly match the application's needs, but also about knowing the implications that the other "non-matching" components might have to the system. Here, the system builder needs support in weighing the options and eventually modify the application's needs. It typically pays off to lower application's needs instead of developing a custom solution. With the overall robot system / application in mind, composition is thus application-driven rather than being only restricted to the technical level or to particular technical details (Fig. 3.9). An application is composed with needs that break down to particular needs of each component. This introduces new composability properties on an application-level to express the application's needs. Application-related composability is important between components and application, but also between components. It covers composability attributes that make components work meaningfully as intended by the application.

Application-related composability attributes are, for example, semantical, non-technical w.r.t. interaction of components, and non-functional properties. They add a meaning to an interface on top of technical composability. They include, for example, qualities (localization accuracy, object recognition probability, etc.) and other properties (image resolution, language of speech interaction) of a component.

The more application-related the composability properties are, the more domain-specific and the less generic they become. For example, composability properties on the technical level (e.g. communication semantics, data types, latencies, communication Quality of Service) are generic and can be applied in any domain (e.g. object recognition, manipulation, localization). Application-related properties such as the accuracy of a location or recognition probability can only be used in specific domains (e.g. recognition probability can be used in object recognition domain, but not in manipulation and not in localization).

Considering application-related properties for composability is important for system composition since components might be suitable on a technical level but not with respect to the application-level. Defining and thus agreeing on the meaning of such properties is as essential as using these properties through the complete workflow.

Freedom from Choice

Providing means to apply freedom *from* choice [Lee10] rather than freedom *of* choice within a structure improves the composability of solutions built with that structure. Freedom from choice is a principle that supports in realizing the separation of roles and in gaining composability (section 3.3.2).

Freedom *of* choice is often favored over freedom from choice. Freedom of choice offers all available options, leaving the user to decide whatever suits the needs. It does not limit the decision space, thus leaving the whole variety of options to use at high flexibility to build a solution. There is, however, a high price to pay for this flexibility, since there is no guidance for roles towards composability and system level conformance: There are many options on how to implement, for example, a simple communication. These many options might result in non-composable behaviors when not everyone follows agreed and documented principles by discipline. Freedom of choice is an approach that is typically favored by academia.

Freedom of choice works well in case one can oversee the whole problem and solution. It works well when there is access to every element and level of detail to understand the system ("have it all in your head") and to influence the system by modifying each part when necessary. There are several situations where this is not possible or desirable, for example when taking another role's perspective or when new people join a project. What typically happens is that one starts to build around the existing parts that are not understood to fix issues or re-implement it, thereby messing up system consistency. This is a situation that will not scale to an ecosystem level and freedom of choice is the wrong way to go for an ecosystem.

Freedom *from* choice [Lee10] is a principle that positively limits the number of available options in order to provide guidance via selected structures, thereby removing unnecessary degrees of freedom (see [Voe13]). Selecting the appropriate structures comes with a high responsibility to identify where guidance is needed and how it can be accomplished without limiting the use of the structures for system design. Limiting structures might be perceived as a disadvantage, but is in fact a positive aspect in the overall scope. It provides guidance, gains composability and ensures system level conformance. A good structure is one that hits the right balance between as much freedom as necessary but still providing or enforcing the needed guidance. Thus, the separation of roles and composition in an ecosystem are the guiding principles to find this right balance and evaluate it; supporting freedom from choice is mandatory for system composition in an ecosystem (see [euR16]).

CORBA and UML in general and ROS in specific for robotics are examples of freedom of choice. They do not enforce any decisions with the consequence and the risk of coming up with conflicts with respect to composition due to taking different implementations or assumptions. For example, ROS provides the "maximum flexibility, with nothing prescribed or proscribed" [Ger15]. Even though ROS is robotics-specific, there are only very few structures that enable the exchange of software and structures for important aspects are missing. CORBA as a general-purpose "swiss army knife" for communication in any distributed system on the other hand, cannot address individual needs of each of the domains in which it is applied. CORBA is a perfect toolkit for communication middlewares for experts in distributed systems, but it is not suitable for robotics experts that do not want to—or should not need to—deal with these kind of details.

3.3.2 Composition Workflow

The composition workflow is the *activity* of putting together building blocks. The workflow defines the steps and order to bring together all participants. It has to address their individual needs for system composition (Fig. 3.10). Stakeholders supply and use artifacts, e.g. provide or use components for composition. This requires prior alignment of what is provided and what is expected: functional boundaries, interfaces, and other necessary information. Collaboration within the workflow includes handover of these artifacts between stakeholders and workflow steps while ensuring, managing, and maintaining composability (section 3.3.1) during the workflow.

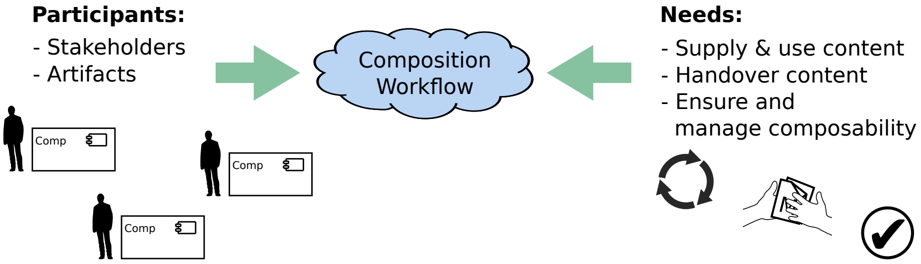


Figure 3.10: The composition workflow brings together all participants and their needs for system composition.

As previously argued, collaboration and composition in an ecosystem must be addressed by structure rather than by management. The objective of the workflow is to allow to create and to use a structure for system composition. The workflow defines the roles and artifacts and the according steps for composition. Individual software development processes (such as e.g. Scrum, Unified Process) or other methodologies (e.g. Software Product Line (SPL)) can be applied within these steps to, for example, develop building blocks (see sections 2.2.1 and 3.2).

Stakeholders in an Ecosystem

Finding a workflow that defines and uses a structure requires to understand its stakeholders. The two main stakeholders in the ecosystem (Fig. 3.11) are content suppliers that provide building blocks and system builders that use them to compose applications. Even though there is a connection between them, they do not necessarily work together as a team or even know each other.

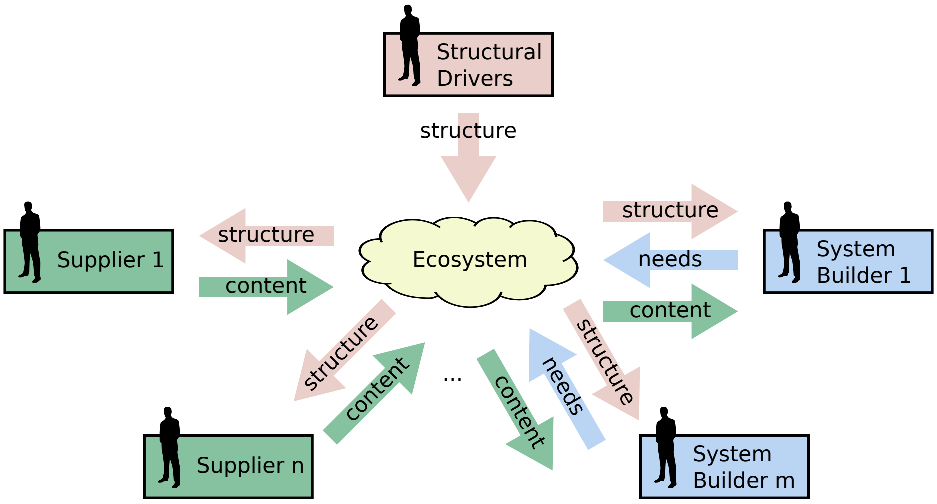


Figure 3.11: Stakeholders collaborating and interacting in an ecosystem.

The decentralized distribution of participants in an ecosystem is a special challenge [Bos09], since they need some way of coordination. This coordination is typically covered by development processes that they apply and communicate with. An overall development process, however, is not possible in an ecosystem (see section 3.2). In a compositional approach, coordination and agreements must be introduced via structures to which all the participants adhere to. Besides content suppliers and system builders, this brings a third stakeholder: the structural or ecosystem driver, which shapes a basis for collaboration, giving suppliers and users a common ground to build upon.

Content Suppliers develop building blocks for robot abilities as a solution for particular problems to provide them to others. They eventually reuse other building blocks. Suppliers have deep expert knowledge in a specific (sub-)domain and for example want to provide solutions for object recognition, speech synthesis or mobile manipulation.

They want to rely on the given structure so that their contributions can be used as-is by others. They want to be free within this structure to address the solution their way. Suppliers want to know what is needed by system builders (technology push vs. market pull). They have an interest in describing what they provide and how this distinguishes from the contribution of others.

Content suppliers will later be represented by the component developer role (section 4.1.2).

System Builders are the main "users" of the ecosystem. They build new service robotics applications from building blocks. They have knowledge of a certain application or of a complete domain of applications in which a robot shall operate. For example, knowledge in a service robot that assists in a house-hold or executes fetch-and-carry tasks for hospital logistics.

System builders do not have the knowledge in each particular robot ability or do not want to spend the effort to develop each part of the application on their own. They do not want to look inside building blocks to understand them or even modify them but still use them. They know the abilities and their qualities that are needed for the application. They might be overwhelmed and even lost by all the available building blocks and want guidance in finding a suitable one (see also: the tyranny of choice in psychology [Sch04b]). They want to know if their intended composition will work, prior to running it.

System builders will later be represented by the system compositor role (section 4.1.2).

Structural Drivers shape or define the structure of the ecosystem. They provide guidance for the contribution of content. Within such a framework, all suppliers and system builders can rely on stable structures. They can work within clear boundaries and their building blocks can connect through the defined interfaces.

They must find the structure such that it meets the "sweet spot" between being open to allow freedom for the supplier to fill in content and constituting a fixed and reliable frame setting the basis for technical collaboration. This structure might change as ecosystems will evolve [OW13] to adapt to new technology, so it must remain flexible, or must allow for creating alternative structures that might then be adopted.

The structural drivers close the gap between suppliers and system builders. They need broad knowledge of the (sub-)domain. They also require an understanding of the required abilities and the technical landscape that covers them. The drivers thus cannot be a single organization (cf. [IL04]). It must be a union of organizations to ensure to find the right structures for robotics.

Structural drivers will later be represented by the service designer role (section 4.1.2).

The position of the stakeholders is not static (cf. [IL04]) and might change. System builders, for example, might also supply whole applications or parts to the ecosystem for use by others, thereby becoming suppliers. Even further, one might create an own platform with own sub-suppliers, for example for object recognition where others can contribute concrete algorithms to detect objects in order to fuse the results.

Following Oster and Wade [OW13], system builders should or will be closer to ecosystem drivers. They argue that system builders are in close contact with the robot "end-user". They provide the overall solutions and thus know the requirements and the needs for new technologies best. They can thus form a technology pull.

Separation of Roles and Concerns

Separation of concerns [Dij82; Erl08] is a principle from computer science. Its aim is to identify different problem areas and to divide them into distinct parts such that one can look at them and solve them independently at a lower complexity level. It is an important principle for software development (cf. [ITE09]). Separation of roles [euR16] is closely related and achieves such a separation between the involved stakeholders. It is about identifying the roles and their task, setting their responsibilities (positively defining the freedom but also the limits) to ensure that they can collaborate, i.e. to organize the handover of parts and to ensure that the individual parts they contribute will finally not only fit together but also will work together.

Separation of concerns is focused on the individual problem (e.g. communication, data structure, business logic) of a design or implementation. Separation of roles is focused on how the stakeholders influence and contribute to the problem or design. Identifying and defining roles and taking their point of view helps to identify and separate the concerns.

Considering separation of roles and separation of concerns throughout the complete workflow is an important aspect to help designing the approach in such a way that all involved stakeholders can find their place of contribution. With separation of roles and separation of concerns, there is no need for a stakeholder to be an expert in every field. He can just focus on his role and his field of expertise to contribute with maximum efficiency, lower effort and thus lower cost. The role-perspective helps to answer what information or views are necessary where and helps to find the right abstractions and level of detail or information. Separation of roles and concerns is one of the key properties for robotics software development (cf. [euR16]).

Horizontal and Vertical Separation of Roles in the Workflow

The stakeholders in the ecosystem take over particular functions in the workflow, thus "play" or "cover" particular roles. Roles have to be separated such that they can work independently but still organize the information and artifacts that are exchanged to finally form a composition. This must be achieved by defining and managing the handover between the roles. The separation of roles must be organized on a horizontal, vertical, and lateral axis (Fig. 3.12).

Horizontal. The horizontal axis describes roles that contribute within the same workflow step. The computer vision and navigation communities, for example, each drive their particular domain in the ecosystem and consist of vision experts and navigation experts that supply components.

Vertical. The vertical axis describes the roles that work on the different steps in the workflow: for example, the definition of the structure by domain representatives, the contribution of components that follow these structures and using these and other components to build an application.

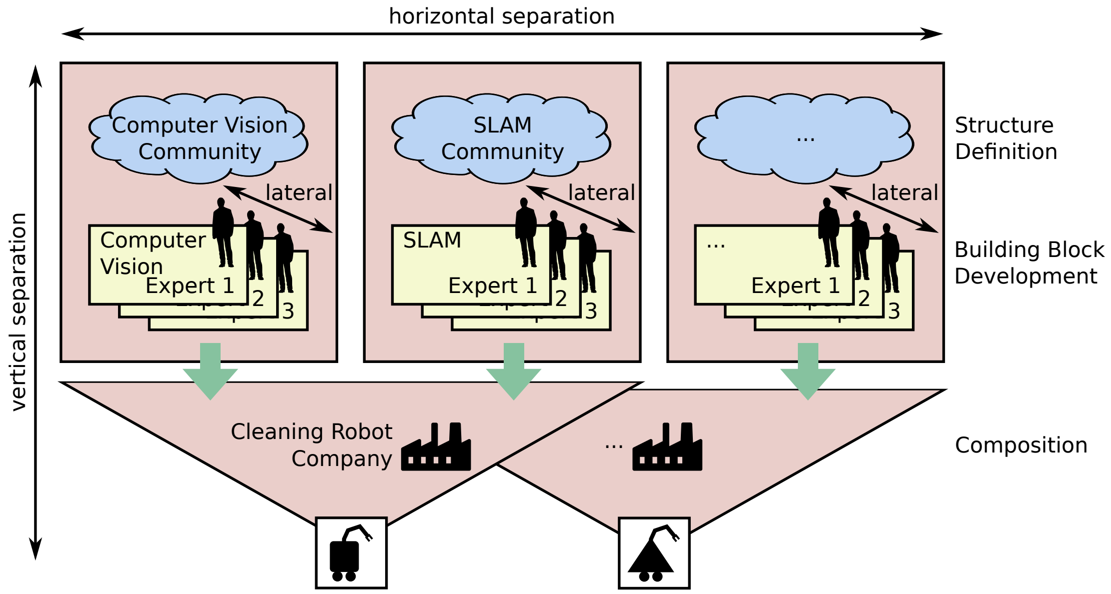


Figure 3.12: Horizontal, vertical, and lateral separation of roles is needed for system composition.

Lateral. A lateral axis can be drawn for the suppliers. The workflow needs to separate and organize the contributions of expert organizations that work on alternative solutions (e.g. localization solution A and B). These need to be identified as alternatives and need explicated distinctions (e.g. accuracy) such that system builders can choose from the existing building blocks. The lateral axis also needs to organize components that need each other to work (collision avoidance shall work with motion execution).

Assisted Component Selection and Management of Component Alternatives

Component selection is the identification of components among the set of all or all available components that suit a given need for the application under development. A structure for system composition must support stakeholders in selecting components. In context of this thesis, component selection is considered a manual step by the system builder who needs assistance and support by according tools and mechanisms. It is not intended and not considered feasible to *automatically* select—and even assemble—the whole system based on the explicated needs only.

For component selection, it requires that relevant information is available which can be used as a basis to decide whether a certain component matches or does not match the needs: What is the purpose of a building block, i.e. what service does the building block provide? This comes back to the definition of the interface and its description. What level of service, properties, or performance does the building block provide? This refers to application-level information (see section 3.3.1) and is of special interest to "component alternatives" (see section 3.1.1).

An alternative building block in general is something that suits the same purpose as another building block. Alternatives thus have a common ground to make them stand out as exchangeable building blocks in a set of more elements. On the other hand, alternatives have distinguishing properties that make them relevant alternatives; i.e. not all alternatives are the same. For example, two components might provide localization. This is their common ground and same purpose for which they are used. This makes them alternatives. But they might differ in the way they perform based on their internal solution, implementation or used technology. For example, high versus low localization accuracy. This "diversity of performance" makes one alternative more suitable than the other based on the needs of a particular application.

For system composition, a workflow must support the stakeholders in the creation of component alternatives and explication of their diversification attributes. It must support system builders to define their needs so that they can identify and choose the component from the set of alternatives that suits their needs as expressed in the workflow. For example, to select the localization solution that provides a sufficient localization accuracy. Without support in choosing the right component, system builders will end up with incomposable components or will end up with components that do not match the application's needs in an adequate way, e.g. in terms of quality, resources, and performance.

Workflow Dimensions

The best case for system composition is to build the application 100% out of existing building blocks. This, however, will rarely work in practice since every application will have specific needs that were not addressed before or some building blocks do not yet exist. The latter is the case especially while ramping up a platform or ecosystem. Typically, general-purpose parts of the system can be reused while specific parts need specific solutions (cf. [Frö02]). Specific solutions often are not reusable in other applications because they are not required in other applications. Two directions or dimensions can be identified for which the workflow should be suitable:

- Building a system by developing all building blocks from scratch: from system-level, top down to the specification of building blocks (decomposition).
- Building a system by only composing existing building blocks: from building blocks, bottom up to the complete system.

Both dimensions are extremes. For both dimensions, it is important to know where to stop in terms of granularity: In the ideal case, only as far as another role is concerned since all further details are in one's own responsibility. Coming up with building blocks is necessary to build and establish both the structure as well as the content (building blocks) for an ecosystem such that it enables the composition of existing parts, allowing to get from parts to systems. In practice, most activities will find themselves in between the two extremes. Thus, both need to be supported for successful system composition.

3.3.3 Support via Tooling

Support can have many forms. Adequate support in terms of tools for participants is critical towards system composition in an ecosystem (cf. [OW13]) as illustrated in Fig. 3.13.

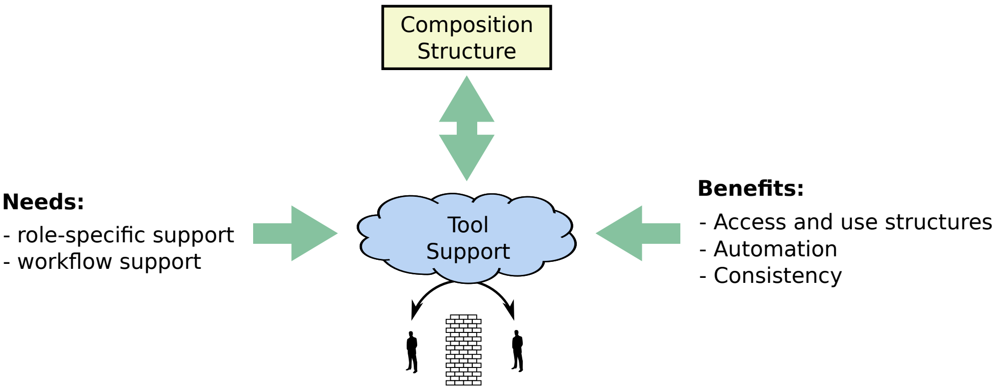


Figure 3.13: Adequate support via tooling for participants is critical towards system composition in an ecosystem.

Tools support in accessing and using the ecosystem by ensuring that parts adhere to its structure. Tools will realize the underlying structures of the approach and utilize them to prevent errors and provide automation, thus speeding up the development. Without adequate support by tools, participants of the ecosystem have a hard time "accessing" the methods and concepts. These concepts thus remain unused or are used in the wrong way, causing less acceptance and even leading to decreasing consistency (cf. [BB10]) and assets that cannot be composed. Tools play an important role in applying freedom from choice. Tools lower the effort, realize the handover, and realize the link between the different steps and participating roles of the composition workflow.

To realize separation of roles, tools must provide role-specific support for the stakeholders to use it in the way as defined by the overall composition workflow:

Workflow Support. Tooling must organize the steps that are necessary for composing a system. It must realize a smooth handover of the development artifacts between these steps. A workflow may be defined in "best-practice documents", but tool-guidance supports and ensures that the workflow is applied in the way it is intended by the structure to ensure composability.

Role-Specific Support. Tooling must establishes adequate representations and views that are necessary to support each participant in fulfilling a certain role. Views must be shaped for the task or activity and its required abstraction, for example modeling a component versus using a component. Views and abstractions should only present necessary information, allowing only changes within the role's function or concerns.

Building adequate tools needs proper foundations. An approach for system composition must provide a stable structure in a conceptual model that can be implemented in tools. Working with machine-readable models allows support for participants in the ecosystem by partial automation, e.g. by checking the models for consistency, providing code-generators or realizing support for selection of a suitable component for composition.

Software development in robotics so far uses a wide variety of tools: standard tools such as editors and compilers, and collections of tools that are tailored to the domain of robotics as for example reported in the "Journal of Software Engineering for Robotics (JOSER)" [Joser] and "Domain-Specific Languages and Models for Robotic Systems (DSLRob)" workshop series [DSLRob]. Dedicated tools support in addressing particular problems, but cannot address the overall systems engineering challenge with system composition because of the missing link and the missing handover between the involved steps but also because of the cross-cutting nature of composability. Especially towards an ecosystem and system composition, not only the existence of one or more tools, but an integrated approach is mandatory for its success. "Integrated tooling" means the seamless interaction between steps, views, models, and roles. It might even mean seamless interaction between different tools on the basis of models for the purpose of model handover. With respect to a composition approach, the tool and thus the composition structure that it realizes must be balanced, to work well and to work "hand in hand" through the workflow. For the integration aspect of the tool, this means that the parts of the underlying structure must be designed such that the parts of the tool(s) can interface via a clear interconnection [Völ11].

Consequences to Structure: Modeling Point of View

There is a close link between structures for system composition and Model-Driven Software Development (MDSD). On one hand, MDSD is the key enabler [euR16] that can express and thus realize these structures via meta-models. This section describes consequences on structures for system composition from a modeling point of view.

Finding a structure and models that provides the correct levels of abstractions is essential. The question is: Where to start and where to stop modeling? It is not necessary to cover all in-depth parts of a system via the composition workflow in models. While it may be possible to completely model a system, the advantage of having something represented by a model in the workflow must be carefully weighed in relation to the effort and additional complexity that may come with modeling. Trying to cover too much or too deep with models will lead to more effort and complexity, finally overweighing the initial benefit of MDSD. Structures for composition should focus on identifying the islands and key elements of the composition workflow and the robotics system which are critical to be covered. For example, it is not necessary or even feasible to cover the implementation of algorithms for robotics in meta-models. It, however, makes perfect sense to model the abstraction of the component that includes algorithms. Acting as a kind of digital data sheet, the models can then be composed on the level of services as services offer an adequate level of abstraction and granularity that is relevant to other ecosystem participants (see section 3.2.3).

MDSD is an enabling technology to realize separation of roles via applying freedom from choice. The following questions can serve as a guideline to shape and verify adequate structures for models that support composition in an ecosystem.

- What are the necessary elements and the information required for the composition of software components? What is their appropriate abstraction and separation for a representation in models?
- When in time and where in place will these elements and the related information be provided or used (during a workflow)?
- Who in person (role) with what knowledge is able to or is allowed to provide or use these elements and information?

Separation of roles is not only a key requirement for composition, but it is also a means to find and evaluate adequate composition structures. This is similar to architectural design where taking a concern-specific point of view improves the architecture [BCK12]. In analogy thereto, considering the workflow from each role's point of view supports in finding a suitable structure. Models or information that seems to belong together might be separated since the creation or use of the parts is separated in time and/or space by different roles or steps. For example, the definition of interfaces, their use in components for implementation and again the use of implementations in compositions happens at different points in time. It is important to identify which roles will interact. Bringing relevant information too late or not leaving room for information being filled in later during the development process may bring unusable applications (cf. [euR16]). It must be considered who (participant, role) shall provide this information when (workflow: when is this information available at all?). For example, variation points can be specified for a certain class of components and values might be assigned during their development. The concrete values of these variation points, however, will only be known when the component is being composed in a particular system. It is also important to identify which element carries what information. For example, it matters whether the interface of a component is defined in the component itself or externally (cf. [Bro+98]). This influences the composability of the component but also the reuse of the interface.

Applying all the guidelines presented in this section will bring structures and models with additional modeling complexity. This solution complexity, however, lowers the problem complexity as it separates the overall structure into manageable parts. These parts are tailored to the specific role and/or task at hand. As a result, however, this reduces the problem complexity for each single participant. This is one of the trade-offs in applying modeling and MDSD but is worth the effort and it is the only way to address system composition.

3.3.4 Summary of Consequences

This section first summarizes the current practice in robotics. It then summarizes the consequences on an approach for systematic engineering of software for service robotics based on system composition in an ecosystem.

Current Practice

In service robotics, reuse and exchange of software is being made on the level of libraries and code, for example as in MRPT [Bla], OpenCV [Its] as toolbox or even complete solutions such as Gmapping and others [SFG] for SLAM. A huge effort is required to integrate and to use these assets in a new robotic application. There is no systematic structure and modeling for integration or even composition in use. The dominance of code-centric and integration-centric approaches is not only the case in robotics—it has been identified quite some time ago for general software engineering [BB10]. Applications work since they are carefully designed and third party software is carefully integrated. To manage such a system, one requires access to and expertise of every area of the whole system which leads to tight coupling between involved developers. There exist assumptions by the individual parts that no longer hold true when reused in new applications. Pieces are ripped out of their environment, but the assumptions about the "old" environment still exist (cf. [GAO95; SW02]).

The Robot Operating System (ROS) is a representative and prominent example for software development in service robotics since it is the framework that is in most widespread use. Parts are packaged in nodes, but clear separation from other nodes as required for system composition is not possible. It is necessary to inspect the sources and wiki documentation to understand a node, for example its internal behavior and its assumptions with respect to other nodes (see [Ger15; Del14]). Only then, the node can be integrated into the new application. The lack of description of structures in meta-models and descriptions of interfaces of nodes independent of the implementation hinder system composition. ROS was intended for and is most used by academic institutions and research [Ger14]. This kind of users are technology experts. They are familiar with programming and willing to dig down deep into code. There is no immediate need for black-boxes—the advantage is the availability of accessible code that can be modified as required (see e.g. [Del14]). By purpose, there is "maximum flexibility, with nothing prescribed or proscribed (e.g., we don't wrap your main())" [Ger15]. In consequence, there is no support for separation of roles and composability on the conceptual level through structure. There is no Integrated Development Environment (IDE) for ROS [ROS11]; any preferred general-purpose IDE can be used [ROS] (see section 2.6.4). Design and implementation is thus not guided through tooling either, and it is in the responsibility of the developer to make nodes re-usable.

ROS pushed forward robotics through its widespread use thanks to its flexibility. It enabled researchers to exchange not only papers but executable code. It helped to identify that there is this need to share and exchange software. ROS seems adequate enough to build and demonstrate impressive service robot applications and robot capabilities. If software development for service robotics, however, does not go beyond the current level, robotics will not make the step change towards assembling systems from readily available parts as it has successfully been demonstrated in other domains, for example by the PC industry. Addressing this step change requires superordinate structures that enable system composition as addressed in this thesis.

Consequences on the Approach

The consequences on an approach for systematic engineering of software for service robotics based on system composition in an ecosystem can be summarized as follows (Fig. 3.14).

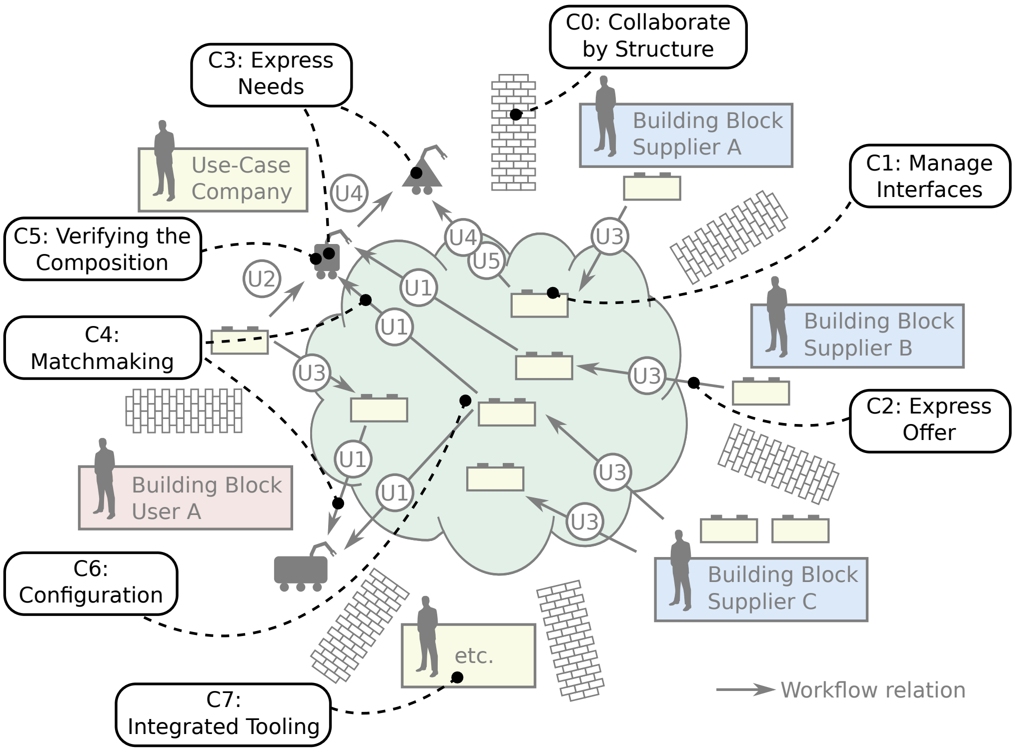


Figure 3.14: The consequences on an approach for systematic engineering of software for service robotics based on system composition in an ecosystem. This figure is a refinement of Fig. 3.4. Not all occurrences of consequences are illustrated to simplify the illustration and to maintain readability.

Collaborate by Structure (C0). The natural separation of roles in an ecosystem and thus the lack of common coordination for agreements requires to establish agreements by structure rather than by management. This is a high-level consequence.

Participants of the ecosystem must be able to establish and use structures within their domains to independently (in time and space) allow for supply and for composition of building blocks to applications. Such structures might evolve towards a (de-facto) standard. Providing means for system composition in an ecosystem is about finding a meta-structure that supports the ecosystem participants through horizontal, vertical, and lateral separation of roles via applying freedom from choice.

Manage Interfaces (C1). An “interface” is considered to be the “boundary across which two independent entities meet and interact or communicate with each other” [Bac+02]. Defining, managing, and maintaining interfaces plays an important role in guaranteeing that components can interact (cf. [GAO95]).

Means must be provided to define standard interfaces for components in service robotics. It is important for the definition of components to provide or require interfaces that rely on stable definitions. Speaking in terms of SOA, this is a “service contract”. A meta-structure for system composition must allow to establish domain-specific interfaces independent of the building blocks that implement or use them. The interfaces that building blocks offer and use must be at a certain level of abstraction, the service level. The service granularity must be of immediate use to an application that is being composed out of components, as components are the unit of exchange in the ecosystem.

Express Offers and Needs (C2 and C3). Component suppliers need a way to express what their components provide (C2). System builders who want to use these components need means to express the needs of their application (C3). Both is important to serve as input for later component selection (see C4) and for verifying the composition (C5). Expressing an offer or need is both about expressing (i) what is offered/needed (e.g. a localization service) and (ii) what are their according properties (e.g. qualities). Expressing offers and needs includes explicating syntactic and semantic properties on the technical level but also on the application-level. Both are relevant to maintain composability. These offers and needs must be managed through the overall composition workflow to ensure composability of software components.

Matchmaking (C4). System builders need support in choosing the one component from the potentially overwhelming set of available components in the ecosystem that matches the needs of their application. An approach for system composition thus must provide means for matchmaking between offers (C2) and needs (C3). The system builder requires automated support in component selection and composition. System builders must be sure to know which component is suitable for their application. On one hand, this means knowing whether a component candidate suits the application’s needs. On the other hand, this means knowing if the component will be composable to the other parts in the application.

A meta-structure for composition must allow the identification and handling of component alternatives (two or more building blocks that suit the same purpose) and their diversification (express and use distinguishing properties) for component selection.

Verifying the Composition (C5). It is desirable to support system builders in coming up with systems where it is known that they are most likely to “work” even before testing them: The composition shall be correct by construction. This does not aim at making testing obsolete. Systems that are made for the real world must be tested in the real world. But verification shall support the developer to prevent design errors as much as possible.

Verification is the “test of a system to prove that it meets all its specified requirements at a particular stage of its development” [ISO15]. Boehm [Boe79] considers verification as the correspondence between the software and its specification. Verification in context of this thesis is the check whether the system composed from components meets the expressed application's needs. The expressed needs must be tracked and must hold true during the whole development process, e.g. even after a component was selected during wiring and configuring the robotics application.

Configuration of a Building Block (C6). Building blocks for composition should be used "as-is". Each application in which a component might be used, however, is different. Most often, a component's standard settings are not suitable for the application in which it might be used. Component suppliers thus need a way to allow for expressing dedicated variation points of a component to allow modification from the outside of the component without modifying the component itself. Modification of a component, e.g. modifying an open source component contrasts with that. Using a variation point is an intentional modification that does not break the component in any way. A variation point is a variation that is foreseen already at design-time (see "bounded adaptivity" [Sch07] as cited in [BHA12]). Modifying the source code of a component for the purpose of configuration is not intentional ("open adaptivity" [Sch07] as cited in [BHA12]). It should not be foreseen in an ecosystem approach as these modifications are out of control of the component supplier and thus may trigger unwanted and unknown side effects.

Integrated Tooling (C7). An approach for system composition needs proper support for all participants in the ecosystem to apply it. Besides structures, an integrated tooling is essential for composable building blocks. An integrated tooling makes the ecosystem structures, concepts, and content (building blocks) for composition available to ecosystem participants. It guides them through the composition workflow.

An integrated tooling requires a balanced underlying structure that considers system composition as a cross-cutting concern to provide seamless interaction between steps and views while maintaining composability. The tooling must manage the creation and use of a structure and all involved artifacts and roles for system composition (service-level interfaces, building blocks, and compositions, including documentation) on the model-level and using MDSD and Domain-Specific Languages (DSLs).

The next section addresses the consequences C0–C6 and considers possible solutions. Consequence C7 is considered in chapter 6. All consequences must be considered in context of composability, the composition workflow, and support for participants of the ecosystem. It must be noted, that a particular consequence cannot be solved in isolation as all of them have strong influence on each other and depend on each other. The challenge, thus, is to come up with a consistent structure that addresses and carefully balances all these consequences.

3.4 Considerations

The consequences on an approach for system composition were described in the previous section (see summary in section 3.3.4). The remaining sections of this chapter consider possible solutions to address each of the consequences C0–C6 (Fig. 3.15). Considerations with respect to the implementation of the approach within an integrated tooling (C7) will be discussed in section 6.1.

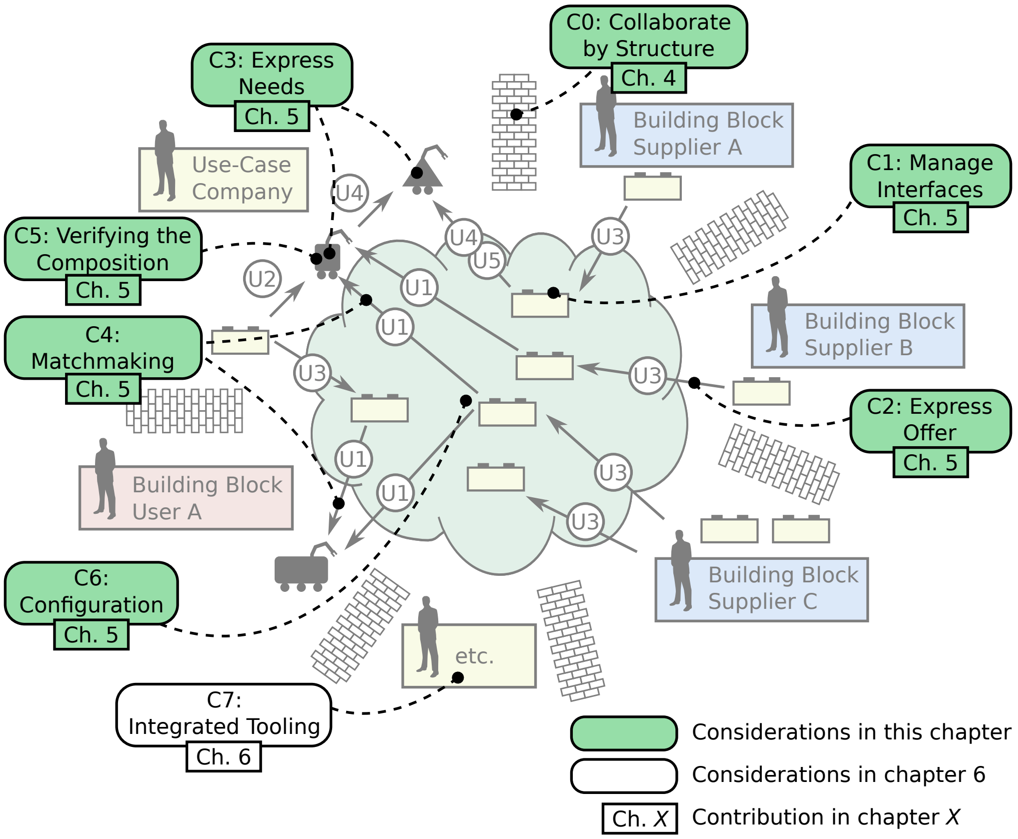


Figure 3.15: The consequences on an approach for system composition: The remaining sections of this chapter consider possible solutions to address each of the consequences C0–C6 (summarized in Fig. 3.15). Figure based on Fig. 3.14

3.4.1 Collaborate by Structure (C0)

The main consequence for system composition in an ecosystem is the need for structure (C0, Fig. 3.15). This section discusses very general approaches on which such a structure can build upon.

Service-Oriented Architectures and Component-Based Software Engineering

In Component-Based Software Engineering (CBSE), components shape the architecture of a system [HKF08]. In this thesis, the main benefit of components is seen in them being suitable as units of composition and exchange in the ecosystem where components come with Service-Oriented Architecture (SOA) services: Service-oriented components enable service-based composition.

The SOA [Erl08] concept introduces services as an appropriate level of communication between components. Services act as the main architectural elements to shape the architecture. Service-orientation uses low coupling and raises the abstraction from technical descriptions on an API-level to service-level agreements (e.g. provide “localization”). CBSE allows for separation of functional concerns of an application within components as the unit of composition (e.g. a specific implementation of a localization algorithm). Both SOA and CBSE are considered mandatory towards system composition as also argued in the ITEA Roadmap for Software-Intensive Systems and Services [ITE09, p. 278].

CBSE is state of the art in robotics software development (see e.g. Brugali and Shakhimardanov [BS10]). Among the existing approaches, the SmartSoft Framework [Sch04a] (see section 2.8) is the most suitable approach for system composition: It applies SOA and CBSE, it already provides basic support for freedom from choice, it provides basic support for separation of roles, and it provides a basic structure that can be expressed in meta-models. It is thus used as a foundation in this thesis. The SmartSoft Framework uses communication objects and communication patterns to separate what (data structure) is communicated in which way (communication semantics: e.g. push, query, send) between components. A communication pattern and a communication object together form the “service” of a component. These services separate the communication from the implementation of a component and therefore separate between a component-internal and component-external view. The clear separation between internal view and external view allows to consider components as black-boxes which is required for system composition. The small but adequate set of communication patterns limits the choice of developers positively, since all will rely on the same set instead of creating endless communication mechanisms that are all similar but not composable: SmartSoft already supports freedom from choice (see [Sch06]).

The SmartSoft Framework already provides a suitable baseline for system composition. However, it requires all developers using it to strictly adhere to policies in building components the way they are supposed to. This results in tedious manual effort and overall management in the workflow to craft all parts and agreements of the system. It requires coordination among all involved developers and system-knowledge. Finally, system-knowledge will end up in documents or even in source code and mismatches will be detected too late, possibly not before runtime.

The thesis builds upon SmartSoft (Fig. 3.16). It applies MDSD and formally expresses policies for ecosystem-collaboration in meta-models. It introduces a meta-structure for system composition (the "composition structure") that enables the organization of an ecosystem in three composition tiers (see section 4.1.1). A central part of the ecosystem organization are service definitions: They formalize the use of communication patterns and data structures of communication objects, thus manage the descriptions of services in a reusable way. They allow for working with application-related properties to enhance composability. Additional meta-models realize a composition workflow that enables handover of building blocks between different collaborating roles. Adequate tools support the roles in creating and using domain-specifc structures. This will guide roles through the composition workflow.

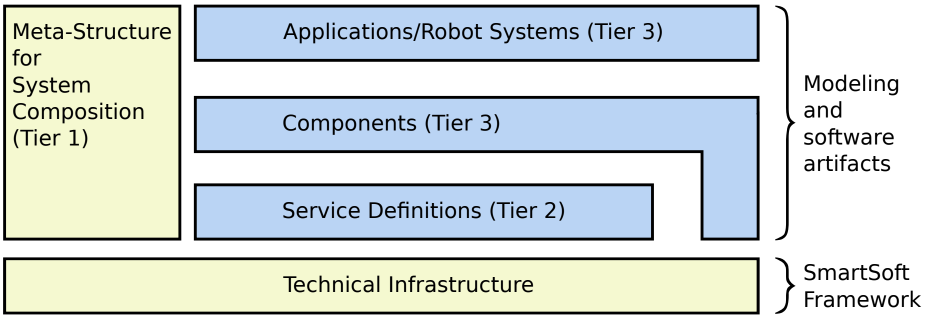

Meta-Structure for System Composition (Tier 1)
Applications/Robot Systems (Tier 3)
Components (Tier 3)
Service Definitions (Tier 2)
Technical Infrastructure
Modeling and software artifacts
SmartSoft Framework

Figure 3.16: The thesis applies the service-oriented and component-based framework SmartSoft and manages a meta-structure for system composition and artifacts via Model-Driven Software Development (MDSD).

Even though this work is based on SmartSoft communication patterns and the concept of services, the approach presented in this thesis can also be applied to other robotics frameworks or robotics development approaches. The SmartSoft Framework comes with a specific reference implementation, but the underlying concepts of the "SmartSoft Approach" are generic (see section 2.8). System composition is a challenge to collaborate in structure with separation of roles while maintaining composability. Its realization and implementation requires adequate underlying structures as a baseline. As argued before, SmartSoft is such a suitable baseline. This thesis can thus also be applied to other robotics frameworks or robotics development approaches that provide an equivalent baseline or that support the adoption of such a baseline. If such a baseline does not exist, the target approach is not suitable for system composition in the same way that Windows in its baseline is not adequate to support real-time capabilities and therefore should not be used for this purpose.

Software Product Lines

The concept of a Software Product Line (SPL) is widely adopted in software development to address reuse and to manage the various combinations of a family of related products in a systematic way (section 2.2.2). For this purpose, SPLs are an adequate structure for collaboration within an organization [HHJ08; RMM08; BB10]. Ecosystems, however, require approaches on the inter-organizational level [BB10] and SPLs yet must evolve to this level as also argued in the "ITEA Roadmap for Software-Intensive Systems and Services" [ITE09, p. 279]. Compositional approaches can be seen as the next logical step after establishing a SPL within an organization [Jan12, Part I][Bos09]. While there is no reason for a SPL to not span across organizations [Cap+14], SPLs aim at a product family that share a common architecture [BHA12; NOB11; Nor08]. There is not going to be a product that is not foreseen at design time [Voe13]. An approach for system composition should not rely on a specific architecture (i.e. be open ended) and should not need to foresee all possible variants in advance.

Because of this, the SPL is a decompositional approach in its nature [RMM08] and it works well when the domain is well known since the feature model must be specifically designed for the product family in a top-down fashion to explicate possible variations that cover all intended variants. This is not the case for composition in an ecosystem, where neither the possible contributions of component suppliers nor the applications that will be built are known beforehand. Further, management plays a crucial part in applying SPL approaches [Nor08] which is not feasible in an ecosystem approach (see section 3.2).

SPLs focus on managing the variants of "the whole". In contrast to that, a compositional approach focuses on the individual parts of a system. They must work with others without being developed together (composability). At development time, it is not known what other parts there will be in the final system.

SPLs are suitable when variants have very similar requirements [Ost14]. This is not the case for robotics in general. It can, however, be the case for component suppliers to manage variants of components. It also can be the case for system builders to manage variants of the robots they build. These are settings in which SPLs are considered quite useful in the context of this thesis since the SPL is applied localy within one workflow step or role.

Ontologies and Meta-Modeling

This thesis considers the use of ontologies for modeling the composition structures as an improvement of the traditional MDSD class-based meta-modeling. Ontologies are also considered to improve composability of domain structures. Ontologies will later be considered for matchmaking for component selection (section 3.4.4).

Modeling Composition Structures Meta-modeling and ontologies both capture the knowledge of a domain. Ontologies focus on providing semantics through relations between elements. Ontologies are of interest to this thesis to consider modeling the composition structures (composition Tier 1, section 4.1.1). Meta-modeling enables explicit typing and is thus stricter than the use of ontologies [Sil+10]. Ontologies are descriptive models, while meta-modeling enables specification models [AZ06]. Meta-models are mandatory to express composition structures that follow freedom from choice and that enforce structures. The benefits of ontologies and meta-modeling, however, are complementary [Sil+10; AZ06]. Applying ontologies and meta-modeling would allow to express semantics of the models explicitly. Semantics would be integrated with the ontology instead of implicitly encoded in tooling or in natural language descriptions that accompany meta-models (see [Sta+10; Mik+13]). This is a consideration that is to address for a formal implementation that is performant and consistent. This can enhance the consistency checking for models. Thanks to reasoning and classification, ontologies further can be used to come up with suggested modifications in case of invalid models to make them valid ("repairing" models [Sta+10]).

The first step to system composition is to come up with adequate structures and their semantics in natural language. Their benefit must be evaluated in a vertical implementation. Meta-modeling is suitable to formalize the structures for such a vertical implementation that demonstrates the benefit of composition. Applying ontology technology is considered a possible extension. The benefit of applying the complimentary concept of ontologies is not mandatory for this first step.

In practice, ontologies are only recently and rarely used for software engineering (see section 2.3.2). Tooling support for ontologies for domain-specific software engineering is rather low in comparison to the maturity, availability, and support of class-based metamodeling. Class-based modeling is much more matured and well supported via tooling, for example within the Eclipse modeling project (see section 2.4). From a practical point of view, this is a reason against ontologies since adequate tooling to realize structures for composition *is* necessary.

Improving Composability in Domain Structures Ontologies can improve composability of building blocks since they explicate the relations between elements and allow reasoning on them. Ontologies are descriptive models and can describe the building blocks. Meta-modeling enables prescriptive/specification models [AZ06]. The first step, however, is to come up with the necessary concepts and structures that enable system composition through enforcing and specifying superordinate structures. Meta-modeling is adequate to do so. Only then, the power of ontologies can be applied to them to describe and organize building blocks.

An ontology per se is the explication of domain-knowledge, that is, concepts and relationships of a domain. The main advantage to apply an ontology in context of composition lies in organizing robotics domain structures. Domain structures, for example, are definitions of interfaces or definitions of data representations/types. In context of this thesis, these are elements of composition Tier 2 (see Fig. 1.7 and section 4.1.1). Building an ontology would allow to describe the elements of the domain structures and possible relations between them. That is, ontologies can link "worlds" and mediate between them. Explicating a relation between two elements would enable the identification of elements that are equivalent, are equivalent by transformation, are a subset of each other, are not related at all, etc. Being able to express and use this knowledge about links of individual structures improves system composition. For example, components that are not composable will not be suggested for component selection (matchmaking), inconsistencies can be detected during design time or interfaces that are different in the first place might become composable through transformation of data representations. To give a very simple example: The representation of a color image and a greyscale image are two different representations. However, both are images. Even if expecting a greyscale image, one might use a color image with the appropriate conversion.

In order to build an ontology for robotics and system composition, one requires the abstract concepts that enable system composition (a meta-structure), before going into domain structures and starting to organize them. This thesis presents such a meta-structure for system composition ("composition structure" in short). It considers the whole composition workflow and involved roles. The composition structure enables the definition of domain structures, that ensure composability. On top of them, ontologies can be applied to express relations between domain structures to gain from the benefits as described before. Thus, this thesis paves the path towards applying an ontology on top of domain structures to express relations between the elements as the next logical step to further improve composability.

General-Purpose Modeling Languages

The Unified Modeling Language (UML) [OMG15b] is a multiple purpose modeling language that is very flexible and generic (see section 2.3.1). The flexibility and generality is one reason for its popularity in communicating and documenting software design, but also the main limitation to use it for building systems as it lacks semantics for the available elements. UML and its extensions (e.g. SysML [OMG15a], SoaML [OMG12b], RobotML [Dho+12]) favor freedom of choice. They provide different views, but the different views, models and parts of the system are not properly linked. They are thus not adequate to apply separation of roles which is mandatory for system composition. The proposed structures of UML and its extensions can, however, suit as inspiration or implementation basis wherever appropriate (see also [Bon+16]). For example, the system configuration model as well as the deployment model of the presented approach are inspired by UML and implemented using UML profiles.

The UML itself and its profiling mechanism is not considered a suitable *baseline* for structures supporting system composition. However, it is suitable to *implement* such structures: The SmartMDSD Toolchain utilizes UML profiles for graphical modeling, using synergies between the proposed approach and UML. For more considerations for tooling (C7) and a discussion of UML profiling, refer to section 6.1.

3.4.2 Manage Interfaces (C1)

An “interface” is considered the “boundary across which two independent entities meet and interact or communicate with each other” [Bac+02]. Organizing interfaces has a long history in component-based systems. The CORBA Interface Definition Language (IDL) [HV99] is a prominent example that describes interfaces in a language- and machine-independent way. IDLs typically focus on the level of an API with respect to subroutines, function calls, or remote procedure calls with tight coupling. With the raise of SOAs and loose coupling for flexible and distributed reuse, Service Definition Languages (SDLs) were introduced as advanced IDLs for services. The Web Services Description Language (WSDL) [W3C07b] is the most well-known IDL for web services. The WSDL is intended for the self-description of services with remote reuse. It is a machine-readable declaration of the interface to describe the way how clients can interact with a service and to generate code from this description, even to adapt dynamically to that service. As such, it is made for machines and not suitable to model services in terms of a DSL, but may be used as an underlying technological implementation. The WSDL separates the definition of services from the implementation of the communication endpoint (component port), which is also applied in this thesis. The WSDL focuses on the syntactic expression of an interface and thereby does not consider its semantics and (non-functional) properties. WSDL is used to model very fine-grained operations, comparable with an API while supporting loose coupling. Composition in robotics requires a higher level of granularity for interfaces way beyond the level of APIs. The granularity must be such that services and components are of any use for a broader set of robotics systems (see section 3.2.3).

This thesis proposes a lightweight SDL for robotics system composition and an according workflow to manage the interfaces between components on a service level. A definition of an IDL was given by Henning and Vinoski [HV99]. On this basis, this thesis defines an SDL as follows:

- Separate the definition of the service from its implementation
- Purely declarative: describe the interface but not the implementation
- Describe a service contract between provider and requestor
- Describe the syntactic and semantic aspects of the service
- Be lightweight and human-readable while at the same time be machine-readable through MDSD

3.4.3 Express Offer and Needs (C2 and C3)

Managing interfaces (section 3.4.2) is a first important step towards expressing the "technical" offer and needs. Interfaces alone are not sufficient as that requires looking up for details in data sheets, documentation or even testing a component to know about its behavior and assumptions. Such information must be expressed in a uniform way to be accessible by tooling to support the ecosystem participants.

Expressing offers and needs is also addressed by yellow-pages approaches to discover components. For example, the use of meta-tags [Zin05] and the use of ontologies [Zan+15]. This is of great benefit to enhance accessibility of robotics technology, but does not address composability as it requires integration and it requires ensuring that the component works as part of the overall system. The proposed approach thus focuses on composability by explicating properties and uses these properties to realize a yellow-pages approach for component selection as side-effect.

Matching offers and needs to enhance composability can be formulated as constraints on expressed properties. One of the well-known constraint languages is the Object Constraint Language (OCL) [OMG14] that is included in the UML. OCL is useful to model fine-grained constraints within individual models of a holistic top-down modeling approach. Addressing development with one holistic model contrasts with system composition and separation of roles. OCL is not intended for general architectural use. It is suitable for use in a dedicated model. For example, OCL can be applied to ensure consistency of values with the definition of a service's data structure or with the definition of a component's variation point. OCL cannot be applied for refinement through composition tiers (see section 4.1.1) as part of defining domain-specific structures. The approach presented in this thesis thus proposes a simple form of properties and their constraints for modeling and refinement through the composition tiers. This enhances composability of building blocks.

The responsibility of declaring relevant properties and their constraints is put to the modelers of the domain structure. They have the knowledge to come up with relevant properties. The responsibility also is put to the component supplier to assign useful and realistic values that are covered by the implementation. It is desirable to have automatic support to ensure that the properties hold true (e.g. through verification or deriving them from other models). However, having means to express them in the composition structure and use them through the composition workflow is a first major step. This first step already comes with an immediate benefit over the existing practice to not address this kind of information at all.

3.4.4 Matchmaking (C4)

In the context of SOA, matchmaking answers the question whether a service offer matches a service request [YL07] (see [Lud03; ZSK15]). In the context of this thesis, matchmaking answers the question whether a component—that can be retrieved from a component market as the unit of composition—offers a service that is needed for the robot application. Matchmaking for component selection in system composition in the end is a binary decision: A component or service either matches the expressed needs or it does not. Finding a match that is close to what is needed is not sufficient in terms of composability as there are interfaces that must match.

Kritikos and Plexousakis [KP08] distinguish two approaches for matchmaking: matchmaking based on ontologies and matchmaking based on constraint solving. Both will be considered in this section for the approach. It will turn out that a simple constraints-based filter is sufficient for component selection to support the system builder.

Use of Ontologies for Matchmaking

The classification and reasoning abilities of ontologies can be considered for component selection to find a component in terms of matchmaking for yellow-pages/directory approach. In robotics software development, the ReApp project (see section 2.6.3) has demonstrated this using ontologies based on a component's capabilities. In traditional CBSE, ontologies have been used for component selection as well [HKF08].

Searching and finding a component (e.g. based on capabilities), however, does not yet mean that the component is composable and will fit into the system—adapters and integration might be required. These adapters might be generated automatically to some extend, if an ontology was used to relate elements of domain structures (see section 3.4.1). However, ontologies as descriptive models can prevent selecting a component that will not meet the needs. Since ontologies are powerful to come up with hierarchies and relations, they can be used to implement a "yellow-pages" approach to find adequate service definition models: Ontologies can support the component supplier and system builder in finding service definitions to create the component model and to model the application's needs. The approach presented in this thesis would benefit from ontologies as an extension to the fundamental structures for system composition.

This thesis focuses on establishing a meta-structure that allows to express domain structures on composition Tier 2. It does not apply an ontology to relate elements found on this tier. The matchmaking is thus based on strictly matching explicated application needs (expressed as service definitions) with components providing the adequate services.

Constraint Solving for Matchmaking

Constraint solvers provide a solution to a problem that is specified through limitations (constraints). A constraint satisfaction problem is expressed by a set of variables, possible values that they can take, and constraints on variables that must hold true for a valid solution [RN03]. Matchmaking for component selection can be expressed as constraint satisfaction problem as shown by Hartig, Kost, and Freytag [HKF08]. Following their description, a constraint satisfaction problem for the approach presented in this thesis can be formulated as follows: The variables are the service definitions, the possible values that they can take are the component candidates, and the constraints are the expressed needs of the application.

Constraint solving is powerful to find the one or the few solutions in a huge solution space. The solution space for finding a component that matches the needs of an application, however, is small since the number of potential components is manageable. Applying constraint solving to matchmaking might speed up component selection, but speed is not a number-one concern.

The matchmaking problem for component selection is addressed by signature matching [Bac+02] (syntactic matching of the service definition) and constraint-based filtering [YL07] on service properties to prune service candidates of components that do not meet the needs of the user. This is a benefit to software development in robotics since so far there is very weak support for component selection.

Applying constraint solving for matchmaking is considered an extension for using the composition structure. It can, for example, support the system builder with suggestions for modifying an existing composition such that a certain component for selection fits into. Constraint solving can be applied to select multiple components at once such that they satisfy the needs of the application, but also the needs within the selected components. This enables coming up with a consistent selection. The current challenge in robotics, however, is to provide structures, means, and tools such that the system compositor is supported at all. Constraint solving for matchmaking requires structures to express the offers and needs in a consistent way. Only then the information will be accessible by tooling. This thesis provides such structures where more complex means for matchmaking, e.g. constraint solving, can build upon. The thesis thus uses the rather simplistic approach of a constraint-based filter to complete the vertical implementation. Even though it is rather simple, it already demonstrates a huge benefit in component selection.

3.4.5 Configuration of a Building Block (C6)

Composing components "as-is" triggers the need to adapt, i.e. configure, the selected component to the given robot system during composition-time. Configuration through the modification of source code is not an option (see section 3.3.4). There are two kinds of systematic configuration of components to be distinguished and will be discussed in the next sections:

Variant selection. Choosing sub-functionality or alternative functionality that is available inside a component. For example, select one specific algorithm in an object recognition component that can come with many algorithms[2].

[2] a run-time example that selects algorithms based on the expected objects is presented in [SLS12a]

Parameterization. Setting attributes in components. For example, to parameterize the component by assigning a value to a variable that holds the threshold for an algorithm. Another example is to modify the string holding the device name for the serial port via which a laser scanner component accesses the hardware.

Variant Selection through Feature Modeling and Software Product Lines

Selecting among alternative functionalities during reuse is addressed by variant management in the research fields of Software Product Line (SPL) and Dynamic Software Product Line (DSPL) (see sections 2.2.2 and 3.4.1). Feature models and (D)SPL can be used within a component for the purpose of configuration and adaptation. It depends on the granularity of variants and other factors whether the SPL is to be included into the component to select the variant at composition-time or whether the SPL is used to produce different variants as separate components for the market.

Fine-grained variants within a component may be suitable for selecting the final variant during composition-time. An example is selecting the object recognition algorithm during composition-time based on the expected objects, such as deactivating feature recognition as the objects come with a solid-colored surface. Higher-level variants may produce an entirely different "product" and better do not expose the SPL to the components in the market. In this case, companies might apply the SPL to manage the variants of components. For example, a company might want to produce several components for the market that have the same internal architecture such as the same feature recognition used in visual localization or in object detection. Other factors that might be involved in deciding on the application scope of (D)SPL are business factors or licensing considerations. For example, product diversification and a more expensive pricing of the "all-inclusive component" in comparison to a component that has exactly one feature.

Parameterization via Variation Points

Parameterization has successfully been applied for many years using traditional configuration-files, for example in Linux with various formats (ini, xml, json, etc.) or Windows with more widespread use of .ini-files. This can also be used to select variants. In a survey on variant selection during component reuse, Webber and Gomaa [WG04] conclude that using attributes and parameterization for variant selection works and is flexible, but remains complicated.

Parameterization addresses both the variant selection scenario and the setting attribute scenario. Setting attributes is a typical task in component reuse. The thesis thus applies parameterization as means for component configuration. Applying methods from SPLs for more comfortable systematic selection in a component is to be addressed in future work. Parameterization will be required anyway. Even without supporting comfortable means for variant selection within a component, variants can still be built by providing separate components for each variant.

Parameterizing all components in a uniform and common way is desirable for system composition and should be realized on the model-level. This enables the definition and reuse of a common set of parameters for a specific purpose, e.g. a family of exchangeable object recognition components adheres to the same set of parameters and thus also becomes composable with respect to parameterization.

This thesis uses groups of typed name–value pairs for configuration. System composition must come with proper support for parameterization that well supports separation of roles and composition. It must be powerful enough to demonstrate a usable vertical structure and implementation for system composition. This is supported by a yet simple but effective name–value pair parameterization that is established through the complete composition workflow. By experience, it already has proven its benefit for component configuration (see chapter 7). Later extensions and more powerful modeling of parameters, means for consistent modeling (by applying e.g. OCL), or even applying variant selection from SPLs can be built on top without changing the underlying structure fundamentally. Once having the parameters on the model-level and being able to manage them through the composition workflow enables to connect them with other models: e.g. deriving the laser ranger's serial port from the hardware model instead of manually assigning the value "/dev/ttyS0" by the system builder.

3.4.6 Verifying the Composition (C5)

This thesis addresses verification (section 3.3.4) by making sure that the expressed needs hold true with the composed system even during composing and configuring. This must be guaranteed even after all components were selected. Verification must include the needs that are expressed for the composition itself as well as the needs that are expressed by each component towards other components (see section 3.3.1). This rather simplistic approach is already a huge benefit to the system builder. It provides support to the system builder:

- In robotics, non-functional properties are not expressed and are hidden in source code: they are now expressed as application needs and component offers. Verification can make sure they match.
- The overall needs are distributed as each component brings its own needs: Instead of requiring the system builder to maintain an overview on all these needs, the tooling makes sure that they match or raises the system builders attention to a certain need that is in conflict.
- Needs may change during development: A component that was selected based on an outdated need may no longer be adequate.

Providing support to these challenges is a huge benefit in comparison with current development practices. They will ensure the consistency of the composition during design-time.

The expressed needs and properties are available on the model-level which opens new opportunities for more sophisticated verification methods. For example, the expressed properties can be used for evaluating the composition's correctness in terms of timing, performance, or quality in cause-effect-chains as described in [Lot+16]. The approach described there is in line with the general structures presented in this work and is part of the successor of the SmartMDSD Toolchain v2 (the SmartMDSD Toolchain v3 technology preview) that is presented in this thesis. The expressed needs can also be used for run-time verification, e.g. to monitor at run-time wether the current localization accuracy is within the design-time specifications.

3.5 Summary

This chapter has presented the vision of system composition in a robotics software business ecosystem. It has motivated the distinction between system integration and system composition that can be drawn by the effort to modify existing software. It has described how collaboration in an ecosystem works and has provided practical use-cases. The chapter has identified the need for structures and their importance in ecosystem collaboration. Such structures do not yet exist for robotics.

Based on the vision and the need for structure, the consequences of an approach for system composition in an ecosystem were derived and elaborated within three categories: (i) improving the composability as the ability of building blocks to be combined and recombined into different systems, considering the (ii) composition workflow that involves all steps and stakeholders to *do* system composition, and (iii) supporting the ecosystem participants through tooling to apply the approach and to provide guidance.

The chapter concluded by considering possible solutions that address each of the consequences. These considerations lay the foundation for the next chapter, which describes the organization of a robotics ecosystem in three tiers and the core elements of a structure for system composition. The next chapter provides a consistent picture of the contributions of this thesis to system composition in a robotics software business ecosystem.

4 An Approach for System Composition

This chapter outlines the main contributions of this thesis in providing the structures that enable system composition in a robotics software business ecosystem. The purpose of this chapter is to provide the overall setting and to explain the overall organization of a robotics ecosystem in three composition tiers. The chapter introduces the main elements of the approach and explains their interrelations. The next chapter (chapter 5) will go into the details of these structures and present the meta-models.

This chapter is organized as follows. The first section introduces into the approach in a nutshell. The next two sections describe the approach from an architectural perspective and from a workflow perspective.

4.1 The Approach in a Nutshell

This section describes the approach in a brief but consistent way to provide the overall picture.

4.1.1 Composition Tiers

The approach distinguishes three tiers for system composition in an ecosystem (Fig. 4.1).

Tier 1 structures the ecosystem for robotics in general. This tier provides basic structures for composition that are independent of the robotics domains. It is shaped by the drivers of the ecosystem that define an overall structure which enables composition and which is to be filled by the lower tiers. Tier 1 defines general concepts and meta-models for system composition. For example, the concept of service definitions, the concept of components, and the composition workflow that is tailored to service robotics. In terms of modeling,

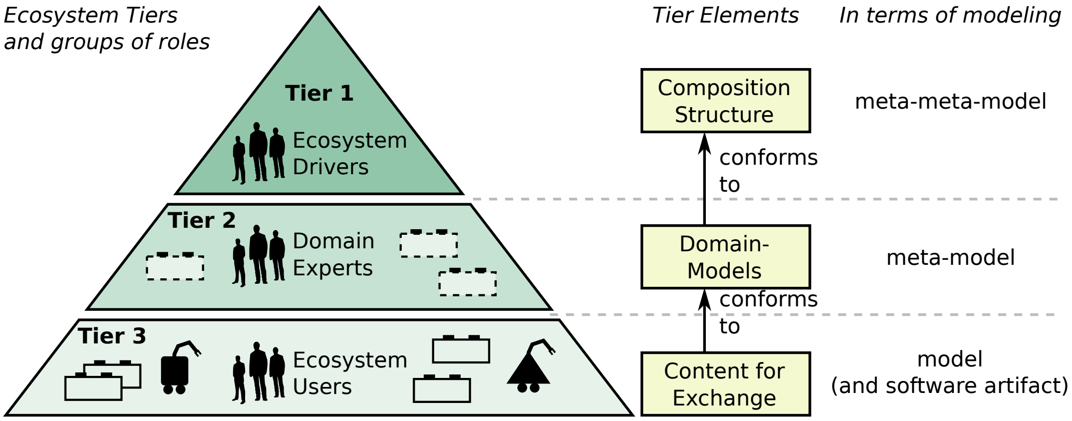


Figure 4.1: The approach distinguishes three tiers for system composition in an ecosystem. They are an abstract workflow for system composition: Tier 1 creates composition structures in general, Tier 2 structures robotics domains, and within Tier 3 users supply or use content based on these structures. The width of a tier in the payramid indicates the number of using/contributing participants.

this is the meta-meta-model level. The challenge within Tier 1 is to come up with superordinate structures such that they guide the domains in Tier 2.

Tier 2 structures the various domains within service robotics. Elements at this tier build a vocabulary and partition the domains. It is shaped by the experts of these domains, for example experts from object recognition, from manipulation, or from Simultaneous Localization and Mapping (SLAM). This is a community effort which structures each robotics domain by creating domain-models. Experts working at this level define concrete service definition models, for example a service definition for robot localization. In terms of modeling, this is the meta-model level.

Domain-models, for example, are "Service Definitions" that cover data structures, communication semantics and additional properties for specific services such as "robot localization". To find such a service definition, domain experts of each domain discuss and agree how to represent the location/position of a robot and what additional attributes are required and how they are represented (e.g. how the accuracy is represented).

Tier 3 uses the domain structures to fill them with content. Ecosystem users supply and use content, that is, models and the software artifacts they represent. This tier is shaped by the users of the ecosystem, for example component suppliers and system builders. They use the domain-models to create models as actual "content" of the ecosystem to be supplied and used. On this tier, for example, a concrete Gmapping component for SLAM that provides a localization service is supplied to a system builder to compose a delivery robot. In terms of modeling, this is the model level.

All consequences on system composition in an ecosystem (section 3.3.4) are cross-cutting through all tiers. They need to be addressed on a conceptual level in Tier 1 to make use of solutions for C2–C6 in the lower tiers (Fig. 4.2).

There is an analogy that illustrates the composition tiers with the PC domain. In the PC domain, many suppliers of building blocks can provide them independently of where or how they are used. This is thanks to (i) superordinate structures with standardized interfaces for clear separation between the inside and outside structures of building blocks and (ii) an abstract representation of the building block in a data sheet. For example, at Tier 1, the PC domain would place generic standards such as USB to connect almost any devices. On Tier 2, the "mass storage for USB" is defined as a domain-specific standard in the domain of storage devices. On Tier 3, a hard drive manufacturer can provide a particular product of a portable hard drive. For someone needing storage capacity, it does not matter whether the product is internally using a SSD or spinning disc. SSD or spinning disc, of course, might be a property that he evaluates while selecting the disc depending on his needs in order to narrow down the alternative choices.

4.1.2 Roles for System Composition

This thesis addresses the composition structure on Tier 1 by providing service definitions and an according composition workflow to enable the organization of the lower tiers. While this thesis contributes to Tier 1, the content of Tier 1 must be agreed within the overall robotics community and shaped by adequate representatives and committees. Several roles can exist within all tiers. This thesis focuses on the following roles as they are considered necessary for service-level composition of components (Fig. 4.2):

Service Designers (Tier 2) model service definitions. They are, for example, representatives of the robotics community (Domain Experts). Service designers are part of structural drivers in the ecosystem (section 3.3.2).

Component Developers (Tier 3) model and implement components. For example, this can be a Small and Medium-Sized Enterprise (SME) specialized in SLAM. They are the suppliers of content in the ecosystem (section 3.3.2).

System Compositors (Tier 3) build applications from components. This can be, for example, a startup which develops a delivery robot[1]. This role belongs to the system builders in the ecosystem (section 3.3.2).

[1] inspired by the profession of a compositor who used to put together physical types (letters and symbols) to words.

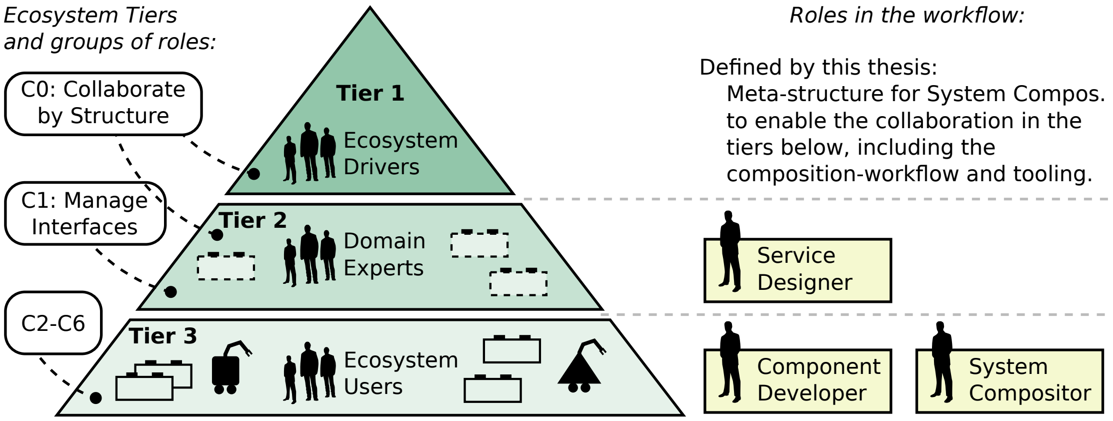

Ecosystem Tiers and groups of roles:
C0: Collaborate by Structure
C1: Manage Interfaces
C2-C6
Tier 1
Ecosystem Drivers
Tier 2
Domain Experts
Tier 3
Ecosystem Users
Roles in the workflow:
Defined by this thesis:
Meta-structure for System Compos. to enable the collaboration in the tiers below, including the composition-workflow and tooling.
Service Designer
Component Developer
System Compositor

Figure 4.2: The consequences on system composition and the roles addressed in this thesis in relation to the tiers.

There are probably more roles involved in developing a robot. These additional roles would group within the described roles of ecosystem drivers, suppliers, and system builders (section 3.3.2). For example, a role that develops action plots for task sequencing (task or behavior developer role as in [SS14b]) or the role of a performance expert (modeling the information flow through a chain of components to evaluate timing issues as in [Lot+16]) could be considered a content supplier besides the component developer role as described here.

4.1.3 Service-Based Composition Workflow

The approach presented in this thesis builds on principles of Service-Oriented Architectures (SOAs) and Component-Based Software Engineering (CBSE) (section 3.4.1). Service definitions play the central role as stable architectural elements that define the functional boundaries of building blocks (components). They ensure composability for system composition. Component and application development is based on these service definitions. Figure 4.3 shows the overall workflow.

- Service definitions as a concept (Tier 1) form the meta-structure for system composition (C0). Concrete service definition models are created by domain experts on Tier 2 using a Service Definition Language (SDL) to manage the interfaces of building blocks (C1). Domain experts can form a committee for a domain out of domain experts. For example, object recognition experts discuss and agree on service definitions as a standard for the domain of object recognition for robotics.

  A certain service definition model represents a common structure for a class of services and ensures that components (Tier 3) offering or using such a service are composable and can be used together. Service definitions explicate composability information that is otherwise hidden in documentation or code: communication data structure, communication semantics, and service properties that a service of a component following this service definition will later offer.

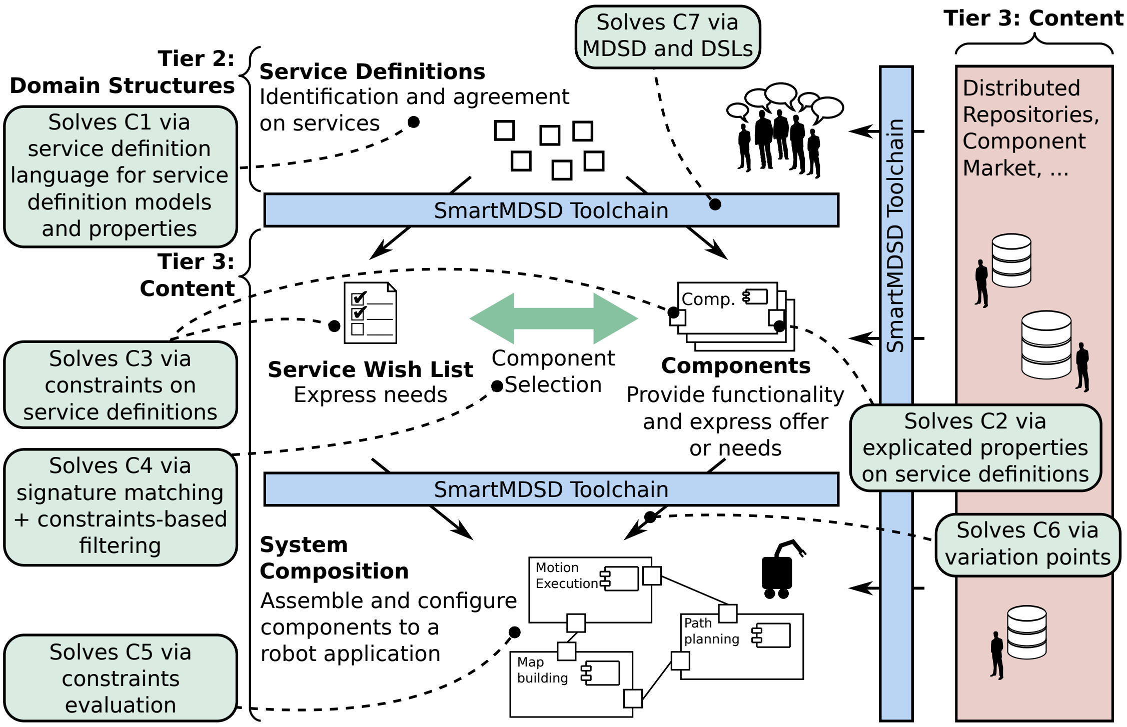

Tier 2:
Domain Structures
Solves C1 via service definition language for service definition models and properties
Tier 3:
Content
Solves C3 via constraints on service definitions
Solves C4 via signature matching + constraints-based filtering
Solves C5 via constraints evaluation
Service Definitions
Identification and agreement on services
Solves C7 via MDSD and DSLs
SmartMDSD Toolchain
Service Wish List
Express needs
Component
Selection
Comp.
Components
Provide functionality and express offer or needs
SmartMDSD Toolchain
System Composition
Assemble and configure components to a robot application
Motion Execution
Path planning
Map building
SmartMDSD Toolchain
Tier 3: Content
Distributed Repositories, Component Market, ...
Solves C2 via explicated properties on service definitions
Solves C6 via variation points

Figure 4.3: The approach for system composition based on service definitions. The consequences on the approach as summarized in section 3.3 are annotated in green.

- Component developers (Tier 3) model components as the unit of composition and the unit of exchange in the ecosystem via repositories and market places. A component provides functionality through formally defined services at a certain level of abstraction (cf. Szyperski [Szy02]).

  Component developers use service definition models to create service endpoints that communicate with service endpoints of other components. Service definitions align the components to the domain structures in Tier 2 to ensure composability (C1). Component services refine their service offer by using attributes on service properties (C2); they refine their service need towards other services using constraints on service properties (C3).

- System compositors (Tier 3) that build applications use service definitions to create a "wish list". The wish list expresses the needs of the robot application (C3) by using constraints on service definitions. For example, needing a "localization" service with a certain "accuracy".

- System compositors (Tier 3) have to find suitable components from third parties. They are supported in selecting components through tooling: Based on the selected services and specified properties (wish list), a system compositor can select components from the ecosystem that provide the needed services with according properties. Matchmaking is made based on offered services and on other properties, e.g. on the required accuracy.

It uses signature matching [Bac+02] and constraint-based filtering to provide a list of components that match the needs of the robot application (C4).

- Finally, the system compositor assembles, configures, and deploys the robot application. To ensure correctness by construction, the composition is verified by evaluating the constraints in the wish list and by evaluating the constraints that components expressed towards required services (C5). Components (used as they are, without modifying source code) are configured to the application by only adjusting explicitly modeled variation points (C6).

The challenge of a workflow for system composition is to manage the handover between the different activities and roles. On one hand, the approach addresses this by the carefully selected meta-structure for composition in general (composition structure, Tier 1) which is used to create domain structures (Tier 2) to which the building blocks adhere to (Tier 3). Secondly, the interaction and handover of artifacts between activities and roles happens at the model-level and is supported through the SmartMDSD Toolchain (Fig. 4.3), an Integrated Development Environment (IDE) for engineering software for service robotics, that implements the composition structures and the workflow using Model-Driven Software Development (MDSD) and Domain-Specific Languages (DSLs) to make them accessible and guide users through the workflow (C7).

4.1.4 Service Definitions to Support Separation of Roles

Service definitions act as a link between the roles and activities in the composition workflow and thus decouple their interaction (Fig. 4.4).

With service definitions, suppliers can identify the needs that emerge from the ecosystem (technology pull). On the other hand, application builders can identify the available solutions (technology push). As they can (but do not have to) take multiple roles, both can contribute and shape the structures to come up with accepted structures in the long term. This might lead to similar domain structures that co-exist. The approach, however, helps to identify them as similar which can help to align or merge them towards a (de-facto) standard in the long term. Such activities on Tier 2 have political impact that are beyond the scope of this work.

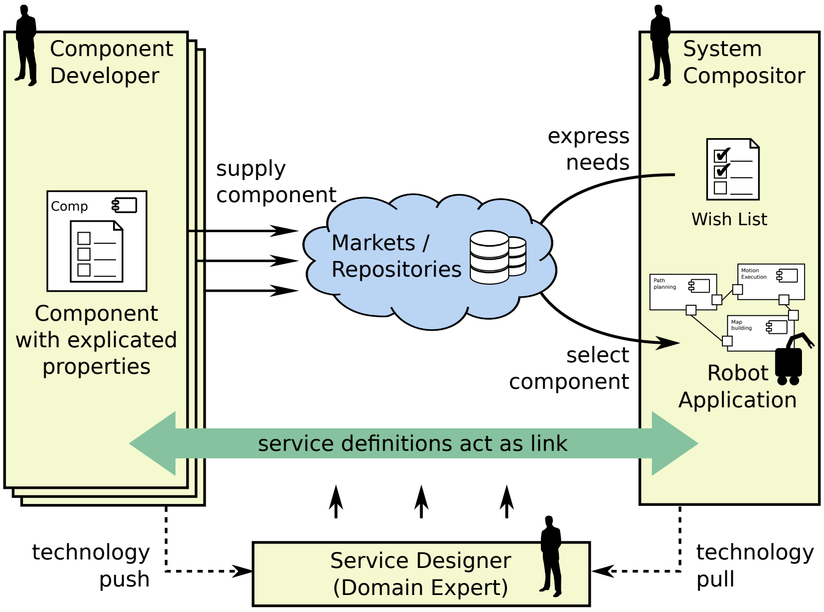


Figure 4.4: Service definitions establish the link between component suppliers and system builders.

4.2 An Architectural View on the Approach

This section explains the parts of the approach and their relations: the approach's architecture. It is the basis for the meta-models (chapter 5).

4.2.1 The Three Parts of the Approach

The meta-structure for system composition on Tier 1 can be categorized in three parts (Fig. 4.5): structures to create domain-specific models (structural building blocks), structures to model components (functional building blocks) and structures to model robot applications (compositions). Using this meta-structure for system composition results in the classic schema of definition, implementation and instantiation.

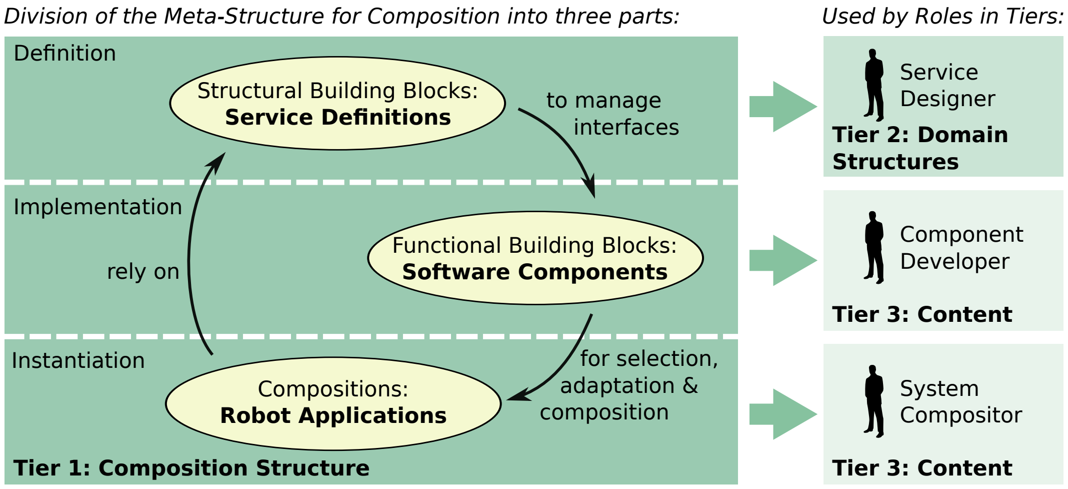


Figure 4.5: The meta-structure for system composition (Tier 1) is organized in three parts: structures to create domain-specific models (structural building blocks), structures to model components (functional building blocks), and structures to model robot applications (compositions).

Structural Building Blocks to create domain structures and models. They collect basic elements (models) that are important for composition. These models are once defined and available for consistent reuse in building blocks, thereby creating a structure for composition. Structural building blocks include the data structure for communication, service properties, and parameter sets for component parameterization. Service definitions are the most important element of structural building blocks. Service designers create service definition models using a Service Definition Language (SDL).

Software Components to Provide Functional Building Blocks. Means to model and provide content: software components that offer or use services. Components use domain structures (service definition models) to express their offer to other components—or their need from other components. They model variation points to adapt a component to the robot application. They provide means for documentation to users.

Compositions to Build Robot Applications. They provide means to select and put together components to a robotics application. They rely on domain structures to ensure composability.

The organization of Tier 1 in three parts (Fig. 4.5) has several advantages with respect to system composition. Since both the components and the composition are based on service definitions, the composability of components is increased. The separation of the three parts directly contributes to the separation of roles since every part can be addressed by a specific role (Fig. 4.5). It introduces different levels of abstractions that manage interfaces between roles and therefore it separates between roles, thus allowing for different views and finally managing the roles and contributions in an overall composition workflow. The distinction between the concerns "structure" and "content" allows for handling domain structures (Tier 2, service definitions) independent of the content (implementation/components and compositions, Tier 3) which will enhance the separation of roles. Since composition and implementation are based on formal service definitions, compatible alternative components that follow the same definition can be created. Further, the structural building blocks support in specifying application-related properties that allow to distinguish alternatives to ensure composability on an application-level.

Each of the three parts (Fig. 4.5) contains several meta-models that are provided for reference here: Figure 4.6 illustrates these meta-models and their relations. The figure also illustrates how the elements are linked to the consequences on an approach for system composition as summarized in section 3.3.4 and discussed in section 3.4. The remaining sections in this chapter focus on describing the meta-models and their relations. The concrete meta-models are presented and explained in detail in chapter chapter 5. Figure 4.7 illustrates the use and creation of models by the particular roles in the workflow.

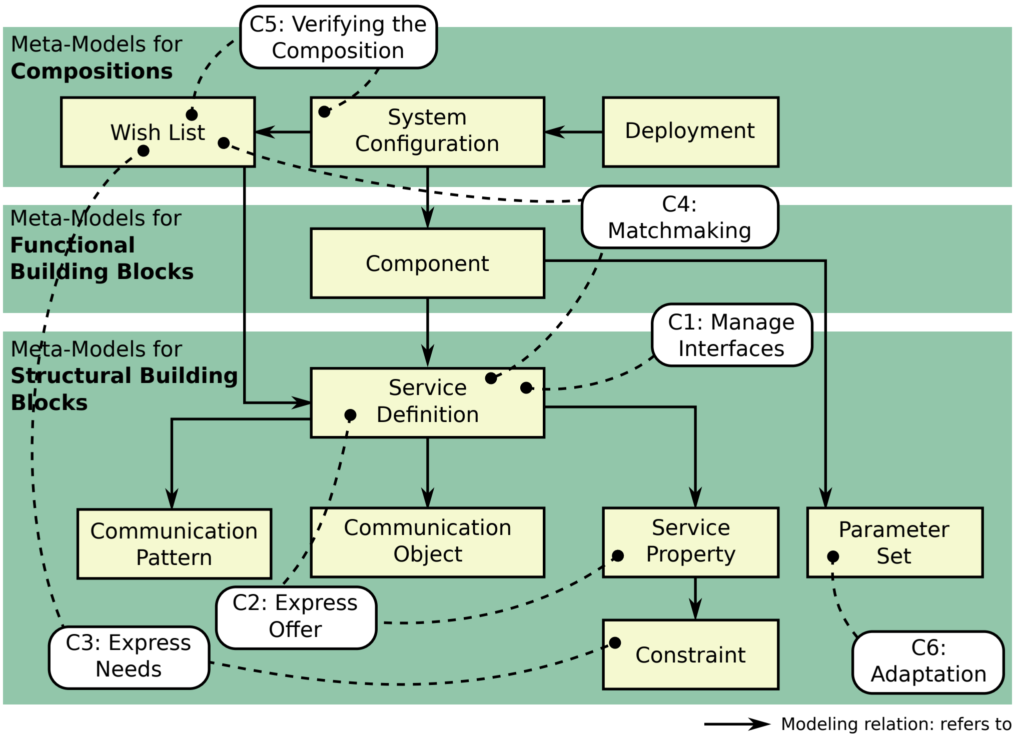


Figure 4.6: The main meta-models of the composition structure and how they are related. The white bubbles annotate their contribution to the consequences on an approach for system composition (section 3.3.4).

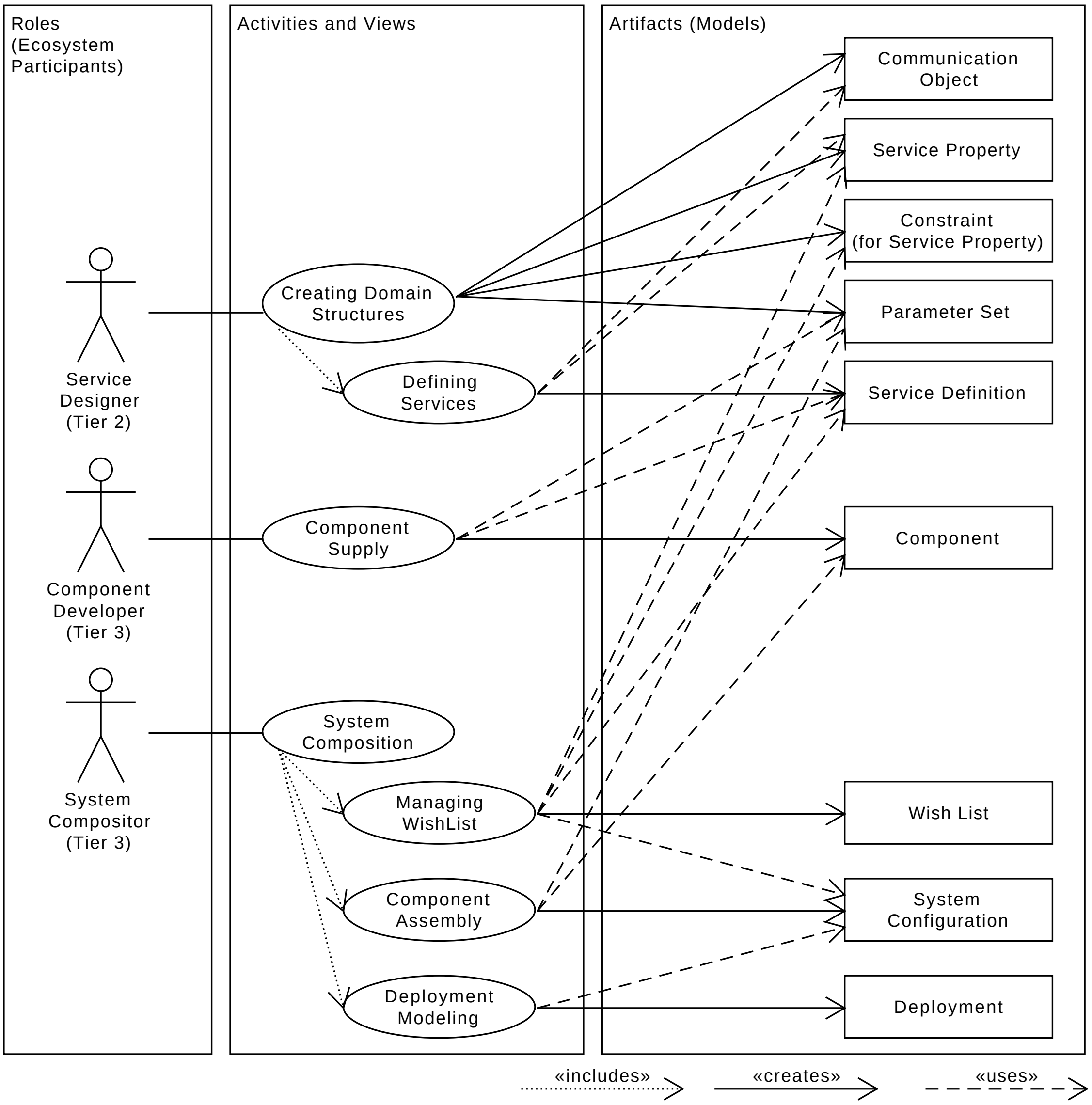


Figure 4.7: The roles of the composition workflow (left) and their interaction (middle) with the elements of the composition structure (right); illustrated in an UML use-case diagram.

4.2.2 Structural Building Blocks

Structural building blocks collect all models which structure individual robotics domains. They include all elements which are necessary for composition through the overall workflow. Service definitions are the most important element. They establish a structure on which content in the ecosystem can rely on, that is, software components as building blocks and the composition of components rely on service definitions.

Service Definitions

Service definitions are stable architectural entities and establish a vocabulary of services in a domain. They group together what is needed to describe a service as a whole. They bridge between functional building blocks and system composition. Service definitions are reusable formal descriptions for a whole class of services. Service definitions leverage the agreements between components, which often only exist in documentation or code, to a model-driven level. Service definitions capture the details that are necessary for communication and composition. They thus raise component interfaces from a technical perspective to an application perspective. Service definitions are created using an Interface Definition Language (IDL) for services: a Service Definition Language (SDL).

A service definition (Fig. 4.6) includes a meaningful name, the communication semantics (one selected SmartSoft communication pattern [Scho4a]), the data structure of the service and service properties. These parts are modeled independently to enable their reuse in other service definitions. Components will provide or require services and the service definition can be considered the “type” of that service (Fig. 4.8).

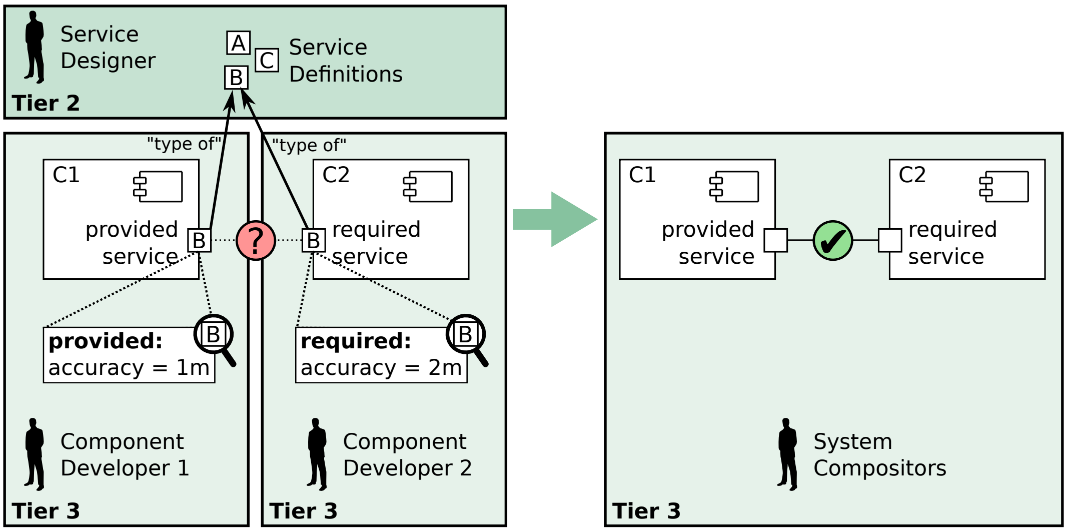


Figure 4.8: Components provide or require services; service definitions can be considered the “type” of such a service.

By instantiating service definitions in components, one can describe what services the component provides or requires including application-related and semantic details in service properties, e.g. provided localization accuracy. When composing components, one can rely on service definitions and use them to express the need of the application with respect to services. This enables the selection of components based on the needs of the application. It is no longer required to manually take care of compatible communication characteristics in components and specific data structures of their communication as these are captured in service definitions.

When two components use the same service definition to provide e.g. a localization service, they become exchangeable. Service definitions therefore enable "component alternatives". These alternatives might differ in quality (e.g. localization quality) or other properties (e.g. the representation of a location). These distinctions are expressed with service properties.

Service Properties

Service properties allow to express the semantics of a service on the application-level (section 3.3.1) to describe what is provided or required by a service. They provide a means to express application-related agreements and information that is important for composability beyond the technical communication-mechanism. Such agreements are, for example, localization accuracy, object recognition probability, recognizable objects, image resolution, language of speech interaction, or robot motion type.

Service properties allow for defining a domain-specific vocabulary. This vocabulary is used within a service definition. During component development and at system composition-time, the property is used to express the agreements from a required and provided perspective, respectively.

Service properties enable to determine the composability of two service endpoints between two components in the composition on a syntactic and semantic level. They enable this both within (i) the vertical axis between component and application and within (ii) the horizontal axis between components (cf. Fig. 3.9). Service properties are also used for component selection since properties express the diversity of performance of component alternatives (lateral axis).

Variation Points for Component Configuration: Parameter Sets

Components for composition should be used "as-is". There is, however, the need to configure a component to the concrete robot system in which it is used. For example, to change the setting of an algorithm, to change the grid size of a mapping component, to change the filenames of maps to be loaded, or to change device identifiers to connect to.

Models of parameter sets are a reusable definition of variation points. They are used by the component developer to express how the component can be configured at composition-time. During composition-time, the component can only be adapted according to the modeled variation points. Variation points are defined as reusable elements as part of structural building blocks and domain structures because the same set of variation points might be used within several components to configure them in a uniform structure (e.g. configuring diverse object recognition components in the same way).

4.2.3 Functional Building Blocks: Software Components

“Functionality is the ability of the system to do the work for which it was intended” [BCK12, p. 65]. Software components provide a container for the implementation of algorithms and serve as building blocks that “host” and provide functionality to the robot. Based on the definition of Szyperski [Szy02], the term “component” in this thesis is used as the unit of composition and exchange in the ecosystem. It provides functionality to the system through formally defined services at a certain level of abstraction.

Components can provide (supply) and/or require (use) services to communicate with one another. These component services base on service definitions which explicate how components will interact. Components thereby align themselves to the given domain structures. One component that requires a service needs another component that offers this service. Two components are composable if their component services adhere to the same service definition. Two components that either provide or require the same service definition, and thus can be exchanged, are called “component alternatives” (section 3.3.2).

The service properties that come with the service definition offer means to describe the semantics and application-related agreements of the service. When components provide a service, they use service properties to describe what the service offers. When they require a service, they use service properties to express their needs; what they expect from that (remote) service which is provided by another component.

Service definitions are the central architectural entities that form the structure of the system, but components are the unit of composition. The services that components provide need to have an adequate level of abstraction that is of use to the application (see section 3.2.3). Reuse of components is typically physical [Die02]); the component is a copy or an instance of the original one for each application in which it is used. This contrasts with logical reuse [Die02], that is typically applied in SOA and cloud-based approaches, where the service of the same component is accessible through a network and used by several applications. During system composition, the component model as well as the source code or binary is retrieved and instantiated.

The component model uses parameter sets to select variation points to express how the component can be configured. These variation points are bound at composition-time and are accessible from within the component’s implementation at run-time.

4.2.4 Compositions

Modeling a composition is about putting together components. For this purpose, the composition structures provide the wish list model, system configuration model and deployment model (Fig. 4.6).

The service wish list model selects services and instantiates service properties to express the needs of the application. For example, a mail delivery robot will need, amongst others, services for localization and motion execution. For both services, certain qualities are required such as location accuracy or the type of robot locomotion. The wish list provides the basis for component selection by signature matching of services and constraint-based filtering of service properties.

The system configuration model provides a software view on the system. Component instances are created in this model, the wiring between components is set up, and the components are configured using variation points. System configuration also utilizes the wish list for verifying the assembled system to detect problems already at design-time.

The deployment model creates a hardware view on the system and assigns instances of components to execution units. It is the basis to finally transfer the composed system to the target platform for execution.

Partitioning models for composition in wish list, system configuration, and deployment separates the concerns for compositions. This supports roles in focusing on one topic alone and provides the relevant views to the role: It separates the definition and selection of what is needed in the application (wish list) from configuration of how these parts work together (instantiating, wiring and configuring in system configuration) from the transfer to the robot (deployment).

While components themselves have a technical view (solving a particular problem), system composition takes an application view: The goal is to create a system for a particular application and therefore all its needs are known. Components are viewed from the application view: no need to know about internal details (black-box) but with knowledge about the outer structure (services that they realize along with their qualities). Thanks to service definitions, one can think of just "localization" and its quality without needing to know about its internal realization.

4.3 A Workflow View on the Approach

4.3.1 Workflow Steps

The workflow maps the overall composition structure to concrete workflow steps and puts them in relation. It considers the transitions between the workflow steps and the roles that stakeholders take when creating or using the approach. Even though "step" implicates a clear temporal order, service definitions decouple the individual steps and the workflow can thus be applied separated in time and space (see section 4.1.4).

Figure 4.9 shows the main elements of the workflow: system design, development, and composition. The workflow starts with modeling service definitions as the central elements of the workflow that also enable the handover between the steps. The component model is created by defining the component's hull using service definitions. Based on the modeled component hull, the user can integrate user-logic or libraries. Component alternatives and components that interact with each other rely on the same service definition.

Service definitions realize the horizontal, vertical, and lateral composability and enable the separation of roles through the workflow steps. They contribute to applying freedom from choice. Service definitions provide stable interfaces for services and build the foundation of the composition workflow for component developers and system compositors. Without stable services, component developers could design and extend their interfaces as they wish, thus breaking the composability of components. Changes in service definitions have direct impact on the functional boundaries of components: The modification of a component's service directly

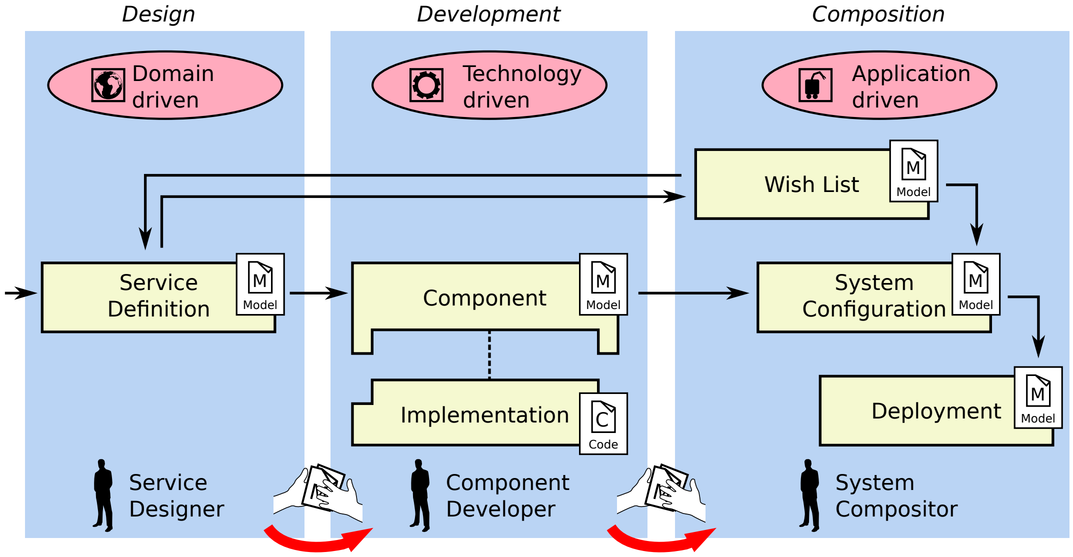


Figure 4.9: The main workflow consists of three steps that are linked via service definitions: design, development and composition. Even though the figure implicates a clear temporal order, service definitions decouple the individual steps and the workflow can thus be applied separated in time and space.

influences the system composition and thus may break the system architecture. There is, thus, no way for component developers to arbitrarily modify a component's service: The component developer can decide which services to offer, but cannot break a service. Such changes require the mutual agreement of domain experts at composition Tier 2. These changes must be made in the models of the design step and then need to be propagated through component modeling as illustrated in Fig. 4.9. Service definitions also enable matchmaking for component selection: They are used in the wish list to express the application's needs and are used in the component to express the component's offer.

The described steps in the workflow do not necessarily map to software development methodologies such as the V-Model, Unified Process or Scrum. These methodologies, however, can be applied within each step of the workflow in an ecosystem (e.g. within component development). See section 3.2 for more details on the composition workflow with respect to software development processes and collaborative development tools.

4.3.2 The Composition Workflow: Intra-Organizational Perspective

So far, system composition and the workflow were addressed in an ecosystem perspective. Applying the approach in a more narrow scope, however, also brings many benefits, for example within a company or research collaboration that develops a whole application from scratch. This can also serve as a starting point to kick-start an internal ecosystem for system composition. The workflow can be applied for this purpose as illustrated in Fig. 4.10. It starts with designing and modeling the architecture based on service definitions to set the functional boundaries of the parts that will form the system. Subsequently, these parts are developed in parallel and combined in the end. The steps of the workflow are separated in time and space, thus supporting development in distributed teams. Since the link between system composition and component development is based on service definitions, it is ensured that at integration time will be available what has been designed previously.

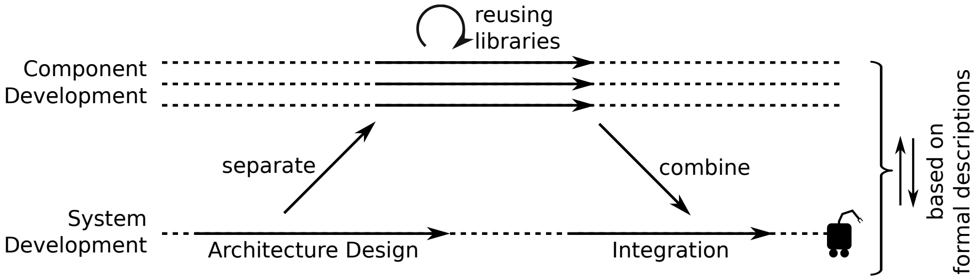


Figure 4.10: The approach applied to a local, non-ecosystem perspective. For example, within a company or collaboration.

Service definitions can be considered an artifact: The service definition model is the *result* of the design activity. In analogy with architecture design, finding and discussing service definitions is an even more important *process* and not only a *result* (see [Sta15; BCK12]). As technical artifacts, service definition models are used throughout the workflow. But the process to come up with them requires interaction and discussion with the involved persons. It requires decisions about structural borders and application-related assumptions to model the service definitions. Service definitions therefore improve the collaboration of participants in a project since they enforce this interaction early. Especially in a very heterogeneous project, discussing service definition models identifies expectations, giving designers a chance to address eventual misconceptions already early in the process before detecting the problem during implementation or even testing. Since services, and thus service definitions, draw the functional boundary between components, writing them down helps to identify white spots: functionality and according qualities that are required but are not (yet) provided by anyone—or the other way around.

Applications are often developed iteratively, adding more functionality step by step. There are situations where changes in services become necessary, for example when additional information is required and the data structure of a service needs extension. Such kind of changes are typically detected at composition-time or later during testing the application and cannot be made during component development or composition since this would break the system structure; it would influence others that rely on that structure. Thus, such changes need to be made by going back from system composition to design of the service definition (Fig. 4.9) with the mutual agreement of everyone who was involved in coming up with that service definition.

4.4 Summary

This chapter has described the way in which this thesis organizes a robotics software business ecosystem in three composition tiers: Tier 1 is defined by ecosystem drivers that define the overall structure, and which thus enable collaboration and composition. The chapter introduced the core composition structure of Tier 1 and motivated its architecture. The meta-structure for composition (the "composition structure") on Tier 1 is the main contribution of this thesis. It enables the definition of domain structures on Tier 2 and the collaboration of ecosystem participants on Tier 3. Tier 2 is shaped by domain experts who define structures that are specific to individual robotics domains (e.g. object recognition, manipulation). Tier 3 uses domain structures to provide actual content for use by the ecosystem participants. This includes software components that are provided by component suppliers and systems that are composed from these components by system builders.

The next chapter provides details of the composition structure at Tier 1 and provides the concrete underlying meta-models.

5
A Meta-Structure for System Composition

This chapter contributes the detailed meta-models for the composition structure at composition Tier 1 (Fig. 5.1). These are used to model the domain structures on Tier 2 which then build the essential frame to model software components and use them to compose systems on Tier 3.

The chapter first provides an overview on the meta-models for reference purposes (section 5.1); it then presents the Tier 1 meta-models in detail and elaborates on why these structures are reasonable structures for addressing system composition.

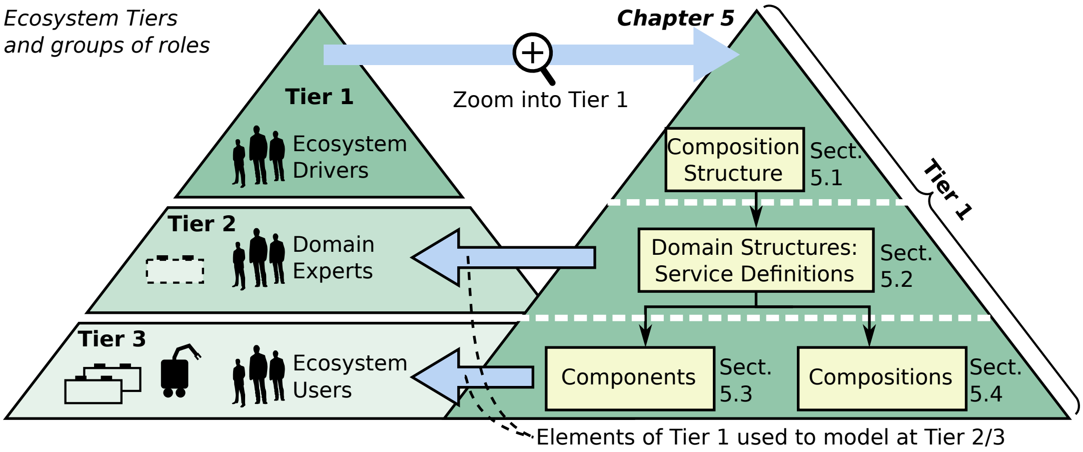


Figure 5.1: The structure of this chapter: It describes the meta-models of Tier 1 (the "composition structure") that are used to model on Tier 2 (domain structures) and Tier 3 (building blocks/components and robot applications/compositions).

5.1 Meta-Model Overview: The Composition Structure

The meta-models that are presented in this chapter are visualized using the graphical Ecore notation from EcoreTools [Ecle]. Ecore [Ste+08] is in widespread use for meta-modeling. The notation is a subset of the Unified Modeling Language (UML) class diagram. The notation can thus be understood without specific Ecore knowledge even though it shapes its visual appearance. The presented meta-models are generic and not tied to Ecore: Any model-driven approach can be used to implement them.

Users of the approach create and use models through graphical and textual DSLs implemented in the SmartMDSD Toolchain. The design and syntax/notation is important for the usability towards the user. From a methodical point of view, the important foundation is the set of underlying meta-models. The look and feel of the DSLs is thus only briefly illustrated by examples in chapter 6.

A high-level view on the meta-models of composition Tier 1 is given in Fig. 5.2. It is provided here as reference and includes all links between sub-models. The remaining sections of this chapter describe each part in detail. The main element under description through each of the following sections is highlighted in green.

Consequences to the composition structure with a focus on modeling have been presented in section 3.3.3. Most decisions in modeling are motivated by the separation of roles (meta-model for each role), by the composition workflow (direct mapping of steps to models), and by the concerns or views that require dedicated DSLs. Section 4.2 motivated the high-level architecture of the approach and the motivation and reasons behind the separation of the main meta-models. The following sections of this chapter underpin the there presented meta-model separation with concrete meta-models.

The core of the approach's meta-models are *ServiceDefinition*, *Component*, *ServiceWishlistModel*, *SysconfModel*, and *DeploymentModel* (Fig. 5.2, highlighted in green). These meta-models address the overall considerations for system composition in an ecosystem as presented in section 3.4. They directly realize the structure presented in section 4.2.1 and Fig. 4.6. In addition to them, the other meta-models that are shown in Fig. 5.2 either provide the infrastructure to realize the links between the core meta-models or and are necessary to realize the overall composition workflow.

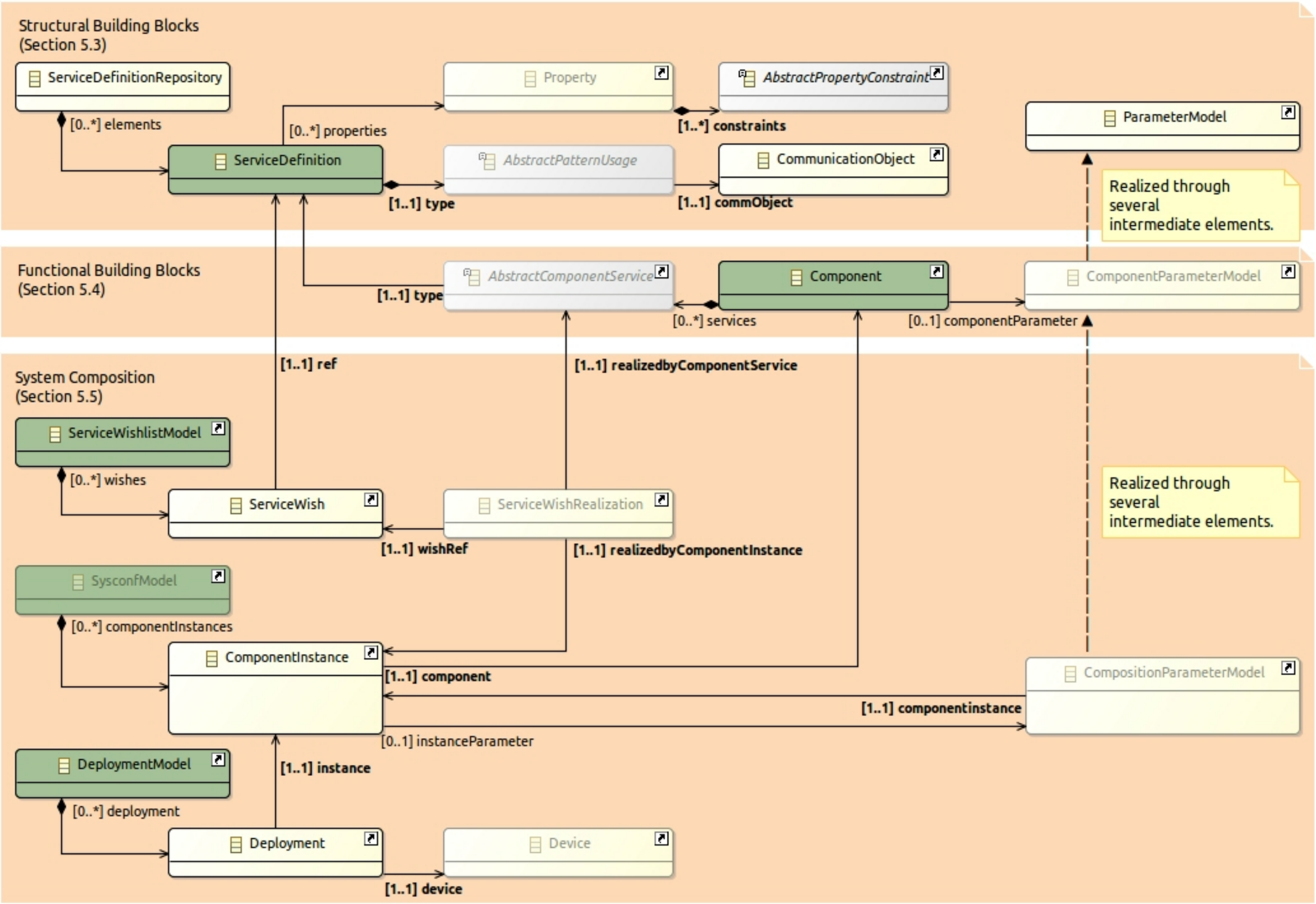


Figure 5.2: An overview on the meta-models of the composition structure at Tier 1 using the Ecore notation from EcoreTools. This figure is a concrete realization of Fig. 4.6. A complete overview is provided here as reference.

5.2 Structural Building Blocks: Domain Structures

Structural building blocks are located at composition Tier 1 and provide the meta-models for modeling at composition Tier 2 (Fig. 5.1). The models on composition Tier 2 capture robotics domain structures. This section presents the according meta-models. The section first presents the service definition meta-model in general and then all its sub-models and other meta-models to model Tier 2 elements.

5.2.1 Service Definitions

A service definition (see section 4.2.2) is the common structure for a class of services in a reusable and formal description that ensures that components offering or using such a service can be used together; the "type" of a service (see [SW04]).

The service definition model is described using a Domain-Specific Language (DSL) that we call a Service Definition Language (SDL) (Interface Definition Language (IDL) for services). The service definition meta-model is illustrated in Fig. 5.3. It includes a meaningful name to identify the service towards component developer and system compositor. This name is an abstract description for the roles to describe in one word what the service is about, using terms from the according domain. A description is used for human-readable documentation in full-text that gives further explanation and is used for component selection during system composition.

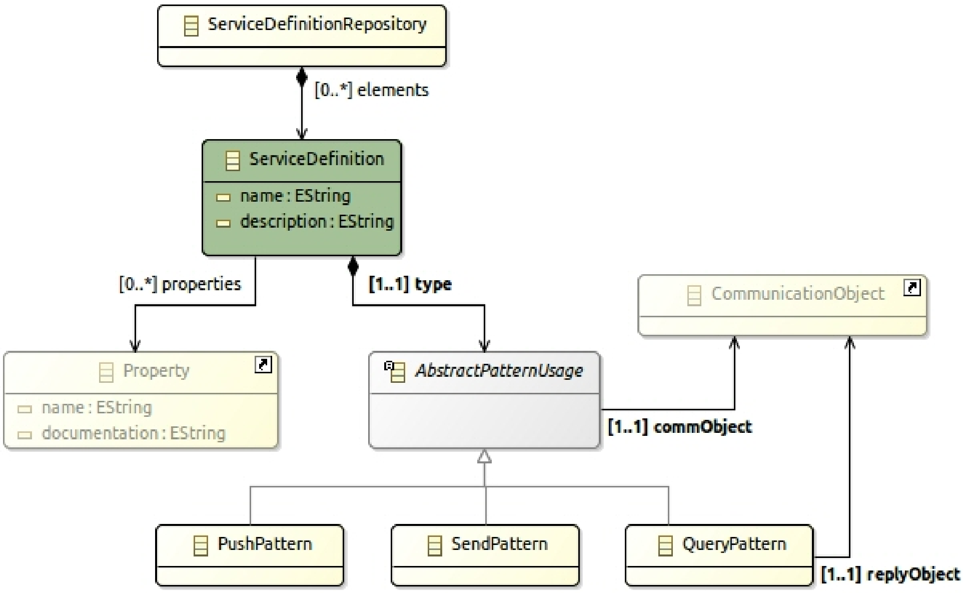


Figure 5.3: The service definition meta-model to model services.

Each service definition model refers to one communication semantics (*type*), refers to service properties (*properties*), and refers to one communication data structure (*commObject*).

Depending on the communication semantics, another communication object (*replyObject*) is needed. Both, the communication object and the service property are defined in an external model and referenced from the service definition. From a composition perspective, both are independent entities since they shall be reusable by multiple-service definitions or might be supplied by different stakeholders.

The *type* of a service defines the communication semantics, here: SmartSoft communication patterns [Sch04a]. The communication data structure (*CommunicationObject*) is attached to the pattern (*commObject* relation and *replyObject* in the derived *QueryPattern*) to simplify modeling and enhance consistency, since some communication patterns require more than one communication data structure (e.g. request/response).

The elements of the service definition (communication objects, communication pattern and properties) are not defined within the service definition model. They are defined separately in external models based on their concerns and are referenced from the service definition. This separation of concerns enhances the separation of roles: separation of what is being communicated (data structure) from how it is being communicated (communication semantics). Having the data structure separated from the service definition enables to use it as "building block" for different service definitions.

Having all necessary elements (communication objects, communication patterns and properties) available in a dedicated service definition model supports composition and creation of alternatives as they define a basis for components to build on: Instead of defining these elements within each component, the components state that they provide or require a service that follows a certain service definition. With respect to separation of roles, this allows to separate stakeholders that define relevant services and the stakeholders that use these services either by supplying components or by using components to build applications.

With respect to the service wish list in the system composition step, it is necessary that a certain service exists (e.g. localization) with certain needs (e.g. accuracy) expressed by service properties. Which component realizes and which component uses that service is not of interest at the time of creating the service definition. With respect to separation of roles and a distributed composition workflow, this allows for defining a "template" of a service in advance. This definition is now separated from refining its details at time of component development and from refining its details at time of composition.

Selecting the communication semantics and the communication object within the service definition narrows the design space for the component supplier to enhance composability (freedom from choice). The component supplier is thus—in a positive way—bound to the communication semantics and data structure as defined in the service definition. If the component supplier were free to decide on the communication data structure and semantics at component implementation time, this would result in not composable components.

From the implementation point of view, a communication pattern (e.g. query) consists of two endpoints in a client/server or publisher/subscriber fashion [Sch04a]. When implementing a component, the component developer must choose an endpoint (e.g. queryServer or queryClient). The same holds true when modeling a component (see section 5.3.1): The component developer must decide which "side" of a service to offer or to require.

A service is at a more abstract level than a service endpoint in a component. This is why a service definition in this thesis refers to the connection between the components. A service definition is modeled without specifying the endpoints: There is no distinction between a client/server or publisher/subscriber within the service definition. This distinction is not necessary at this point in the workflow since the components that use or provide these endpoints are not known yet. However, it is necessary to define the data structure that is being communicated together with how it is being communicated in a service definition model. The concept of service definitions is about structuring domains on composition Tier 2 and about defining a set of service-level interfaces for use at Tier 3 to supply and use building blocks. To define the set of communication data structures and to define a set of communication semantics in isolation is not sufficient for a composition structure. They have strong implications to the component-internals. Being able to combine data structure and communication semantics freely via freedom of choice at component development time limits the composability due to too many possible combinations. Defining the communication data structure alone as a service definition would enable too many implementations of how it can be communicated. Defining the mechanisms of communication alone may lead to too many different data structures for eventually the same information.

If one were to model the endpoints of a service (e.g. localization) with two distinct service definitions (one for "localizationProvider", one for "localizationRequestor"), these two service definitions would require a mapping to each other to express that they belong to each other. The name alone is not sufficient to do so.

During the design of a service definition, it is assumed that the final composition is going to contain a component that provides this service and a component that requires this service. This assumption is valid, because if a service is added to the wish list, but no component is requiring it, it is not needed and can be removed. Vice versa, if a service is required by any component but there is no component providing it, such a component must be found.

We will come back to service endpoints in the component model, where the endpoint is created.

5.2.2 Communication Data Structures

The concept of communication objects is adopted from Schlegel [Sch04a]. The communication data structures translate to communication objects via code-generators. Communication objects and communication data structures are used as synonyms here. Communication objects define the data structure for communication via services. As building blocks for services, they are defined in a stand-alone model for reuse within different service definitions. This contributes to separation of roles such that different participants can contribute communication objects. The meta-model for communication objects is illustrated in Fig. 5.4.

Communication objects are collected within repositories to give them a namespace. Each communication object is a group of named elements with a type. They can be used to define attributes using primitive data types (e.g. integer, boolean, double). Communication objects can be nested and can include other communication objects or simple structs *CommObjRef*. Code-generators later generate according getter- and setter-methods. The generated code-infrastructure also includes mappings to the middleware, so that the defined meta-model for data structures is independent of the chosen middleware.

The provided means to model data structures are simple, but many years of experience in practical application of them showed that they are suitably rich in expressiveness to build complex applications. They provide a benefit in applying them in system composition as demonstrated in chapter 7. Extensions such as adding Object Constraint Language (OCL) constraints for consistency (e.g. range of values or relations between values), adding support for modeling, and managing physical units (as e.g. supported by mbeddr [Voe+13]) are possible within the proposed structure. These enhancements will improve consistency, usability, and performance on top of the composition structures. Adding these extensions requires to extend the meta-model of the communication data structures accordingly, but will not require to alter the overall composition structure.

5.2.3 Communication Semantics

The communication semantics is an important aspect of the agreement between two components that already must be defined for the service definition since it serves as the type of the component's service endpoints. The communication semantics is defined in the meta-model of the service definition and is used in the model by selecting one of the available choices (*type* of a service definition, Fig. 5.3). The available communication semantics are adopted from the SmartSoft framework that introduces "communication patterns" as a set of predefined communication mechanisms to ensure compatibility [Sch04a; Sch06]. Since this thesis focuses on modeling service definitions for the skill layer (see section 3.2.3), the used communication patterns are the relevant ones that focus on communication at the skill layer (see section 3.2.3).

As argued in section 5.2.1, there is no further distinction between the endpoints such as client/server or publisher/subscriber, since this is not necessary at this time. This information is relevant for the service endpoints, which are not known at this stage. Deciding on the component endpoint, and thus deciding on the "side" of a service (client/server or publisher/subscriber), is the decision of the component developer.

5.2.4 Service Properties

System composition requires to explicate the semantics of a service on the application-level to improve composability while supporting separation of roles. If not expressed, the semantics remains hidden within the implementation or documentation and mismatches will be detected too late, possibly not before run-time. The proposed composition structure addresses this by providing means to create and use "service properties" as a part of service definitions. They enable component suppliers to express the offers of their component via properties. They enable system builders to express the needs of their application via constraints on these properties.

The main challenge with properties for system composition is to come up with adequate separation of roles and to apply freedom from choice to do so. For this purpose, it is necessary to establish means on Tier 1 that enable domain experts to establish domain-specific properties on composition Tier 2. The properties then can be used by component suppliers and system builders independently on Tier 3 while both roles still adhere to the same structures on Tier 2.

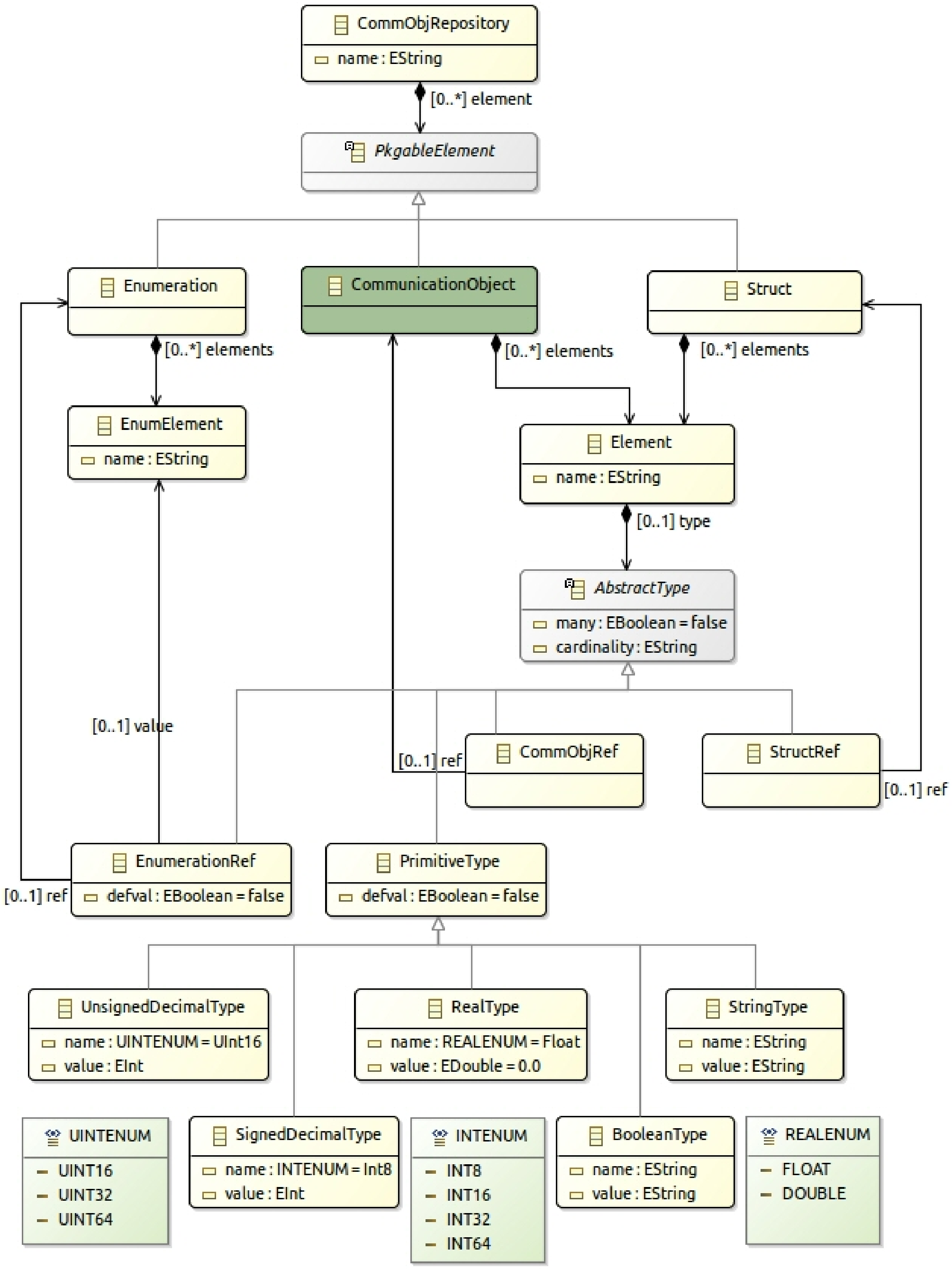


Figure 5.4: The communication object meta-model to model data structures of service definitions.

This thesis introduces the concept of service properties on Tier 1 and uses typed name–value pairs for this purpose. They are simple, yet demonstrate the benefit for system composition. Since these properties are part of the composition structure and overall workflow, they establish a continuous workflow of managing properties. Managing properties already provides a benefit for system composition. The property meta-model provides possible points of extension for future work: More advanced ways of expressing properties, such as hierarchical properties or complex data types, applying OCL or managing physical units (e.g. as in mbeddr [Voe+13]), for example, will enhance modeling, expressiveness, and usability; they can be applied to refine property management but do not require to alter the proposed composition structure and workflow in any way. All changes will remain within the properties-part of the composition structure. The same holds true for constraints on the properties to express needs. These constraints are rather simple in expressiveness but demonstrate that using service properties increases composability and enables the selection of composable components.

Service Properties in the Composition Workflow

Service properties explicate the semantics of a service on the application-level that is otherwise hidden within implementation or documentation (see section 4.2.2). The concept of service properties is defined at Tier 1 (Fig. 5.1). Concrete service properties are defined at composition Tier 2 by domain experts; they thus contribute to building domain-specific vocabularies and structures. Service properties are refined with values on Tier 3 by component suppliers and system builders.

Constraints on property values on Tier 3 are used to determine the compatibility of services (see Fig. 4.6). They are used in components to express needs to other services and in the wish list to express needs to components for component selection and system verification. The use of constraints enhances the composability since the constraints allow to check compatibility on a syntactic level (service signatures match) and semantic level (constraints on properties match). This is applied both within the vertical axis between component and application. It is also applied within the horizontal axis between components (cf. Fig. 3.9). Service properties also enhance component selection (section 5.4.2) since properties allow to distinguish between component alternatives (section 3.3.2).

A service property is used in three steps as illustrated in Fig. 5.5: definition of a property and definition of a constraint, instantiation of a property, and the evaluation of the constraint.

Definition of a Property and Definition of a Constraint Evaluation. Service properties are modeled as typed name–value pairs. They are referenced from the service definition model to express that a certain service property is part of the service definition. Since service properties define a vocabulary that might be of relevance in other service definitions as well, they are modeled as reusable elements and are not defined within the service definition.

To support separation of roles and freedom from choice, expressions for use in constraints are also defined along with the properties in the same step. Only these expressions are available for use with constraints, thus limiting the available options to the necessary ones (applies freedom from choice). As the evaluation uses meaningful names, they can be

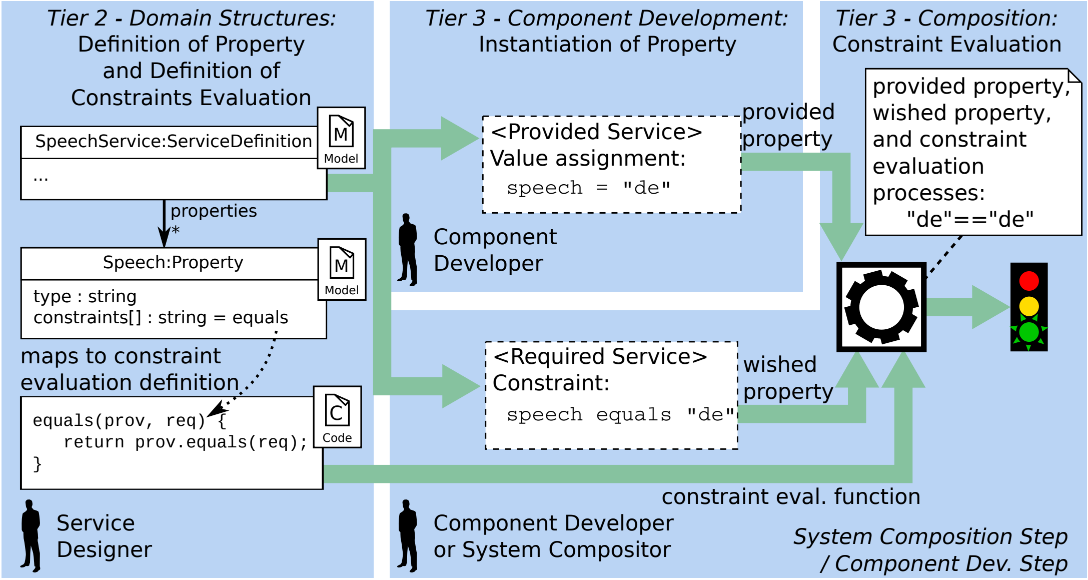


Figure 5.5: Service properties along the composition workflow.

defined using the vocabulary of the domain language which it is intended for (applies separation of roles). Expressing the needs is thus closer to the user and is easier to read and understand.

Property Instantiation. Service properties are refined when service definitions are instantiated, e.g. during component modeling. A value is assigned to the property in case of a provided service to express what the service offers. Its counterpart, a required service, expresses its needs by expressing a constraint on the property: It consists of a value and a predefined evaluation function that expresses a constraint. When modeling a wish list, a service property is modeled as a “required” property.

Constraint Evaluation. Service properties are evaluated for component selection by constraints based filtering of available components. They are also used for verifying the composed application by evaluating the properties of connected components. The selected service property constraint is evaluated with the values from provided and required services.

Service Properties Meta-Model

The service property meta-model is illustrated in Fig. 5.6. Service properties are defined as name–value pairs with a name for a constraint evaluation. A property can be of simple data types such as a string, an integer, and a boolean. The name of the property provides a human-readable identifier of the property and it should be chosen using the vocabulary of the domain in which this property is used. A detailed prose text description can be attached for documentation purposes.

Each property can define one or several constraints for selection when the property is instantiated. There are standard expressions such as =, >= and <= for use as very basic constraints. The

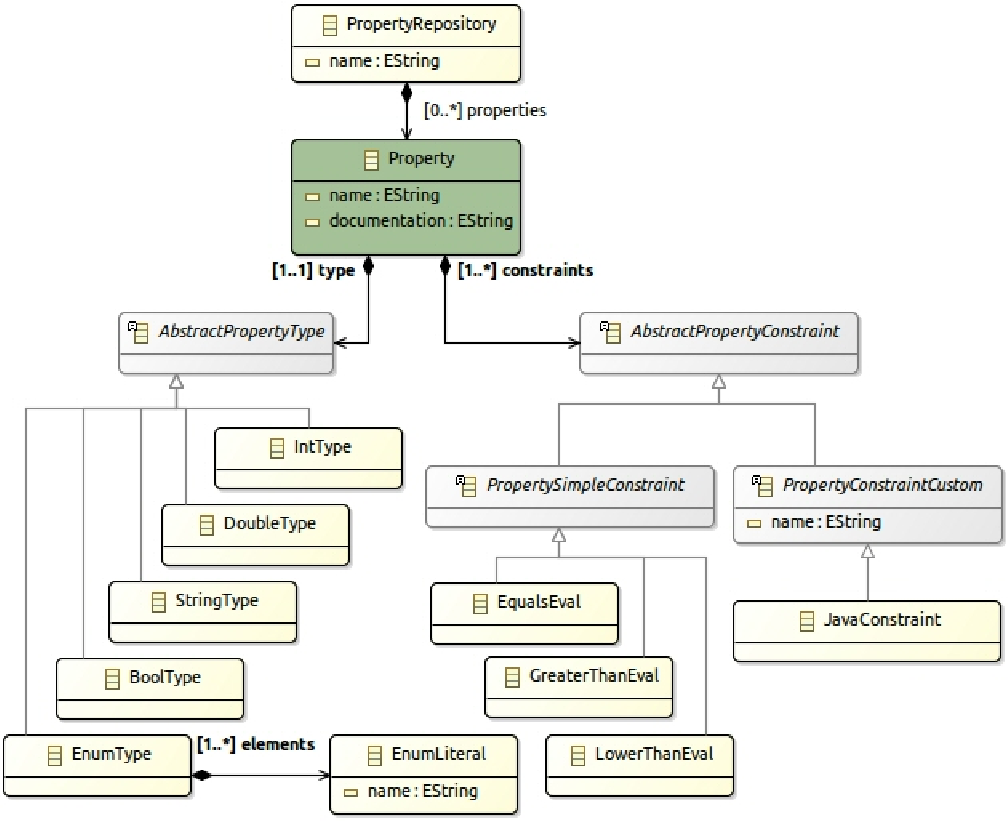


Figure 5.6: The meta-model for service properties.

most common use-case, however, is to define custom constraints. When modeling a new constraint, only the name is of relevance. The constraints will always be referred to by name as the meaning of a constraint heavily depends on the vocabulary that is used in the domain. When using the property, referring to the constraint by name immediately provides a meaning and understanding to the user instead of the need to interpret and understand a cryptic expression. Currently, the only available class for custom constraints is *JavaConstraint* that maps to a Java method implementing the constraint (see also: Fig. 5.7). It would also be possible to add further constraint classes by extending the meta-model and deriving from *PropertyConstraintCustom*.

The definition of properties does not come with assigned values. Values to the individual properties are assigned later during using and instantiating the properties by the component developer or system compositor.

Modeling Considerations: Fixed vs. Domain-Specific Properties Concrete service properties are not part of the meta-model, i.e. concrete properties are modeled on composition Tier 2 and not on Tier 1. The reason is that service properties are domain-specific. The properties used in one robotics domain might differ from the ones used in another robotics domain. For example, the domain of computer vision uses different properties than the mobile manipulation domain. From the meta-model perspective, this is an open set and cannot be defined in advance within a meta-model at Tier 1. Therefore, there is a need to provide means within the meta-model to model and use such properties by domain experts on Tier 2 such that ecosystem users on Tier 3 can use them. In other words, it is about the consistent extension of the Tier 1 composition structures by domain-specific structures on one hand. On the other hand, it is about not giving full freedom to use them by the ecosystem users but to still apply freedom from choice via domain-specific structures on Tier 2.

While the above paragraph described examples of structures that are specific to particular domains in robotics, some properties exist that may be included in Tier 1. Communication characteristics, like frequency or latency of communication, are not bound to a particular domain in robotics (e.g. manipulation, computer vision). These properties are of use in any domain and therefore can be included in Tier 1 as part of the component model. Including such general properties on Tier 1 should be addressed in future work. Another example of concepts that is general to any robotics domain are the communication patterns, which are for this reason on Tier 1.

Modeling Considerations: Service Properties vs. Communication Objects Service properties and attributes of communication objects are closely related, sometimes even identical. A localization service, for example, holds the accuracy both in the communication object and the service property. Instead of using one model for service properties and communication object, they are still separated into two models. This makes sense from a composition point of view to allow reuse of individual properties, but also practical considerations lead to this separation since the attributes are of different concerns and might be different in representation. The following two sections elaborate this in detail.

Different Concerns. Communication objects and service properties are different concerns. The communication object is about describing the data structure of communication while the service property describes the semantics of the service on an application-level to someone that uses it.

The availability of values of service definitions and attributes of the communication object also differs in time. Concrete values of service properties are available and relevant at design-time—values of communication objects are available at run-time, depend on the state of the robot and are thus not accessible at design-time. Design-time service properties could be used at run-time, for example to monitor the required localization quality, but this is part of future work.

Further, the separation of roles requires different models for communication object and service properties since both might be provided independently in time and may be contributed by different ecosystem participants.

Different Representations. Communication objects and service properties might use different representations. Since they are intended for different roles and concerns, they might vary in abstraction or structure. For example, the accuracy of localization might be relevant for other components and thus be part of the communication object. It might be represented using a 9x9 covariance matrix with double values to cover the current uncertainty of location and heading of the robot. The service property at design time, however, shall only consist of an abstract representation in a single value. The value shall represent the maximum allowed variance as this information is sufficient at that time.

The absence of a property also might require different representations. There are cases in which values are only needed in the communication object or where they are only needed as a service property. Communication objects, for example, might contain an identifier to identify recognized objects at run-time. This identifier is not of interest as a service property at design time.

Evaluation of Custom Constraints

Constraints are defined within the service property and consist of a single name that identifies the evaluation function on a model-level (Fig. 5.6). The name contributes to the concept of properties supporting the domain-vocabulary for domain structures at Tier 2. The constraint evaluation is expressed by implementing an evaluation method to which the constraint name maps to (Fig. 5.7). The method is executed each time the according service property is evaluated, like a comparison operator known from programming languages. The evaluation method is implemented in a general-purpose programming language: in the implementation of this thesis, this is Java.

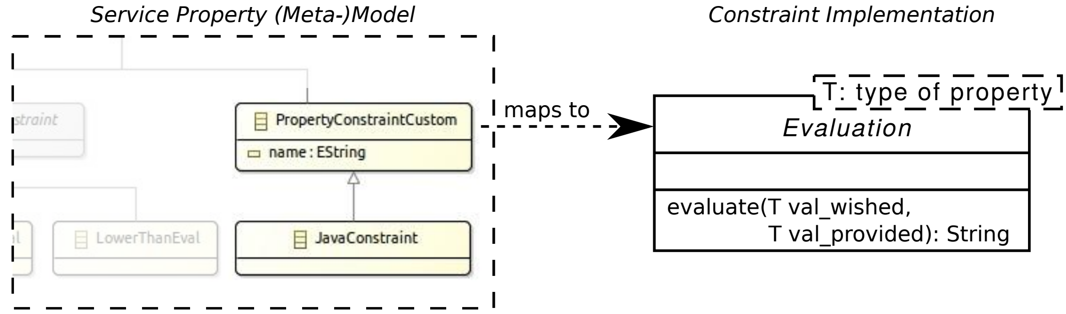


Figure 5.7: A named constraint of a service property maps to an evaluation method (illustrated by a UML class template) that implements the constraint.

The return value is expected to be an empty string if the constraint holds true, that is, if the value of the provided property *val_provided* matches the needs expressed in *val_wished*. Otherwise, the method shall return a message providing more details to the user. The input arguments are taken from instances of service definitions as illustrated in Fig. 5.5.

The evaluation method is called when properties are compared. At this point in time, it is not further defined what happens if the evaluation fails. Its effect to the overall composition cannot yet be decided. In other words, a negative evaluation of a property does not yet mean that the composition is invalid. Only when expressing needs of a required service, it is known how a mismatch effects the composition and if it is still valid. Section 5.3.3 will discuss the effect of a failed evaluation to the composition in more detail. It introduces the annotation of a "severity" when service properties are instantiated.

Using a general-purpose language for expressing the constraint provides flexibility for its implementation. It allows using tools of the language to build complex expressions, such as loops, string operations, mapping tables, and other operations. However, this is freedom of choice instead of applying the freedom from choice philosophy, which is argued as means for separation of roles. Applying freedom of choice in this case does not break the separation of roles. Separation of roles is about managing the handover and interfaces between different roles. The interface between the two involved role is given by the model: The evaluation function is modeled and implemented by the same role, i.e. freedom of choice only applies to the same role that models the constraint on Tier 2 as part of the domain structures. Freedom from choice is still applied to the users of the constraint: Component developers and system builders on Tier 3 can only use the constraints modeled on Tier 2. The model comes with the constraint implementation, but the implementation cannot be changed by the users.

Separating the definition of the constraint by a name from its implementation supports separation of concerns as it separates its name from the execution semantics. Introducing a name for the constraint in the model also supports separation of roles since it allows for building a domain vocabulary: The user of the constraint can just use the (meaningful) name instead of the cryptic constraint expression where the meaning of, for example, "equals" or "greater than" might depend on naming and semantics the property. From a composition point of view, the constraint can be reused by its name instead of repeating its implementation each time it is used.

An alternative approach is to express the constraint at the model-level. This would require updating the meta-model such that it can express the constraint directly. It then could be directly evaluated from the model via tools given from the modeling-world. For example, Xbase [Ecld] is considered a good choice since it integrates with Xtext [Ecli] (Xtext was used to implement the approach in the SmartMDSD Toolchain). Further, the constraints expressed in the model could be transformed to an expression engine or to an interpreted expression language such as Duktape [Vaa+]. Duktape is a java script engine that allows for evaluating expressions in a run-time interpreter.

5.2.5 Reusable Variation Points: Parameter Sets

Parameter sets define possible variation points in a reusable model during system design. The component model later refers to a parameter set to provide these parameters for initial component parameterization. At composition-time, the parameter set is then accessible from outside the component to parameterize the component.

A "parameter set" (*ParamSet*, Fig. 5.8) consists of individual parameters (*Param*). Each parameter is modeled as a typed name–value pair (*AttributeDefinition*, Fig. 5.8; see Fig. 5.9). No values can be assigned when using the meta-model, since only the structure / definition of the domain-vocabulary is of relevance at this stage (composition Tier 2). A value for this attribute will be assigned at a later step when this parameter is used at component modeling time (setting the default parameter) or composition-time (configuration of a component to the robot application).

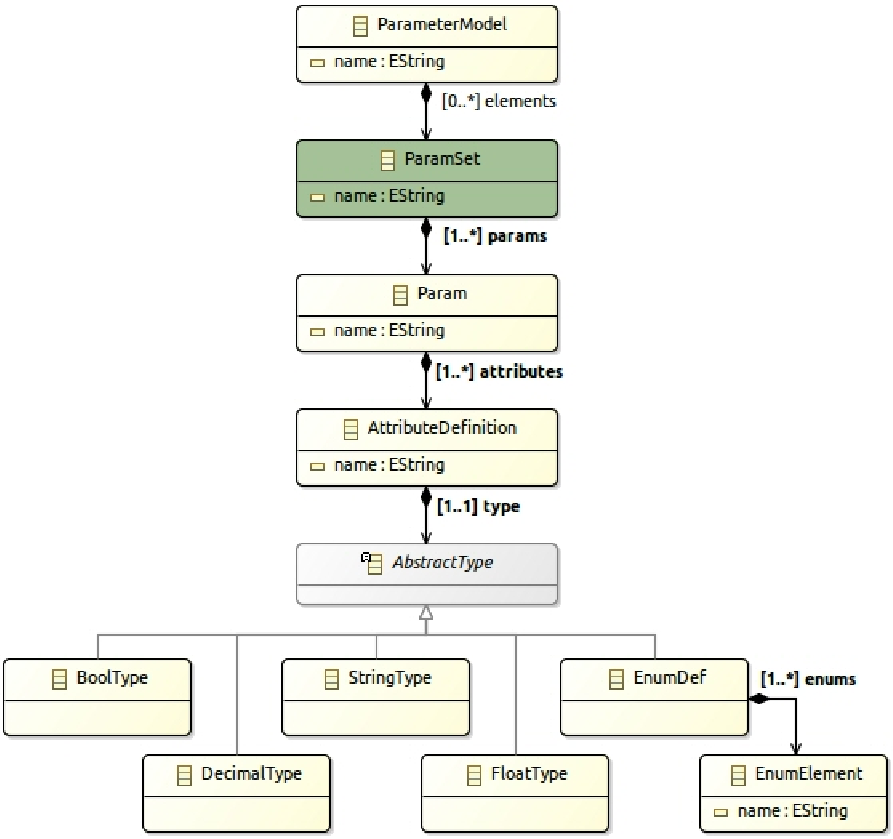


Figure 5.8: The parameter set meta-model defines a reusable collection of parameters for variation point modeling. Components use them to express how they can be configured at composition-time.

Grouping of attributes to parameters (Fig. 5.9) is convenient to allow for complex parameters, for example to group attributes for the maximum forward, backward and rotational speed into a single parameter "speed". Component suppliers (re)use parameter sets to compose the parameterization of their component. Since several components use the same parameter set, the parameterization of these components is standardized and allows for a unified parameterization of component alternatives.

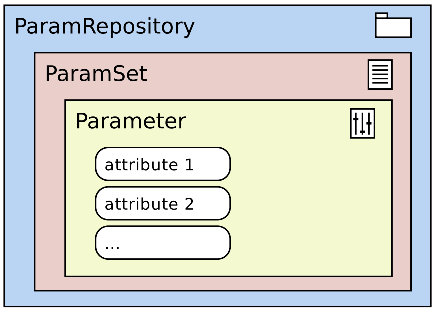


Figure 5.9: Illustration of parameter sets: Parameters consist of one or more typed name–value pairs and are grouped into sets of parameters. These "sets" collect commonly used parameters.

A parameter set does not introduce a hierarchy of parameters. Parameter sets group together the parameters that are typically used together. This group is then consistent. It is a reasonable unit for composition and for convenient reuse of the group instead of picking each single parameter at composition-time over and over again. This would be freedom of choice and would risk inconsistency. The parameter set is not visible at composition-time, but the individual parameters are (see section 5.4.3).

Leaving multiple parameters in a group to assign values raises potential for conflicts in a single value itself. Single values might not be supported or there might be parameters in conflict with another. For example, a parameter for a mapping algorithm that only supports quadratic grid cells. It would require the map width and height to be the same. OCL can be used for this purpose to add local constraints to the parameterization of components, to name one potential extension.

5.2.6 Documentation

Documentation is a cross-cutting concern in system development and is of particular importance for separation of roles. Each element in the presented meta-models of structural building blocks includes a *Documentation* element to annotate prose text documentation. An example is illustrated in Fig. 5.10 for the communication object meta-model. This relation is not included in the presented meta-model diagrams to keep them clean and not to clutter their representation.

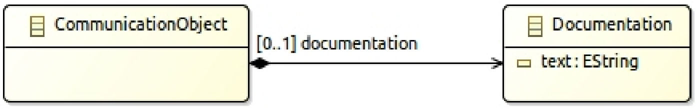


Figure 5.10: The documentation of communication objects as an example of annotating documentation to meta-model elements.

5.3 Functional Building Blocks: Software Components

This chapter describes the foundations for supplying software components to provide "functional content" that can be used by system builders. The concepts described are defined at composition Tier 1 (Fig. 5.1) for use for modeling by ecosystem users (component suppliers) at composition Tier 3.

"Functionality is the ability of the system to do the work for which it was intended" [BCK12, p. 65]. A functional building block is a piece for composition that provides functionality as part of the system, for example recognizing objects as part of a robot butler. A software component is the unit of composition and the unit of exchange in the ecosystem. It provides functionality to the system through formally defined services at a certain level of abstraction (see Szyperski [Szy02]). It provides a platform for the implementation of algorithms or integration of functions that can interface with other components through formally defined services only.

5.3.1 Component Meta-Model

The component is described through a component model that conforms to a component meta-model. The component model describes the component hull to the system compositor. The component model is composed from structural building blocks (section 5.2). The component model is used to provide (generate source code) the infrastructure and hulls for the user to fill with custom algorithms. The component includes variation points that can be used to configure it to the robotics application at composition-time (section 5.4).

Component modeling covers the component hull and focuses on services as most important elements for composition. System composition at large also requires considering the elements of the infrastructure beneath the component hull since they influence the system or are influenced by the system in general. For example, tasks running the implementations influence scheduling and push-rates of services. These details are, for example, discussed in the context of SmartSoft by [Lot+16; Lot+15] and are in line with the approach and component meta-model proposed in this thesis. Thus, component-internals are not further considered in this thesis as its focus is on organizing services for composition.

The presented component meta-model (Fig. 5.11) is in line with the SmartMARS meta-model [SSS09a; Ser] and extends it for use with service definitions. A component consists of one or several service endpoints, called a "component service" (*AbstractComponentService*, Fig. 5.11), that refers to a service definition to denote the "type" of the service they realize. One or more service endpoints are created as part of the component model and refer to a service definition by its name. The name of the service endpoint is used towards the component developer and system compositor to work with the concrete service endpoint of the particular component. Service endpoints are used from within the user-implementation of the component to communicate with other components.

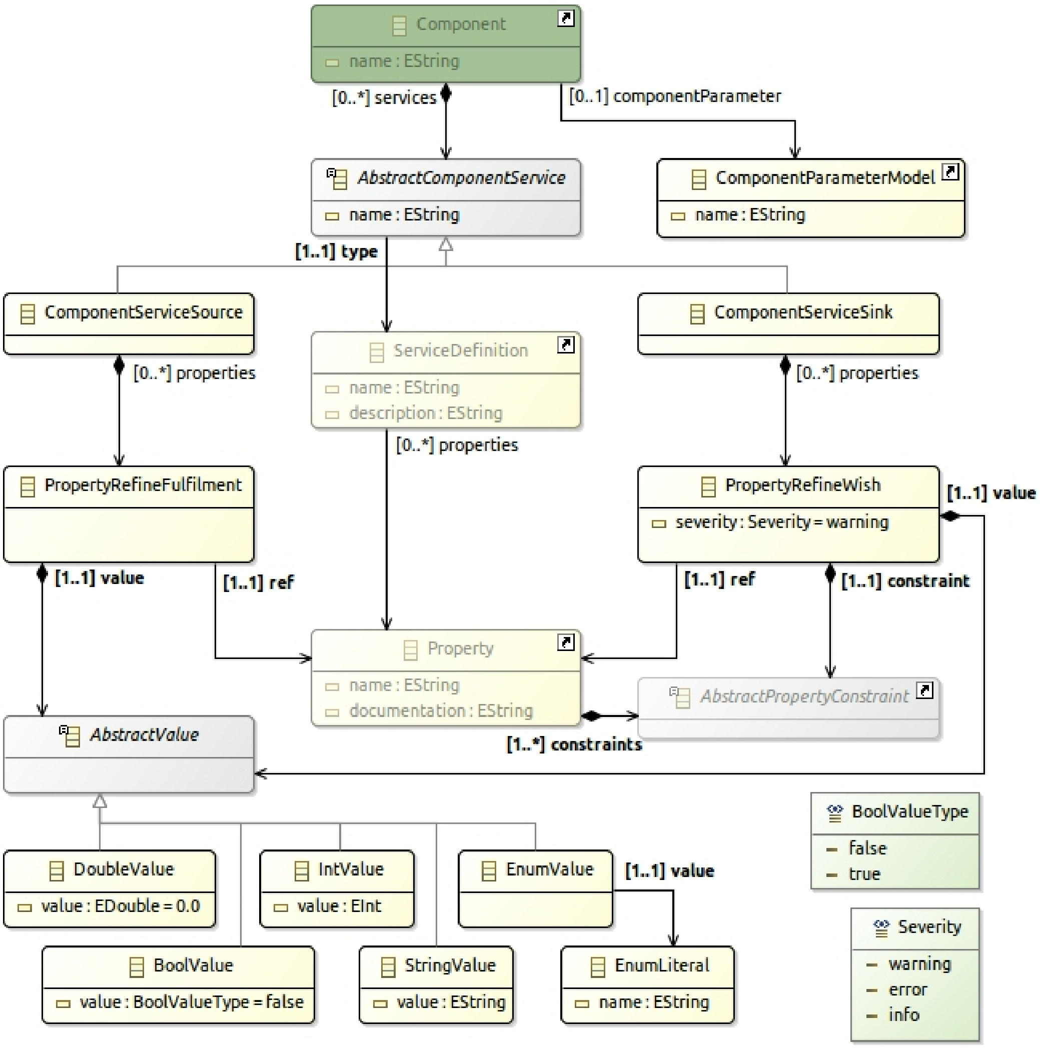


Figure 5.11: The component meta-model to model the hull of a component with respect to services and service definitions.

A service definition describes the service as a whole and does not distinguish between one end offering a service and another end consuming a service (section 5.2.1). The component model is the model where the developer “selects sides” of an endpoint for a component service. Selecting the offering or consuming part of a service is necessary at this step in the workflow, since this information is technically required to generate the source code for the component hull. Further, the needs that are expressed in service properties within service definitions require a directed relation for comparison. It is about one partner providing an offer and another partner expressing a need: The need must be compared with the offer and not vice versa. This directed relation is provided by taking the view of sources and sinks.

A service endpoint instantiates a service definition either as a service “source” or service “sink”. A service source is the endpoint from which information is communicated to the service sink. Service sinks specify a wish towards a service source which annotates what it provides by instantiating properties.

Meta-model details on service endpoints will be provided in the remainder of this section.

5.3.2 Service Endpoints

We call service endpoints “source” and “sink” with respect to the flow of information between elements: messages are sent from source to sink. The chosen wording puts the focus on the flow of information within the system. It is reasonable to understand for system composition since it shifts the interaction between components from a technical perspective to an application perspective. It also applies separation of roles: early definition of the service in general (composition Tier 2, Fig. 5.1) while choosing the “side” during later refinement (composition Tier 3, component supplier).

The commonly used classic terms “provider”, “requestor”, “consumer”, or “user” are considered less suitable terms for defining the service endpoints since the technical wording is on the wrong level with respect to the current step in the workflow which is application-driven. Further, the wording is ambiguous. Using the classic terms can lead to misunderstandings in the process of defining services to draw functional boundaries between components because the flow of information and the direction is relevant for the architecture but is not necessarily covered by these terms.

When using the classic terms, it is not always clear from the wording who is the service provider and who is the service user—this depends on the description of the service. Figure 5.12 provides an example. A collision avoidance component instructs a robot base platform how to move, e.g. based on inputs of a laser scan. In both illustrated cases, the information flow (navigation instructions: speed and steering) is from left to right. However, the expected provider/requestor relation changes depending on the wording of the service: whether the service is described as “motion execution” or “navigation instructions”. Which end the requestor is and which end the provider is depends on the description of the service and its naming. One can describe a “service” from both the provider and requestor perspective.

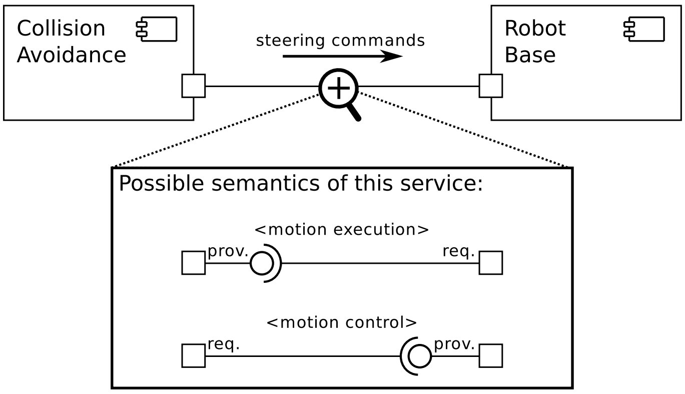


Figure 5.12: Service endpoints on the service level: Who is provider or requestor depends on how the service is described.

The classic terms are most often used on a technical level. For example, the consumer/requestor part is typically the caller while the providing part is the callee in SOA and in UML (see [Jos09]). With respect to SmartSoft, the technical foundation in this thesis, the service provider is the server-part of a communication pattern [Sch04a]. Figure 5.13 shows how the scenario of collision avoidance and robot base can be realized using SmartSoft communication patterns (we neglect the fact that not every pattern is reasonable for every use-case, but the illustrative example can also be applied to others). When choosing the send or query pattern versus the push pattern, the provided/required relation changes. Who is who depends on the technical realization: the chosen pattern.

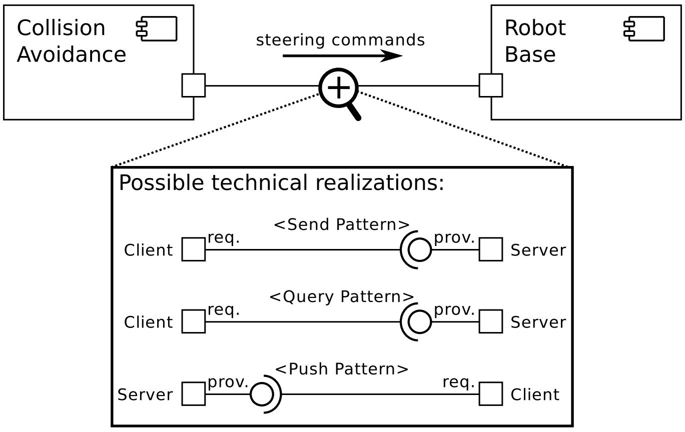


Figure 5.13: Service endpoints on the technical level: With respect to SmartSoft, who is the provider or requestor depends on the chosen communication pattern.

The best-practice for naming and describing the service definition is to take the perspective of the information flow. Names of service endpoints in components should be chosen in a meaningful or "speaking" manner. Names should be of use and "speak" about its purpose to the component developer or system compositor. This supports the roles in selecting the right service definition or component.

5.3.3 Instantiation of Service Properties

This section describes the meta-model for instantiating service properties in the composition workflow (Fig. 5.5). A service definition is instantiated in the component model via a component service. What is left is the instantiation of service properties to express the "needs" that a service sink has towards the "offer" of a service source.

Service Sources

The left-hand side of Fig. 5.11 shows the service source (*ComponentServiceSource*) in the component meta-model. Properties of a service source are refined (*PropertyRefineFulfilment*) by assigning a *value* to a reference of a service *Property* (by its name). A service source can only refine properties that come with the corresponding *ServiceDefinition*. The *value* assigned to a property can be of simple types (double, int, bool, string and enumerations). It must correspond to the types of the referring property (the *AbstractPropertyType* of the corresponding service property that is being instantiated from the service property meta-model).

Service Sinks

The component meta-model (Fig. 5.11) shows the service sink on the right-hand side (*ComponentServiceSink*). Properties of a service sink are refined (*PropertyRefineFulfilment*) by assigning a *value* to a reference of a service *Property*, selecting a constraint (*AbstractPropertyConstraint*), and selecting a *Severity*. Section 5.2.4 describes the service properties in relation to the overall composition workflow.

The constraint on a property is used to evaluate the offer of the service source with the need of the service sink. A service sink can only refine properties that are defined in the corresponding *ServiceDefinition* of that component service. The assigned *value* can be of a simple type such as double, int, bool, string, or enumeration. The type must correspond to the type of the *AbstractPropertyType* as selected in the definition of the service property. The selected constraint must correspond to one of the constraints as defined in the referenced service property. The previous step in the workflow allowed to define one or more constraints in the service property model. The service instantiation in the component model is the right place and step in the workflow to select one constraint.

Severity of Service Properties

The *severity* attribute of a property refinement expresses the effect of a failed constraint evaluation. In other words, it expresses how important it is to meet the expressed need (Table 5.1). The severity is used to interpret the return value of the constraint evaluation method (see section 5.2.4) during matchmaking for component selection and during verification in the system composition step. Severities on service properties are used in the service sink endpoint of a component to describe the needs of the sink. The system compositor also uses them to express the severity of the application's needs in the wish list model.

Severity	Effect of failed constraint evaluation
Info	No effect at all.
Warning	A message will be displayed, the composition is still valid, the component still considered suitable.
Error	A message will be displayed, the composition is not valid, the component is not suitable.

Table 5.1: Severity of service properties: The severity of a property refinement expresses the effect of a failed constraint evaluation.

Using a severity at component development time comes with high responsibility with respect to composability. When considering the system as a whole, the severity in a single component is a local decision that will affect the overall composition. With respect to the workflow, this is a early decision at component development time that will have an effect at composition-time. This is something that can break the composability of components. In consequence, the component developer must only use severities when a failed constraint on a property would render a component unusable. For example, the service sink may use a property to express that a service must be connected or that it only is compatible with a certain data representation that might be encoded in a property. The main use of severities should be in the hand of the system compositor when expressing the application's needs. Only the system compositor has the required overall knowledge of the system being built. Only the system compositor knows about the system's needs. In consequence, only he can decide what to do when these needs are not met.

A possible extension to the proposed severity is to annotate already in Tier 2 which role in Tier-3 may use a severity on a certain property at all. The need for a certain service to be connected may be represented through a distinct attribute in the service sink of the component meta-model.

5.3.4 Service Endpoint Mapping

The SmartMDSD Toolchain generates the source code for the component hull from the component model. At this point in the workflow, all relevant information with respect to services is available: the communication semantics and the data structure comes with the service definition and the service endpoint comes with the component service. Based on this information, a mapping table (Table 5.2) is used as input to generate the source code of component services. In this thesis, this is realized with according code-generation templates for the SmartSoft framework. For example, consider a service definition that is instantiated in the component model as a service source. Using the send pattern and the communication object CommLocation in the service definition would map to the SmartSoft class template *CHS::SendClient<CommLocation>*.

SmartSoft Communication Pattern	Source Endpoint	Sink Endpoint
Send	SendClient	SendServer
Query	QueryServer	QueryClient
Push	PushServer	PushClient

Table 5.2: Mapping of service endpoints to SmartSoft communication patterns.

5.3.5 Parameterization

Parameters in a component model define the variation points that are used to configure a component during composition. The component can access these variation points from within the implementation. The meta-model for component parameters is illustrated in Fig. 5.14.

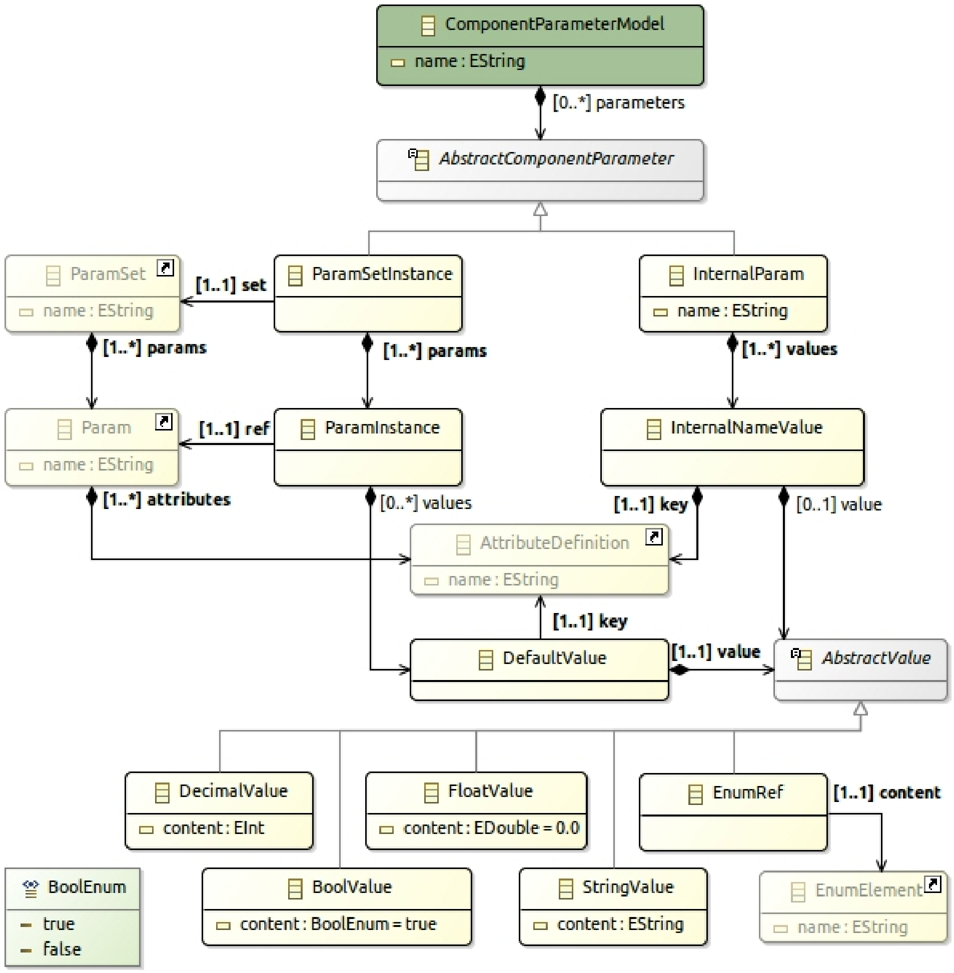


Figure 5.14: The parameter meta-model to describe the variation points of a component.

Parameter Sets and Component-Internal Parameters

A component model can instantiate a parameter set from structural building blocks to express that the component provides these variation points (*ParamSetInstance*). A component can also create a new component-internal parameter (*InternalParam*). In contrast to a parameter set, a component-internal parameter is only available within the component where it was defined; it cannot be reused in other component models. The parameter set unifies the parameterization of several components using the same parameter set. The component-internal parameter is used to describe specific parameters in an individual component only. One might use, for example, a parameter set to configure a class of components for motion execution. One would define a generic parameter set to configure the maximum speed of the robot. The implementation of a particular component might use additional parameters that are specific only for the particular component, such as parameterization of filters in the algorithm. These component-specific parameters are modeled in a component-internal parameter.

Using a Parameter Set

Instances of parameter sets (*ParamSetInstance*, section 5.2.5 left part) refer to a *ParamSet*-model from structural building blocks. It contains several instances of parameters (*ParamInstance*) to which a *DefaultValue* of simple types (integer, bool, string, etc.) is assigned (Fig. 5.15). The default value will be used for the parameter in case the parameter is not set at time of system composition. The types of the default values must correspond to the type of the parameter attribute (*AttributeDefinition::type*, Fig. 5.8) within the referenced parameter set.

The purpose of instantiating a parameter set is twofold. First, the component chooses one or more parameter sets from the available ones to express the variation points of the component. Second, the component developer specifies default values for the component under development. Default parameters are not available when modeling the parameter set since, at that stage in the workflow, the component and thus the necessary default values are not known.

Using Component-Internal Parameters

The meta-model part for the component-internal parameter (section 5.2.5 right part) does not reference to a parameter set. It models the parameters with typed name–value pairs. A component-internal parameter is thus on the same level as parameter in a parameter set (Fig. 5.15).

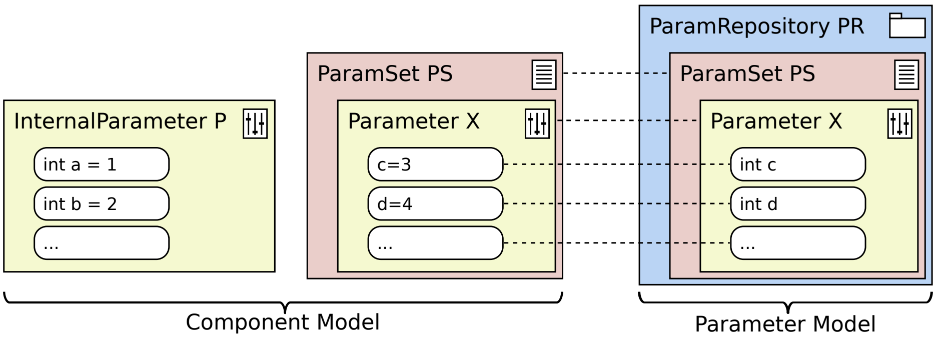


Figure 5.15: The structure and usage of component-internal parameters in comparison to parameter sets.

5.3.6 Component Documentation

The component documentation meta-model is a structure to support the annotation of a service endpoint with a human-readable description. The documentation model is created by the component developer who adds a prose text description to a component service. The code-generator uses the additional information from the component model and from the service definitions to automatically generate a full documentation for the system compositor role. This documentation is written from the outside perspective on the component. It contains information that is to be provided to the system compositor role in the system composition view. The documentation shall describe details such as the used algorithms or methods behind the service endpoint for possible users of the component to provide the insights they need for selecting and using this component.

Each structural building block from the previous chapter is documented directly within the model (section 5.2.6). The documentation of component services, however, is separated from the component model to separate roles and concerns: The technical concern of component modeling is separated from the descriptive concern of writing documentation. The documentation might thus be created or revised by a dedicated role, for example by a technical writer. He can only add documentation to existing elements of the component, but cannot modify the component model (adding or changing services).

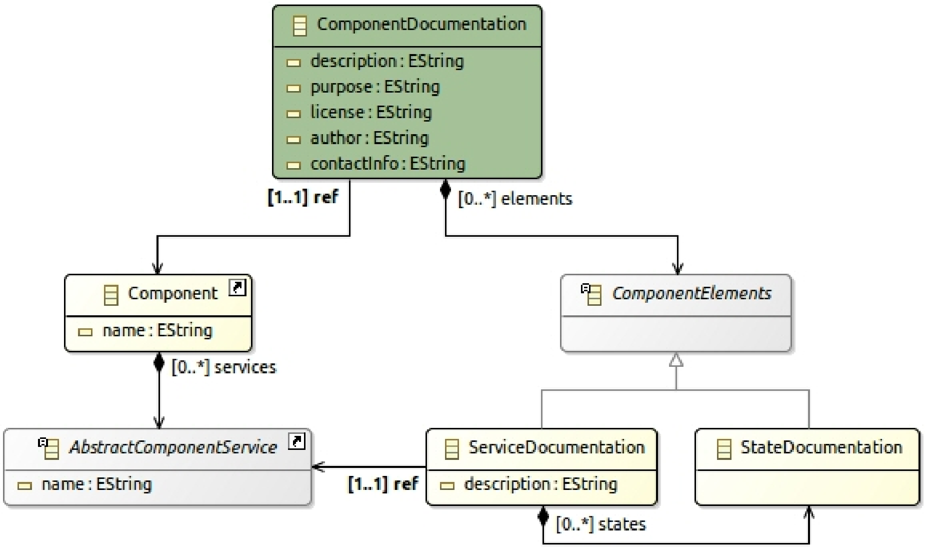


Figure 5.16: The component documentation meta-model to describe component services.

The documentation meta-model is illustrated in Fig. 5.16. The main element *ComponentDocumentation* refers to the component being documented. It links the documentation model with the component model. The documentation contains a set of attributes to describe the component itself, e.g. a description, author, contact information, and licensing information. Each element of the documentation model (*ServiceDocumentation*) refers to a component service endpoint (*AbstractComponentService*) to describe it in prose text (*ServiceDocumentation::description*). The documentation model can only describe service endpoints that are included in the component that it describes. Since there are only references, it is ensured that the component model is only described, but not modified (no change of existing services or adding new ones).

The *StateDocumentation* is used to describe the state-automaton for the component lifecycle [SLS11] in relation to service endpoints. They are relevant in the context of run-time parameterization which is beyond the scope of this thesis.

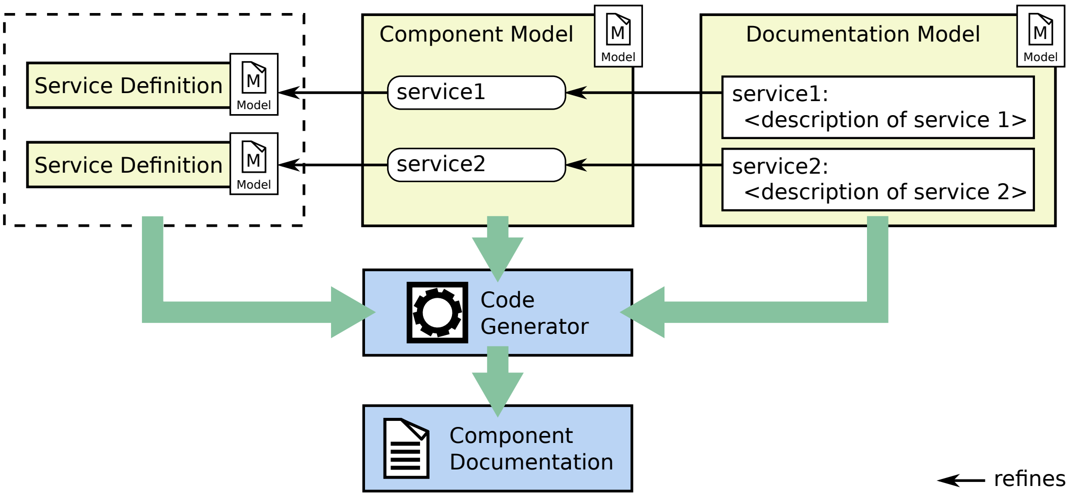


Figure 5.17: Automatic generation of documentation based on the documentation and all referring models. The documentation model is created by the component developer. He adds human-readable descriptions to component services. The code-generator uses additional information from the referenced models to automatically generate a full documentation for the system compositor role.

With the documentation model and all models that are referenced from there, the component hull is complete: the technical models (service endpoints with properties, service definitions, communication objects, pattern, etc.) and the human-readable documentation as described in section 5.2.6. From this information, a complete component documentation can be generated (Fig. 5.17) to support the system compositor role in component selection and system composition.

5.4 Building Robot Applications: Compositions

This section describes the structures and meta-models at composition Tier 1 (Fig. 5.1) that are used at Tier 3 to compose robot applications from existing functional building blocks (software components). These building blocks have been modeled and provided in the previous section (section 5.3). System composition consists of four steps, each with its own meta-models: (i) expressing the application's needs in the service wish list, (ii) component selection and retrieval, (iii) configuring the system using the selected components, and (iv) their deployment to the robot. The next sections describe these steps.

5.4.1 Service Wish List

The system compositor uses a wish list of services to model the needs of the application with respect to services (Fig. 5.18). Each entry in the wish list instantiates a service definition and refines its service properties to express the needs towards that service. An entry of the wish list is called a "service wish" or "wish". The wish list serves as input to assist the system compositor in selecting suitable components. It also serves as input to verify the composed system (see section 5.4.3).

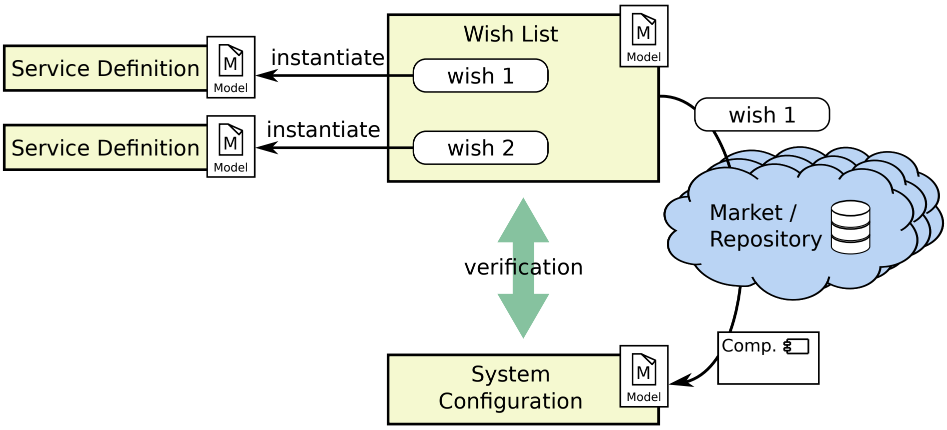


Figure 5.18: The service wish list instantiates service definitions to express the needs of the application with respect to services.

A wish list entry takes the perspective of a service sink, thus using constraints to express the needs towards a service source (see section 5.3.3). It is assumed that for every source there is also a corresponding sink (see section 5.2.1). The wish is thus indirectly expressed towards the sink as well, since the connection between these two must be valid, too. The existence of a corresponding source and the valid connection between source and sink is checked as part of verification during system configuration (see section 5.4.3).

Service Wish List Meta-Model

The meta-model to express a service wish list (Fig. 5.19) consists of several wish list entries (*ServiceWish*). A wish list entry in the meta-model is like a *ComponentServiceSink* from the component meta-model: It consists of a reference to a *ServiceDefinition* and refines the properties (*PropertyRefineWish*) of that service definition (see section 5.3.3). The wish list entry also holds a name for its identification and a description for documentation purposes.

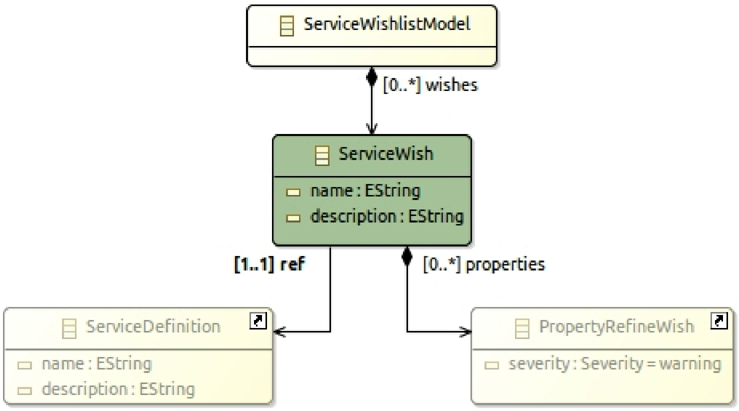


Figure 5.19: The service wish list meta-model. It instantiates service definitions and thereby expresses the needs of the application with respect to services.

5.4.2 Component Retrieval

This section covers necessary actions for retrieving a component for composition from a methodical and from a modeling point of view. It includes the selection of components based on the wish list and the management of the retrieved components to maintain consistency with the wish list.

Component Selection

At this point in the composition workflow, all information is available to select a component from a repository. The system compositor has expressed and modeled the application's needs in the wish list. Component developers have expressed the service offers in the component model. All components are available in a "component market" or repository. The system compositor is now supported in component selection: Based on the models, the supporting tool can now match the expressed needs with the available components in the market. It will present the components that meet the needs to the system compositor. The system compositor will select the final component and compose it to the application.

Component selection is not fully automated. It is not the goal to assemble the whole application fully automated based on the expressed needs alone. There will always be reasons for deciding on one component over another that can only be taken by the system compositor when considering the alternatives. His decision on one component over another may be of technical or non-technical nature. The goal is to provide adequate support to the system compositor and only present the few components that match his needs, but distinguishing them from the many others that do not. He can be supported by trading off the options based on the needs as expressed in the wish list. Supporting the system compositor is necessary, since otherwise, he will get lost in the overwhelming number of available options. He then risks selecting an incompatible or incomposable component but might discover this (too late) at run-time.

Component selection compares an entry of the wish list with the set of available components

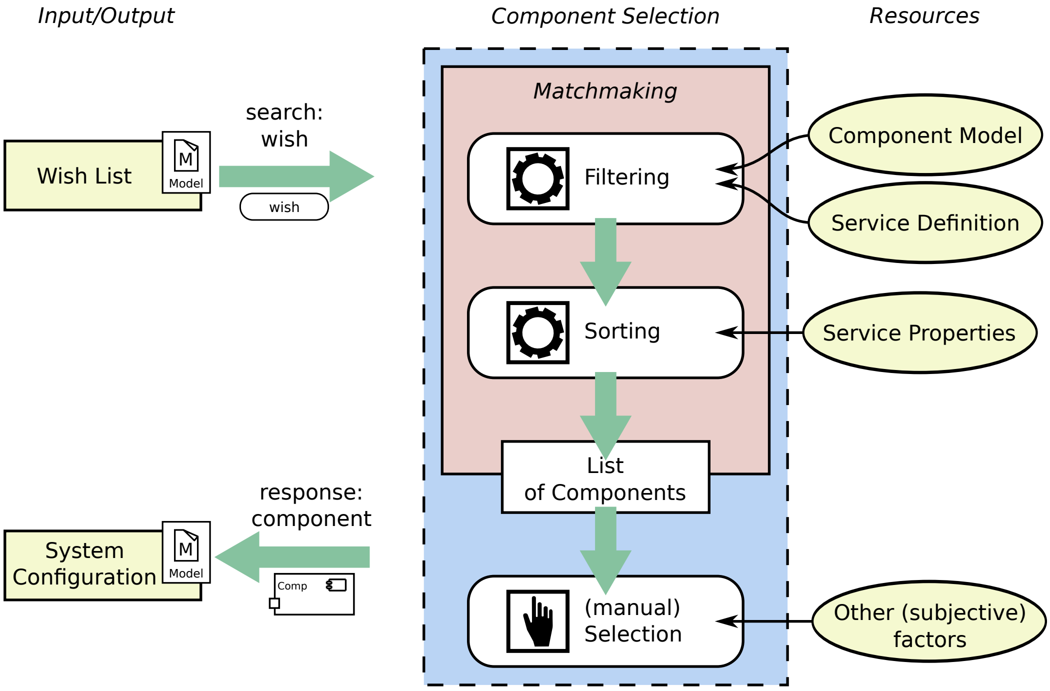


Figure 5.20: Component selection compares an entry of the wish list with the set of available components and applies a filter. The result of the filtering step is a list of components that offer the "wished" service. They are sorted and presented for manual selection by the system compositor. Figure inspired by [Lud03].

and applies a filter. The result of the filtering step is a list of components that offer the "wished" service. These are sorted and presented for selection. The component selection process takes a single wish list entry as input (Fig. 5.20). Matchmaking is applied on each service endpoint of the available components. It uses a two-step syntactic and semantic filter with signature matching on the service definition and the constraints evaluation on the service property. The matchmaking process is described in the next section in detail. From the resulting list, the user can select the final component. After that, the next step is to create the system configuration model to build the application. While doing so, the wishes must be maintained with the selected components to keep track of fulfilled wishes and of components that yet need to be retrieved.

Matchmaking Matchmaking follows the basic strategy of the F-Match algorithm [YL07]. It takes the available components, prunes the list of incompatible components and sorts the remaining components (Fig. 5.21). F-Match consists of two steps: filtering and sorting of results.

Filtering. Filtering prunes out incompatible components. Signature matching [Bac+02] is sufficient for service definitions at this stage. The signature of the service is the service definition. It is thus sufficient to return all services whose name equals to name of the service definition that is instantiated in the wish lists.

Sorting. The remaining components are sorted using the following metric: Evaluate all con-

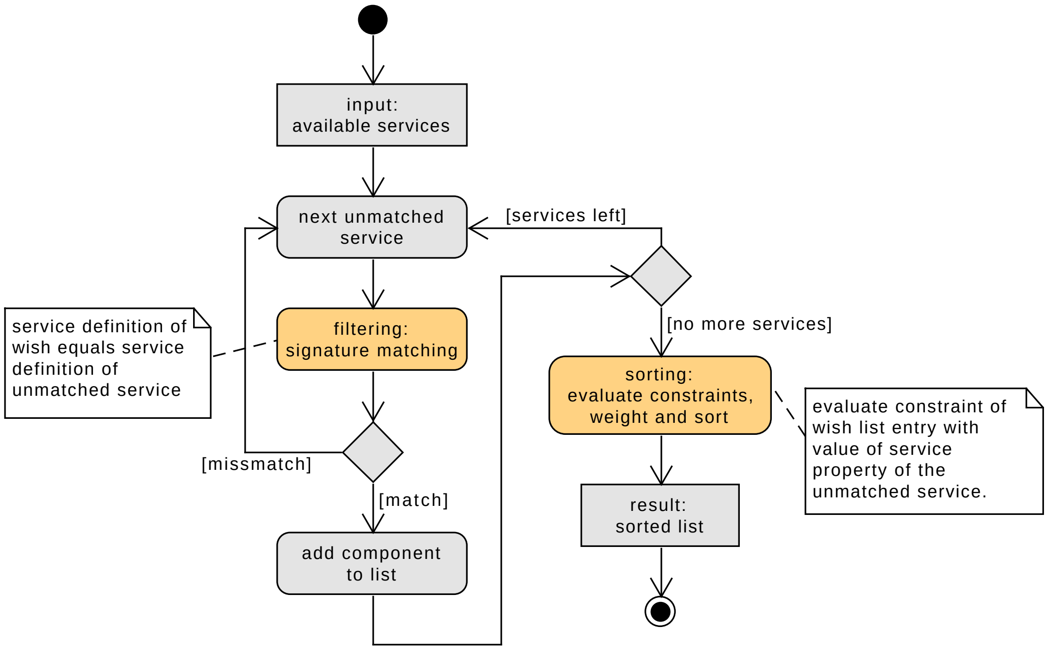


Figure 5.21: A UML Activity Diagram to illustrate matchmaking. The matchmaking process applies signature matching and sorts the remaining components based on constraints evaluation (based on [YL07]).

straints (see section 5.2.4) from the wish list entry on service properties. Count the constraints that evaluated to true and divide the sum by the total amount of constraints that were evaluated. Thereby, weigh the constraints with severity "info" three times, "warning" two times, and "error" one time. The severity is described and discussed in section 5.3.3. Listing the components in descending order will first show composable components with no warnings, followed by incomposable components with warnings, and then components with errors. Knowing about the latter gives a chance to carefully consider the application's needs. The compositor may lower the needs to use an existing component instead of building a new one that fully matches the initial needs.

An alternative to the suggested approach is to prune out mismatching service properties with severity "error" during filtering, then filter remaining results as described. Using this method, however, the compositor could not choose a component that might lower the overall performance but that still would be a composable component.

On Failed Matches An often discussed problem in matchmaking is how to deal with failed matches (e.g. [KP08; YL07]). Failed matches are services that do not match the needs. Yau [YL07] describes hard and soft reasons for failed matches. There is no way to overcome hard failures, e.g. incompatible interfaces. Soft failures might be overcome as a service might "closely" match the needs. From a system compositor point of view, it makes sense to know about softly failed matches and not to hide them. The compositor must find a match for each wish since otherwise the composition is incomplete. Presenting also the soft failures gives the system compositor the chance to adopt the needs such that a matching component is found. For example, if it turns out that no component satisfies the expressed needs, it might make sense to use an existing component that performs less well instead of facing costs, time, and complexity to develop an individual and perfectly matching component. In the end, this is a trade-off between the application's requirements and what is available. Only the system compositor can make this decision.

A possible extension to the presented structures would be to express ranges or alternatives in the service properties. For example, the need for accuracy of a localization service could specify the optimal accuracy, but could also specify the absolute minimum accuracy. This would soften the problem of failed matches. It provides more support to the system compositor, but the problem remains: The compositor must select a less suitable component, modify the application's needs, or trigger the development of a custom one in case there is no suitable component available that fully matches the needs. Supporting the system compositors with suggestions how to adapt the needs should be one of the next steps in future work.

Extending the Wish List: Implicit Services

A newly selected component often comes with additional service sinks. So, additional components are needed that offer the corresponding sources. We call these additional services "implicit services", as they are not expressed as an explicit application need in the wish list. They come implicitly with a selected component when fulfilling a wish. Once the selected component is removed from the application, the implicit services might no longer be required. It is desirable to support the user in managing implicit wishes and in selecting components that match these implicit wishes.

All information required to manage implicit services is available via the models. Supporting the user is thus considered a matter of tooling. The proposed structure can be used to realize this by maintaining the wish list dynamically: For each mandatory sink of a newly selected component, the component's refined sink endpoint can be transformed and added to the wish list as new entry. That is, take "copy" the service sink model to to the wish list. The implicit wish thus becomes explicit in the wish list. Extending the wish list might be done fully automated or semi-automated by providing suggestions to the user. Based on the newly added wish, the user can then select an appropriate component as described. There must be, however, a distinction in the meta-model between explicit wishes and implicit wishes to identify and track the implicit ones when they are no longer required. Extending the wish list keeps all wishes (explicit or implicit) in one place, enabling the compositor maintaining an overview.

An alternative approach is to use system verification checks during the system configuration step (section 5.4.3) to detect dangling (unconnected) service sinks and start component selection from there. This approach keeps implicit wishes invisible since it does not explicate them in the wish list. This can be seen both as positive and negative argument. The second approach addresses the management of wishes in a later step of system composition in which the concern is to configure and assemble the application rather than selecting components: The wish list is the first step in system composition, while the system configuration is a later step.

From a system compositor point of view, it is desirable to maintain an overview of the wishes and maintain all wishes in the wish list. Since the wish list represents the needs of the application with respect to services, it is desirable to have full control there. Since explicating information is a fundamental principle in this thesis, extending the wish list is considered the most suitable approach.

Further research into implicit services, their consequences to the application and how to manage them is required nonetheless. Implicit wishes, for example, might bring additional "implicit implicit wishes", thus creating a chain of wishes. Along these chains, needs might add up or build up to unnecessary indirections and eventually even duplications.

Maintaining the Wish List: Service Realization Meta-Model

Once the components have been selected, the wish list still needs maintenance for two reasons: first, as kind of a check list to know which wishes are fulfilled and, second, to keep track which wish is fulfilled by which component. After a component was selected based on a wish, the wish cannot simply be marked as "done". The wish is only fulfilled when the selected component was instantiated in the system configuration (system configuration, section 5.4.3). If a component was selected but was not instantiated, it cannot be detected that the wish is no longer fulfilled. It thus requires a mapping between the elements of the wish list and the services of component instances during the system configuration step (section 5.4.3).

We call the mapping between a wish and a service of a component instance the "wish list realization". That is, a service of a component instance is said to "realize" or "fulfill" a wish. The mapping cannot be derived from the wish list and system configuration alone (Fig. 5.22).

The mapping is trivial in the case that only one wish exists for one instance (Fig. 5.22, top). But this is not the case in every situation. Suppose there are two wishes for the same service definition. For example, a front laser and a back laser to drive forward and backward, both of service definition "laser" (Fig. 5.22). The service endpoints must exist in component instances in the system configuration. By only matching between wish list and system configuration, it is not clear which service endpoint in the system configuration realizes which service wish. The effect is even stronger when the components come from different vendors with different diversity of performance. A similar situation exists when a component comes with a wish that is already satisfied by another component. This may be the case in multi-purpose components. For example, a simulator component that brings a lot of services of which not every single service is required.

Even though it is not possible to create the mapping automatically, the effort to create and maintain this information can be lowered and semi-automated via tool-integration. For example, by providing auto-completion, model-templates, and other assistants. This can relive from the burden that is put on the developer to maintain another model with only few but necessary

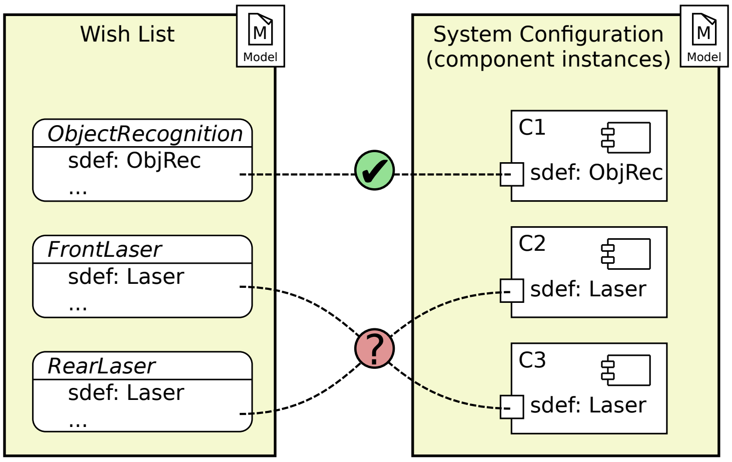


Figure 5.22: The mapping between wish and component instance that fulfills this wish cannot be derived from the wish list and system configuration alone, thus it requires modeling of wish realizations.

information.

The service realization model (Fig. 5.23) links service wishes from the wish list with services of the component instances as modeled in the system configuration model (see Fig. 5.22). Using the realization model, the system compositor can express which service wish is "realized" by which component service of a particular component instance.

A *ServiceWishRealization* element realizes the link of a service wish with a component service. Three elements are necessary (Fig. 5.23): a reference to the wish (*ServiceWish*), a reference to the component instance, and a reference to the service endpoint that fulfills a wish (*ComponentInstance* and *AbstractComponentService*). Referring a component alone is not sufficient. Referring a component *instance* is necessary since a component might be instantiated twice. Referring the instance alone is not sufficient either. The endpoint must be referenced since the instance might have two endpoints of the same service definition (e.g. a simulation component). Which of the two endpoints is used cannot be derived from the instance alone.

5.4.3 System Configuration

The system configuration step takes a software architecture point of view on the application. It puts the selected components together: It creates instances of components, wires the components to set up the final architecture, and uses variation points to parameterize the components for use in the application. The deployment model will later map these component instances to execution units (section 5.4.4).

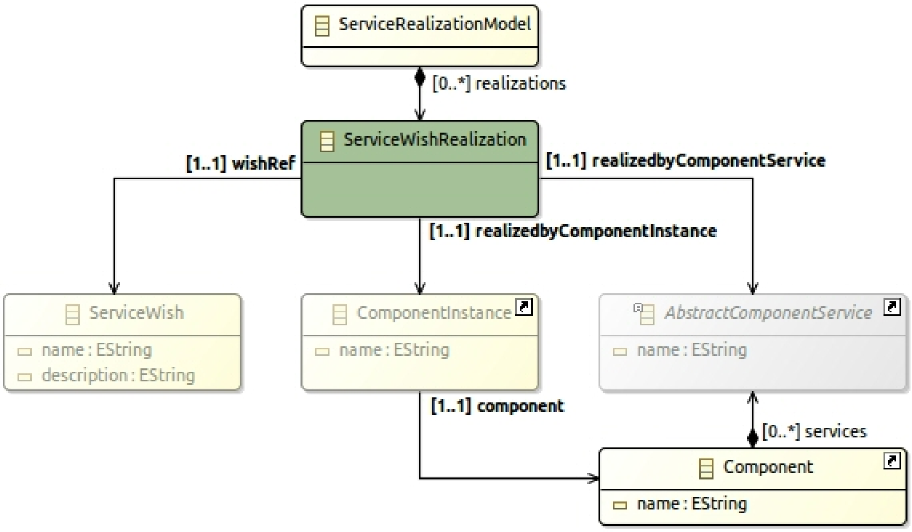


Figure 5.23: The service realization model (Fig. 5.23) links service wishes from the wish list with services of component instances in the system configuration. This expresses that the services realize or "fulfill" the wish.

System Configuration Meta-Model

The system configuration meta-model (Fig. 5.24) holds the name of the application being modeled. It consists of component instances and connections between their service endpoints.

In analogy to object-oriented programming, a component instance (*ComponentInstance*) represents a concrete run-time occurrence of a component. Component instances are created for two reasons: to use a component more than once (e.g. distinguish between two instances of a laser scanner: front and back) and to have a new distinctive element to annotate values for parameterization. The component instance takes a name to identify it in the context of the application (e.g. distinguish between front and back laser). The instance refers to the component it instantiates (*ComponentInstance::component*). Regarding component parameters, the instance refers to the parameter model that assigns values to variation points (*ComponentInstance::instanceParameter*, section 5.4.3).

Connections link ("wire") the service endpoints of the components: A *Connection* refers to two pairs of component instance and service endpoint. One pair for each connection end. That is, one end references *instance0* and *endpoint0*; the other end references *instance1* and *endpoint1*. Connections are not directed since the direction is defined through the linked service endpoints (source and sink, section 5.3.2). A connection only links a pair of service source and sink endpoints. Furthermore, it only links the endpoints of components for which instances exist. A connection can only link endpoints that are contained in the component model of the instance.

A connection must reference both to the instance and to the endpoint of the component to which the instance references. From referring to the endpoint alone, it is not known which instance the endpoint belongs to in case there are multiple instances of a component (instance

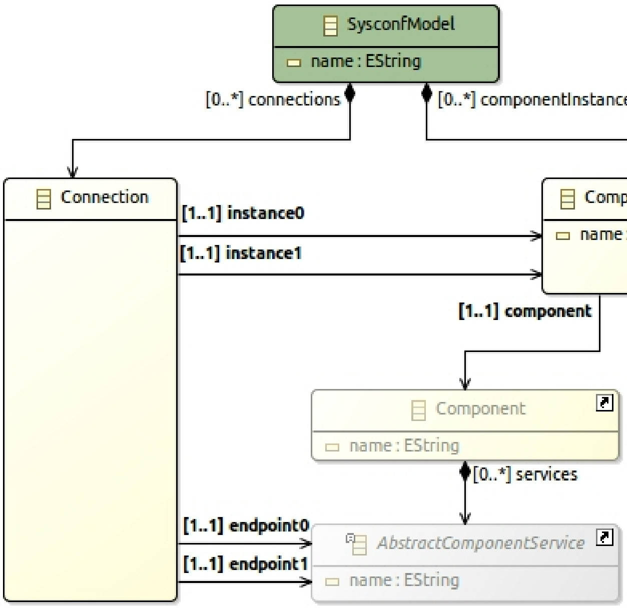


Figure 5.24: The system configuration meta-model consists of component instances and the connections between their service endpoints.

association problem, Fig. 5.25).

A component instance does not refer to or instantiate service endpoints. At this stage of the presented concept, there is no need for a refinement of endpoints. This might change when annotating additional information to service endpoints becomes necessary at composition-time; endpoints might then be instantiated within component instances as well.

Composition Parameter Meta-Model

The component model expressed the variation points of a component (sections 5.2.5 and 5.3.5). The composition parameter model is used to assign final values to the variation points of component instances. The according composition parameter meta-model is illustrated in Fig. 5.26. Each *CompositionParameter* refers to the component parameter to be refined (*ParamReference*). It also creates an attribute to assign values (*AttributeValue*). The reference to the component parameter either points to an internal parameter (via *InternalParamRef*) or to a parameter set (via *InstanceParamRef*). This is necessary since the component model can use both parameter types. The attributes (*AttributeValue*) consist of name–value pairs that refer to the attribute definition (*AttributeDefinition*) and assign a value (*AbstractValue*).

In structural building blocks, parameter sets were said to group parameters, but the groups are not visible in the system configuration when parameterizing a component instance. The parameters lie flat within the instance. The original intention of grouping parameters is to keep together a group of commonly used parameters in one unit of composition (section 5.2.5). This is no longer necessary at this step.

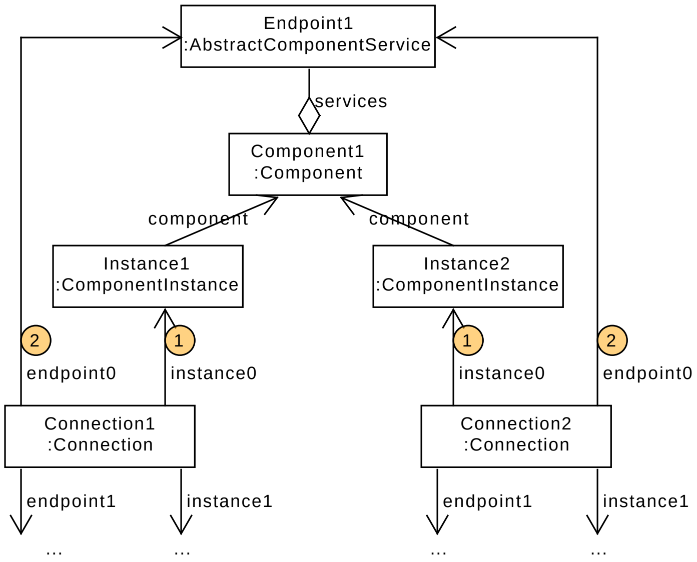


Figure 5.25: A UML Object Diagram to illustrate the instance association problem: A connection must refer to both the instance itself ① and to the endpoint of the component that the instance refers to ②. From referring to the endpoint alone ②, it is not known which instance the endpoint belongs to in case there are multiple instances of a component.

Only the parameters that are used in the component model of the particular instance can be refined. Values can only be assigned to attributes that exist within the referenced parameter. The type of the value must match the type that is defined in the corresponding attribute.

Verifying the Composition

In context of this thesis, verification means to check the "fulfillment" of all wishes in the wish list. Verification checks the composability along the horizontal and vertical composition axes (see section 3.3.1). Primary input to the verification is the system configuration model (Fig. 5.27). Firstly, the composability is checked between the application's needs and the composed application. This means to check the wish list with the system configuration model to ensure that each wish is fulfilled. Secondly, the composability is checked between the connected service endpoints to ensure that instances with service sinks are connected to a suitable suitable service source. All service endpoints that fulfill a wish must be connected to a corresponding endpoint; they cannot be left "dangling". This ensures that the wish is complete.

The verification process (Fig. 5.28) is like the matchmaking for component selection (section 5.4.2). For each source–sink pair from the wish list realization model and system configuration model (Fig. 5.27), the composition is checked by applying signature matching and by evaluating the constraints on each service property (see section 5.4.2).

If there is a single mismatch in signature matching, the overall verification process fails. In

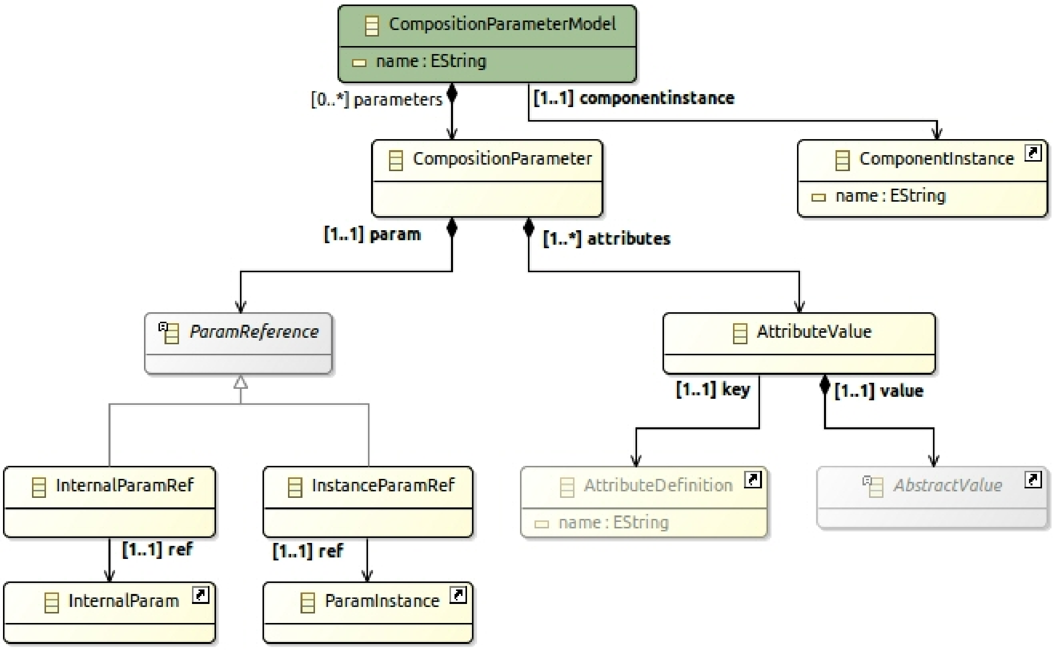


Figure 5.26: The composition parameter meta-model assigns final values to variation points that were expressed in the component model of a component instance.

case the signatures match but the constraints evaluation on service properties fail, it depends on the severity that was assigned to the service property (see section 5.3.3). In case of severity "error", the overall verification will also fail. In case of "warning" or "info", the verification succeeds, but the message that is returned by the evaluation function is displayed to the user. The severity is typically assigned by the system compositor in the wish list. Only the system compositor is able to decide on the actions to take in case a need is not met. The rare cases where a severity may be assigned by the component developer are described and discussed in section 5.3.3.

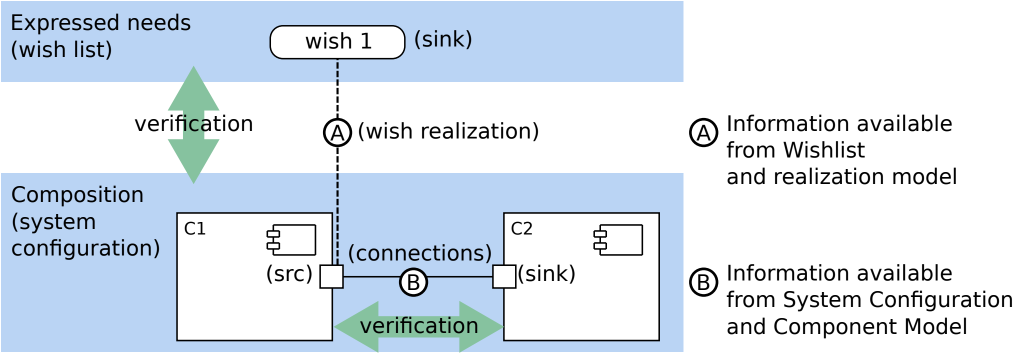


Figure 5.27: Verification of the composition: Composability is checked Ⓐ between application needs as expressed in the wish list and the composed application as modeled in the system configuration, and Ⓑ between connected service endpoints within the system configuration.

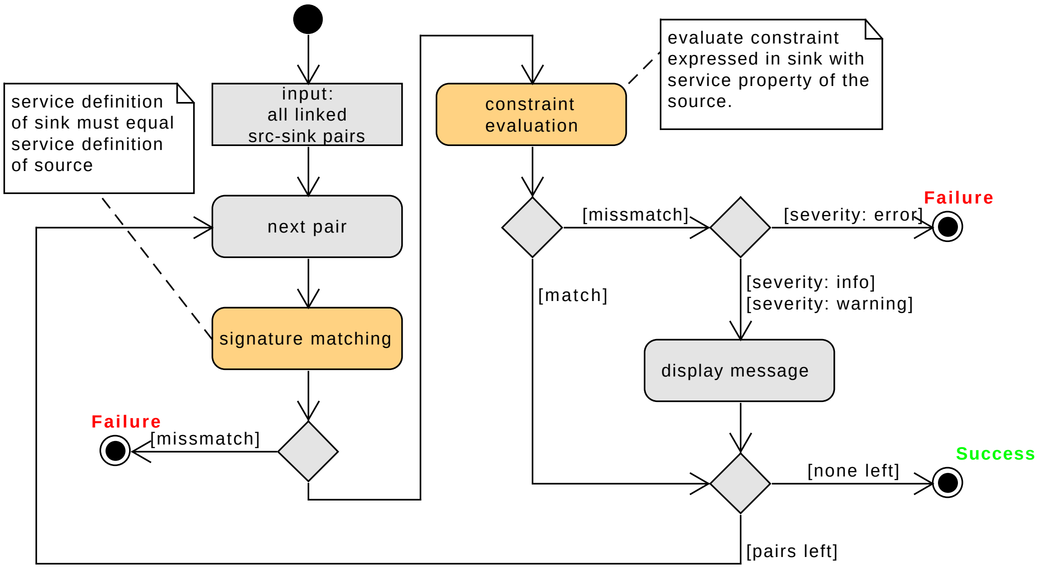


Figure 5.28: An UML Activity Diagram to illustrate how the composition is verified with respect to services. For each source–sink pair from the wish list realization and system configuration, the composition is checked by applying signature matching and evaluating the constraints on service properties.

5.4.4 Deployment

The deployment step is the last step before executing the application. It includes creating the deployment model that maps software component instances to the device(s), on which they will be executed. By a "device" we refer to the computer that will run components. Finally, there is the transfer of the necessary artifacts to the device to run the application. The transfer is realized by the tooling based on the deployment model and described in chapter 6. The deployment step can be considered as the handover from design-time to run-time.

The deployment meta-model (Fig. 5.29) is aligned to the deployment structure in UML [OMG15b]. It contains the basic elements that are required to model how a component instance maps to a *Device*. The device element includes additional attributes that are relevant to transfer the files. For example, the IP address, the login name, the target directory, and the mode of transfer. The *Deployment* is the element that performs the mapping via linking a *ComponentInstance* to a *Device*. A particular component instance can only map to one device, but several component instances might map to the same device.

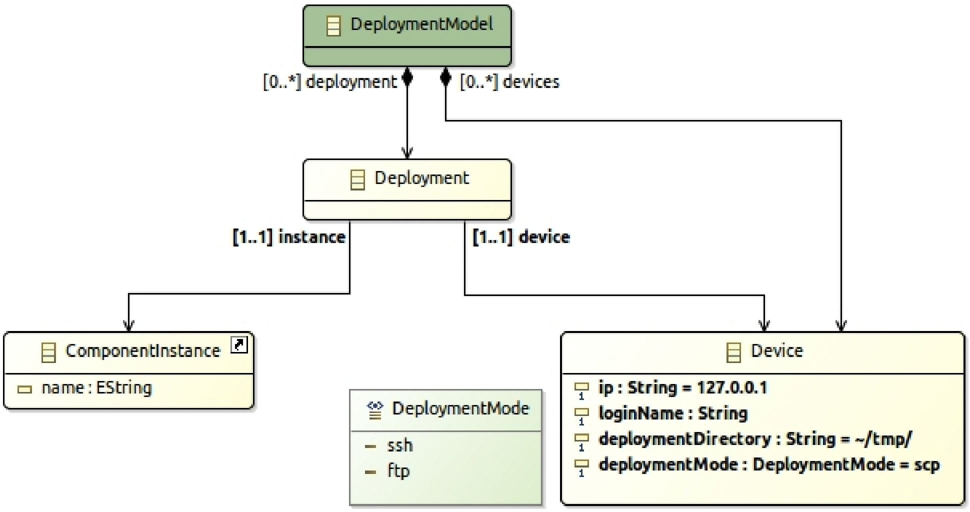


Figure 5.29: The deployment meta-model is aligned to the deployment in UML [OMG15b]. It contains the basic elements that are required to model how component instances map to devices (the computer that runs the components).

The system configuration model and step in the workflow (section 5.4.3) provides a software view on the application. In contrast to that, the deployment provides a hardware view. The purpose of the deployment meta-model in this thesis is to fill the gap between the fully modeled system and its execution on the robot. Without this step, the composition workflow would not be complete. Additional views might be needed to fully model all deployment and hardware aspects of a robot. For example, modeling the physical mounting points of sensors or modeling the network topology between several devices. Modeling the physical mounting points would allow to use this information in components for the sake of coordinate transformation. Modeling the network topology would enable to use the knowledge of the proposed deployment model: use the component–device mappings to detect limitations in the bandwidth between two devices. The deployment model provides the basis for these extensions.

5.5 Summary

This chapter has presented a concrete meta-structure and the according meta-models for system composition in an ecosystem: the "composition structure". The composition structure is located on composition Tier 1. It enables the definition of domain structures on Tier 2. Service definitions are at the heart of the composition structure. They enable the separation of roles and the exchange of building blocks on composition Tier 3. The presented composition structure covers the complete workflow, right from designing service definitions to modeling components, to the composition of applications and their deployment to the robot.

The meta-models presented in this chapter build the basis to provide tools that support ecosystem participants in applying the approach. The following chapter describes one concrete realization of such a tool—the SmartMDSD Toolchain, which is an IDE for robotics software development.

6
The SmartMDSD Toolchain

This chapter presents the implementation of the approach and the contributions to the SmartMDSD Toolchain, which is an Integrated Development Environment (IDE) for robotics software.

The chapter is structured as follows. First, initial considerations are examined. The chapter introduces the realization variants and the final overall setting for the implementation. Second, the chapter describes and explains how the composition structure maps toward concrete implementations and the workflow in the SmartMDSD Toolchain to support ecosystem participants in applying the approach. Finally, the individual steps and the Domain-Specific Languages (DSLs) that are used to create and work with the models are presented to grasp the look and feel from a user's point of view.

6.1 Considerations and Overall Setting

Integrated tooling is an important requirement for system composition (consequence C7, Fig. 6.1; see section 3.3.4). It makes the composition structures accessible to users and guides them through the composition workflow. This section discusses possible alternatives to implement the composition structures within the SmartMDSD Toolchain.

It should be emphasized that the alternatives and the considerations in this section, as well as the implementation that is described in the remaining chapter, is only one of the possibly many ways to realize the approach. The decisions for the realization variant, the used MDSD frameworks, graphical versus textual modeling, the design of the DSLs themselves (in terms of language syntax, usability or use of UML elements to represent elements of the meta-models), and the embedding of models might vary in different implementations. These decisions influence the required effort and the outcome in terms of stability and usability. The underlying composition structures as they were introduced in the previous chapters, however, are independent of these decisions.

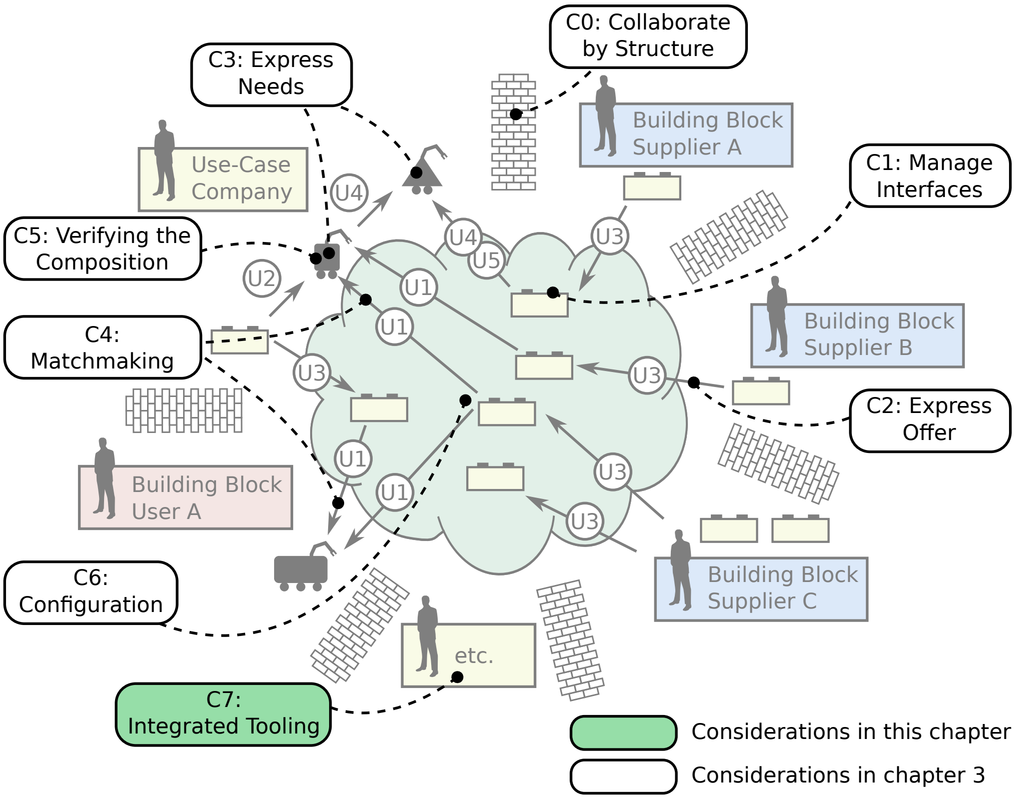

C0: Collaborate by Structure
C3: Express Needs
Building Block Supplier A
Use-Case Company
C1: Manage Interfaces
C5: Verifying the Composition
U4
U4
U5
U3
U2
U1
U1
C4: Matchmaking
Building Block Supplier B
U3
U3
C2: Express Offer
U1
U3
Building Block User A
U1
U3
C6: Configuration
Building Block Supplier C
etc.
C7: Integrated Tooling
Considerations in this chapter
Considerations in chapter 3

Figure 6.1: The consequences on an approach for system composition: This chapter discusses a baseline to implement the approach in the SmartMDSD Toolchain. (See section 3.3.4 for the summary on consequences)

6.1.1 Extension vs. Direct Realization

There are two main directions in which a realization of the composition structures might be pursued (Fig. 6.2). Tooling can be realized by extending an existing modeling approach, for example extend UML via the profiling mechanism to meet domain-specific needs. One can also directly implement the composition structures using, for example, Ecore.

UML can be extended and adapted to a domain-specific approach using the mechanism of UML profiling (variant 1 in Fig. 6.2). Using UML profiles, one gains advantage from access to existing tools. For example, Papyrus [Papyrus] is among the most widespread UML modeling tools in Eclipse. Utilizing Papyrus enables to use graphical modeling editors with no extra effort. Profiling is adequate when the elements of the intended extension have a close mapping (graphical notation and semantics) with UML such that it does not require to alter semantics. Extension works well when adding new elements based on existing elements. Extending UML also means to have available all the elements that UML offers—even if they might not be relevant or might

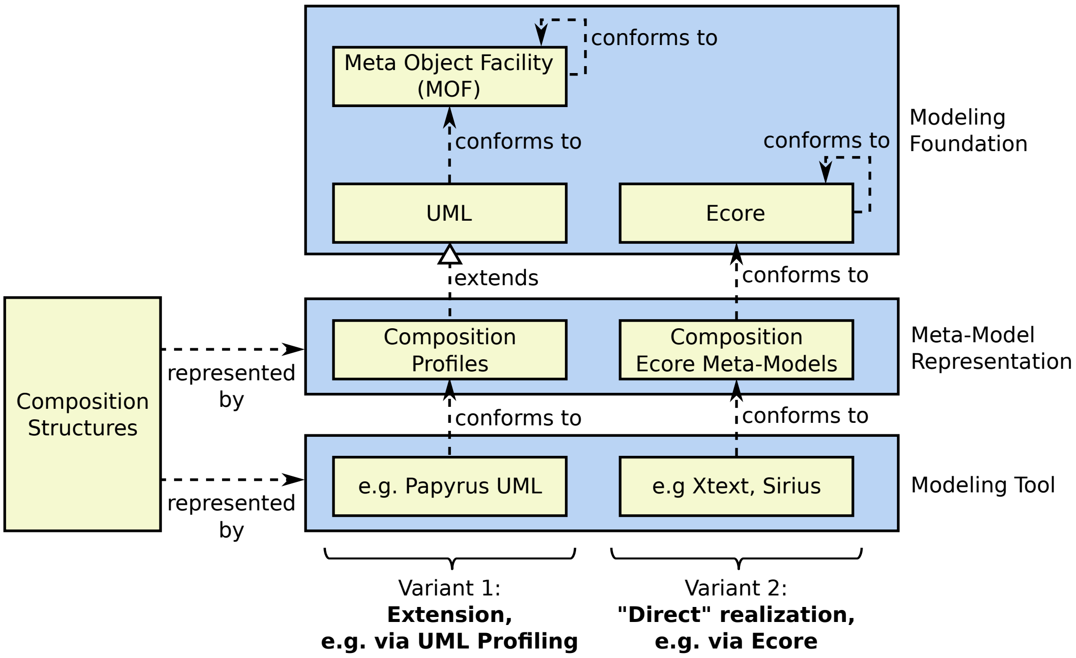


Figure 6.2: Two variants to realize tooling to represent the composition structures: extending UML with the profiling mechanism and "direct" implementation of the approach in e.g. Ecore.

not be used in the target domain for which the profile is made for. It is hard to restrict the set of original elements, i.e. it is hard to remove elements. This must be implemented by tooling. UML is part of the education in many disciplines. Since a UML profile is based on UML, it can be expected that its use and graphical notation is known to potential users. This, however, also can be a pitfall. There is a very biased perception of UML and UML is frequently topic of heated debates. As users may be familiar with UML, they might expect other behavior and semantics to extensions or model things in the way they are used to—but in a way that contradicts with the extension. Additional tooling and semantics are required. This is also discussed by Bonnet, Voirin, Exertier, et al. [Bon+16].

The direct implementation of the composition structures provides more freedom and flexibility in comparison to extending an existing approach (variant 2 in Fig. 6.2). It comes with the cost of coming up with tooling to support it. For example, Xtext [Ecli] lowers this hurdle by providing a framework for the development of textual languages. It is adequate for textual modeling but does not support graphical modeling. The Eclipse GMF and Sirius [Eclg] provide a framework for graphical modeling based on Ecore. Nowadays, Sirius can be used to come up with custom graphical editors with comparatively low effort and in comparatively short time. At the time of introducing the SmartMDSD Toolchain v2, the tool-support to come up with custom graphical modeling was still very undocumented, unsupported and unstable. Back then, custom graphical modeling would have resulted in very high effort. The decision for graphical or textual modeling in editors is a matter of coming up with adequate representations. The need to come up with new textual representations brings a certain risk to end up in a syntax that reflects "programming". The need to come up with new graphical notations brings the risk of overloaded and too "fancy" representations. One must carefully consider the target domain.

6.1.2 Technologies used in the SmartMDSD Toolchain

The goal of the implementation is to demonstrate the approach in a tool that is usable. It shall provide a high enough readiness level to reach impact and to allow adequate evaluation. For this reason, extending UML with profiles is adequate to provide the framework for the implementation. Profiles enable to access existing tooling to implement the approach with low effort but adequate stability. To also address adequate textual modeling (see discussion below), the SmartMDSD Toolchain actually uses a combination of the two presented realization variants (Fig. 6.3). It uses UML profiling for graphical modeling via Papyrus [Papyrus]. The SmartMDSD Toolchain uses parts of UML directly while other parts are extended through UML profiles. The toolchain uses "direct realization" in textual modeling using Xtext [Ecli]. It provides DSLs for both and combines them by embedding or accessing a textual model from within a graphical model. The lessons learned in realizing and applying the composition structures in this setting are discussed in section 7.6 (research question 3: "user support").

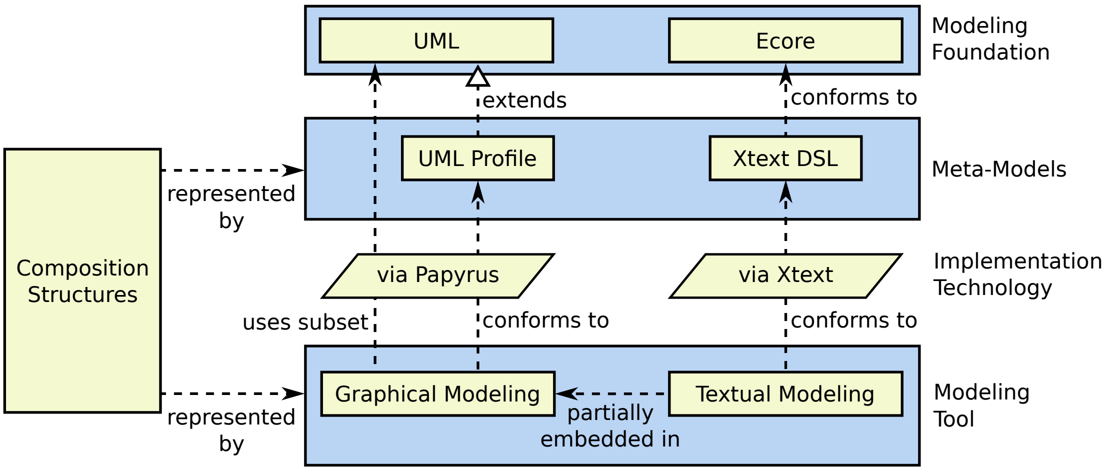


Figure 6.3: SmartMDSD Toolchain implementation technologies. Graphical modeling in the tooling is realized through extension of UML (profiling) where necessary and using a subset of UML where adequate. Xtext is used for textual modeling. In some parts of the tooling, textual modeling is embedded in graphical modeling.

An aspect to be considered when deciding for an alternative is textual versus graphical modeling. It is one of the first questions to answer when implementing model-driven tools. With UML, one is more or less tied to graphical modeling. There is no clear favor for one over the other in the SmartMDSD Toolchain. It depends on the domain and task at hand for which the DSL is made. Instead of having a clear favor, whatever is more adequate in the particular activity, role or step, is used. For a DSL, it is more important to be simple, compact and specific so that the user can focus on the task and model at hand. This is true for simplicity of a textual grammar or look-and-feel of graphical modeling but also holds true for the decision on graphical or textual modeling. Graphical modeling is *typically* more appropriate for models that require or represent graph-like structures (e.g. components and their connections). Linear structures, for example the data structure for communication with a list of attributes, is typically better represented by a textual language. Finally, graphical modeling requires more effort to implement in comparison to textual modeling.

The Eclipse Modeling Project [Eclc] (EMP) is the most widely used tooling to realize model-driven approaches and can be considered as the de-facto standard in the model-driven domain. Because of its active community support, extendable tooling, and good documentation, it is chosen over the existing alternatives in model-driven development suites (see section 2.4). Xtext [Ecli] for textual modeling and Xtend [Eclh] for code-generation are very efficient for language engineering. They have been chosen as implementation technology (Fig. 6.3) for the approach. They are well integrated into the Eclipse world and come with a lot of support, documentation, tooling and rich editors (e.g. syntax checking, proposal provider). Their integration into the Eclipse IDE provides the necessary flexibility for customization and "glue-code" around the MDSD tooling. Papyrus [Papyrus] is also well integrated into Eclipse. It helps to significantly reduce the effort for implementation. Additional glue-logic closes the gap between DSLs, the user interface, and the Eclipse platform to realize validators, checks and the bridges between models and workflow steps. Code is generated using template-based code-generators with Xtend [Eclh], applying the generation-gap pattern [Vli98] to separate user implementation from generated code.

6.2 The Composition Workflow in the SmartMDSD Toolchain

This section describes how the composition structure maps onto concrete implementations in workflow steps, workflow roles, and toolchain projects of the SmartMDSD Toolchain.

6.2.1 Overview on Workflow Steps and Projects

The approach and its models directly map onto Eclipse projects and the workflow steps design, (component) development, and (system) composition. There is one Eclipse project type for each step. It collects the corresponding models of the step in one project (Fig. 6.4).

Design: Repository Project. The user creates the structural building blocks in this step, thus "designing" the way in which components will later interact. These are service definitions, service properties, communication objects, and parameter sets that are reusable elements collected in one or more "repository projects". Structural building blocks are typically organized per application domain.

Development: Component Project. The development step contributes component models and implementations as functional parts. Each component is organized in a component project. It is driven by technical goals since the purpose of a component as the unit of

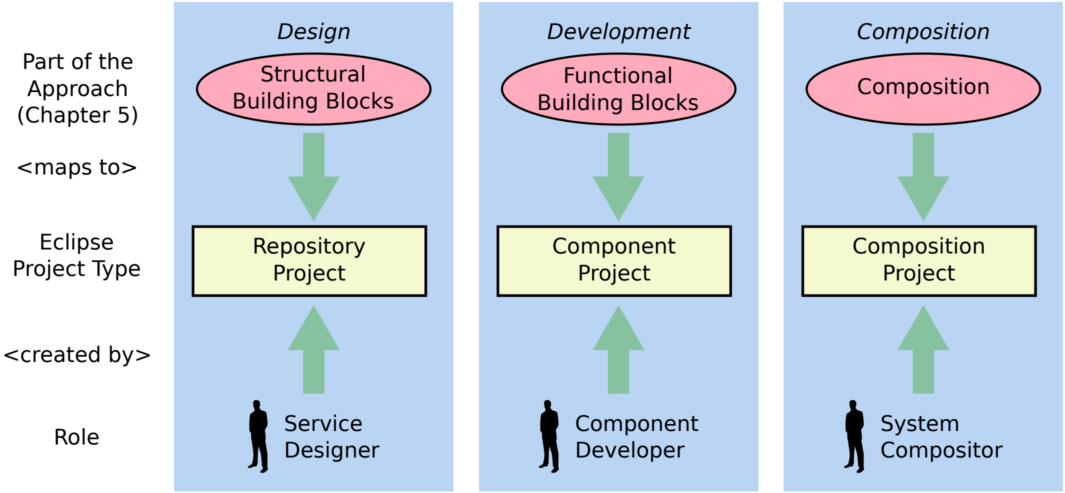


Figure 6.4: The workflow of the SmartMDSD Toolchain. It is organized in three main steps. The approach and its models map directly onto Eclipse projects and workflow steps.

composition is to provide a solution for a technical challenge. Component models use service definitions and parameter sets.

Composition: Composition Project. The composition step is driven by the particular application. One project per application holds all relevant composition models: The user creates the wish list model, selects components, instantiates them in system configuration to compose the application, and maps them to execution units in the deployment model to prepare the transfer to the robot.

6.2.2 Modeling Perspective

Each project in the workflow holds the models that are created in the corresponding workflow step using graphical and textual DSLs (Fig. 6.5). Table 6.1 provides a summary on the approach's DSLs in the SmartMDSD Toolchain, the models that they correspond to and the implementation technology.

A service repository project (Fig. 6.5, left) contains the models for parameters, communication objects, and service properties. All models, except the parameter model, are used by the service definition model that is also included in this project. The service definition model and the parameter model are used by the component model. The user creates the models using a textual DSL implemented in Xtext [Ecli].

Most meta-models directly map to their own DSL and model. Some models, however, are embedded in other models. The main model for the component and the main model for system configuration are realized in UML and contain embedded Xtext models (Fig. 6.5). Since the system configuration model arranges and wires component instances, graphical modeling via UML is preferred for this visual task. To ease the integration between component and system configuration, the main component model is implemented in the same technology (graphical

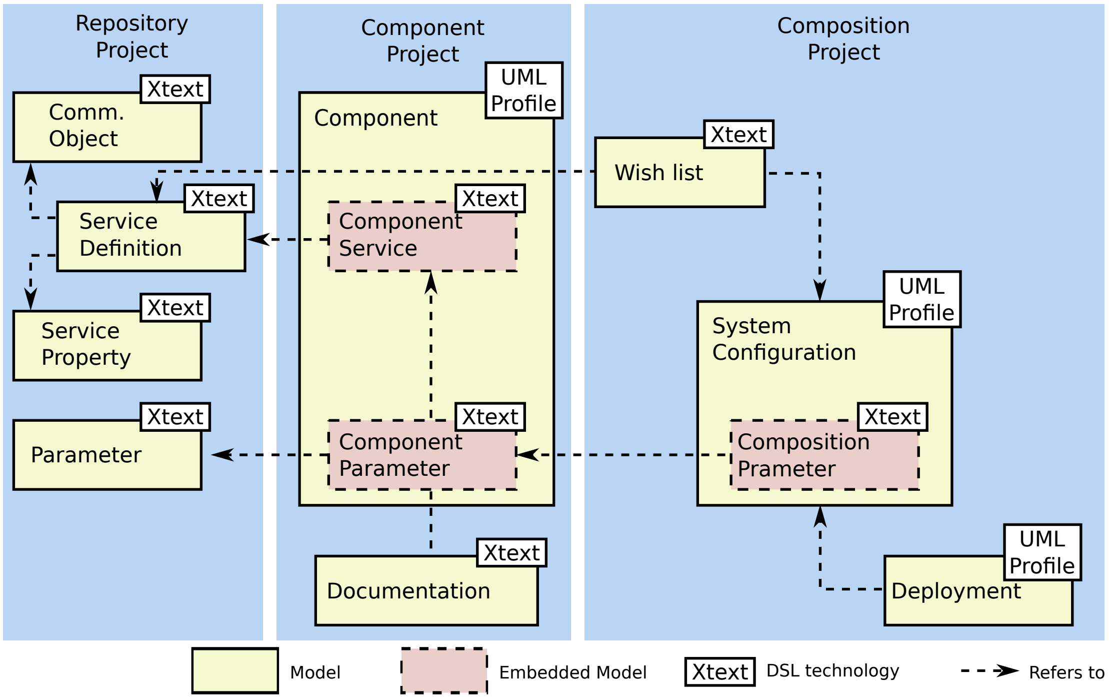


Figure 6.5: The Modeling perspective and the relations between models in the SmartMDSD Toolchain. Each project in the workflow holds the models that are created in the corresponding workflow step using graphical and textual DSLs. Most meta-models directly map to their own DSL and model. Some models are embedded in other models.

modeling using profiles and Papyrus [Papyrus]); experience with the toolchain has also proven that a visual representation of a component is beneficial to give an overview on the component. Models for component parameters, component services and composition parameters follow a linear structure and are implemented as textual Xtext DSL. They are embedded within their main models (component model and system configuration model). Services in the component model (Fig. 6.5, center) are represented by UML ports. The documentation model is a separate model and refers to component services for which documentation is added via a Xtext DSL.

The composition project (Fig. 6.5, right) contains models for the service wish list, system configuration and deployment. The wish list and the wish realization are merged to one single model that is created using a Xtext DSL; it proved to be extremely useful to have these two models together. The rather small realization model interacts closely with the wish list. The wish list uses service definition models to instantiate wishes and links them with component instances from the system composition model to manage the list. Component instances and their wiring are modeled graphically using a UML profile in the system configuration model. Each component instance holds its composition parameter model as an embedded model (Xtext DSL) to refine the parameter sets. The graphical deployment model refers to component instances to graphically map them to execution units using a UML profile.

DSL	Technology	Realizes Model, Section
Repository Project:		
Parameter	Xtext	ParameterModel, 5.2.5
CommunicationObject	Xtext	CommObjRepository, 5.2.2
ServiceDefinition	Xtext	ServiceDefinitionRepository, 5.2.1
Component Project:		
Component	UML Profile	Component, 5.3.1
→ ComponentParameter	Xtext	ComponentPrameterModel, 5.3.5
→ ComponentService	Xtext	AbstractComponentService, 5.3.1
ComponentDocumentation	Xtext	ComponentDocumentation, 5.3.6
Composition Project:		
WishList	Xtext	ServiceWishistModel, 5.4.1 and ServiceRealizationModel, 5.4.2
SystemConfiguration	UML Profile	SysconfModel, 5.4.3
→ CompositionParameter	Xtext	CompositionParameterModel, 5.4.3
Deployment	UML Profile	DeploymentModel, 5.4.4

Table 6.1: The DSLs of the SmartMDSD Toolchain, the models that they correspond to and the implementation technology. Embedded models (→) are listed below their parent.

6.2.3 Code-Generation and Artifacts

Integration of User-Code

Models are the source of structural information. While some models are only used to represent and organize structure, code is generated from most of the models using Xtend [Eclh] (Fig. 6.6). From most models, infrastructure-code and templates for user-code are generated. User-code and infrastructure-code are separated using the generation-gap pattern [Vli98]. Based on the used implementation language (C++, Java, Shell Scripts), the generation-gap pattern is realized by placing generated code and user code in separate files and linking them by inheritance (for C++, Java) or by function calls/script execution (for Shell Scripts).

Infrastructure-code provides the implementations and scripts that are necessary to connect user-code and to realize the structures or the entity that the model represents. Infrastructure-code, for example, implements against the SmartSoft framework (in case of a component) or consists of scripts to collect and transfer artifacts to the robot or manages starting and shutting down the components and application on the robot (in case of system configuration and deployment).

The user implements business logic in skeletons (empty method bodies, scripts, etc.) that are generated once from the model. They integrate with / "dock to" the inheritance structure or function call/script execution structure, so that this relation must not be set up manually by the user.

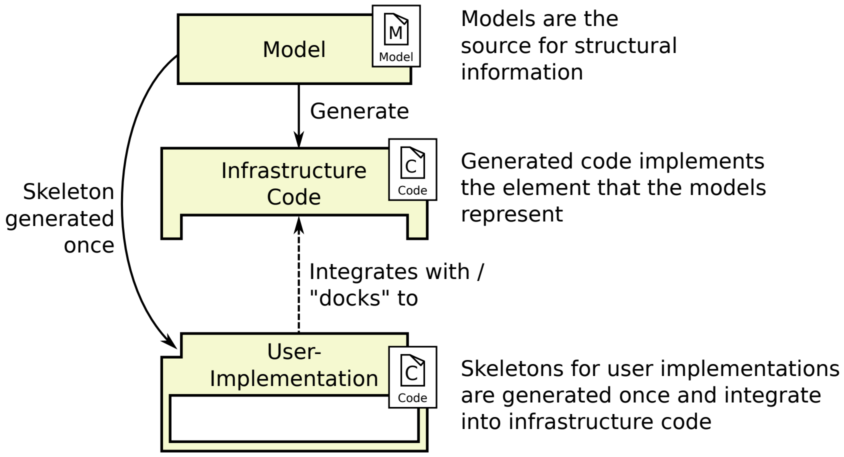

Model
M
Model
Models are the source for structural information
Generate
Infrastructure Code
C
Code
Generated code implements the element that the models represent
Skeleton generated once
Integrates with / "docks" to
User-Implementation
C
Code
Skeletons for user implementations are generated once and integrate into infrastructure code

Figure 6.6: Code generation: Infrastructure-code and skeletons for user-code to implement business logic is generated from most of the models. They are separated using the generation-gap pattern [Vli98].

Infrastructure-code is kept in separate files and directories. It is overwritten when models change. As a result, models and infrastructure-code are always in sync and changes in user implementations do not compromise the model.

Artifacts in the Workflow

There are many artifacts that are generated from models or contributed by the users throughout the workflow. They must be handed over along the workflow towards the robot. Figure 6.7 illustrates the models and the artifacts they represent or the artifacts that are generated from these models (the relations between the models are illustrated in Fig. 6.5).

From the custom constraints evaluation element in the service properties (section 5.2.4), an abstract java class is generated. The user derives from this class and implements the custom constraint function in java (see section 6.3.2). Based on the modeled data structure for communication, the communication object in SmartSoft is generated [Lut+14; Scho4a]. It generates the necessary middleware-bindings. Towards the component developer, getter- and setter-functions are generated for each element. Optionally, separated by the generation-gap pattern, more complex access-methods might be implemented by the user, e.g. to set multiple values at once while ensuring consistency, automatic transformations, etc. as described in [Scho4a].

No code is generated from the parameter or service definition model as both so far only contain structural information. The component hull includes information from these models and is generated from the component model. The component hull contains infrastructure-code to implement service endpoints, to read parameters from a configuration file, and to access them from within the component hull and user-code.

The start-up infrastructure for component instances is generated from the system configuration. It consists of scripts that manage starting and stopping of component instances as modeled in the system configuration model. The start-up infrastructure creates component instances by

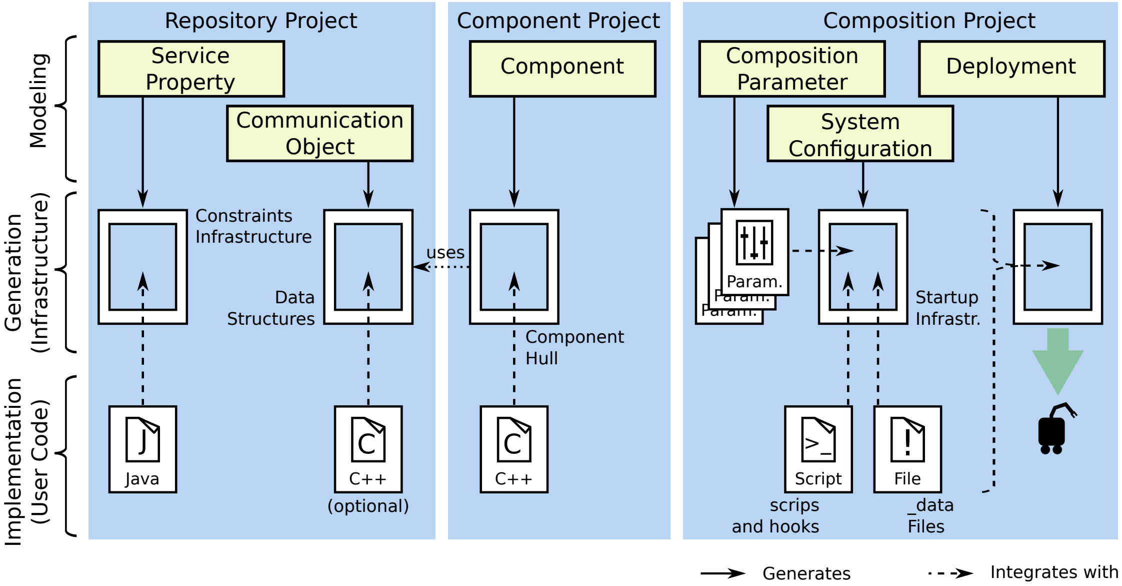


Figure 6.7: The correspondence between models and the artifacts they represent. There are many artifacts that are generated from models or contributed by the users throughout the workflow. They must be handed over along the workflow towards the robot.

launching the component under the name of the instance. The information how to wire component instances and how to parameterize components are read from the system configuration model and generated into configuration files. An instance-specific configuration file is read by the instance at startup to set the parameter values. The configuration file also contains the remote endpoints to which the component shall connect its services to. Several start hooks and stop hooks (pre-/post- and start-/stop-hooks, inspired by Towns and Hess [TH04]) are available to customize starting and stopping of components. These can be used to execute custom shell commands before and after starting or stopping an instance, for example, to start and stop additional servers on the target platform, initialize devices, etc. As part of the startup infrastructure, files can be associated to component instances that may be required for execution. For example, the SmartCdlServer in the presented use-case (section 6.3.1) uses lookup tables that are calculated offline, i.e. not at run-time. Without associating these files with the component instance, one need to copy these files manually. Another common use-case is object recognition where one loads the object database from a file.

The deployment infrastructure is generated from the deployment model. It contains scripts to collect the afore mentioned artifacts on the development computer and to transfer them to the robot's execution units as expressed in the deployment model.

6.3 Using the SmartMDSD Toolchain: A Walkthrough

This section describes the individual steps and the DSLs to create and work with the models in the composition workflow. It presents the look and feel in the individual view that is presented to the different roles.

The next section introduces a comprehensible example. The subsequent sections use it to show the necessary steps from the service definition on composition Tier 2 to component development, system composition, and deployment on composition Tier 3 (see section 4.1.1).

6.3.1 Introduction and Overview

The Navigation Stack Use-Case

The walkthrough is illustrated using the example of a navigation stack for mobile robots (Fig. 6.8). The walkthrough focuses on the obstacle avoidance part of the navigation stack. In order to be compact and comprehensible, it does not go into details with respect to localization, map building or path planning. It is a typical example of composition as it can be found in any wheeled robot. The remaining chapter shows the principle of applying the composition structures using the SmartMDSD Toolchain. To apply these principles in large-scale systems, they just have to repeated for every service or component.

More complex systems can be found in section 7.3. The navigation stack can be found in many robotics applications of the Service Robotics Ulm Laboratory and the SmartSoft World. For example, it is used in the *"robot butler scenario"* and in the *"collaborative robot butler scenario"* where robots open cupboards and operate the coffee machine (section 7.3.1). It is also used in the *"Industry 4.0 Robot Commissioning Fleet in Intra-Logistics scenario"* (section 7.3.3).

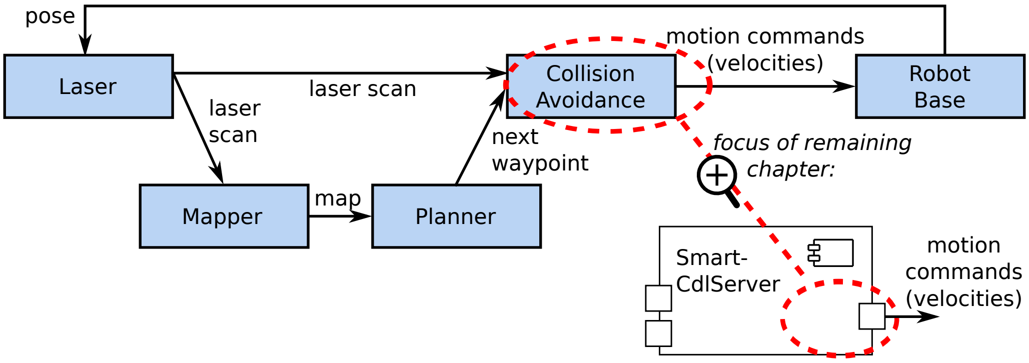


Figure 6.8: The simplified navigation stack. The example shown in the remainder of this chapter demonstrates development and composition of the simplified navigation stack, but focuses on the development of the collision avoidance component: the SmartCdlServer. Arrows denote the flow of information.

In the simplified navigation stack, the robot perceives its environment using a laser ranger (Fig. 6.8). Based on the laser scan, an obstacle grid map of the environment is generated and handed over to the planner to compute a path to the destination. The example shown in the remainder of this chapter demonstrates development and composition of the navigation stack, but focuses on the development of the collision avoidance component SmartCdlServer (available at [Sma]). The SmartCdlServer implements the Curvature Distance Lookup (CDL) algorithm [Sch98]. The algorithm takes laser scans and the next intermediate waypoint as input. It calculates for the next cycle the best combination of the translational velocity and the rotational velocity to generate a robot motion command towards the goal location. It considers the robot kinematics, dynamics and shape to avoid any obstacles. The example focuses on the output of the component: the navigation command, expressed as velocities, that is sent to the *Robot Base* component that is connected to the robot base platform (red circles, Fig. 6.8).

The Walkthrough in the Ecosystem-Context

The walkthrough demonstrates a typical situation in a robotics ecosystem. A technology provider wants to supply an obstacle avoidance component such that third parties can use it (provide and use building blocks at composition Tier 3). Before this can happen at the ecosystem level, the appropriate domain structures need to be established on composition Tier 2. In the presented example, this is the definition of robot motion commands. They are discussed, agreed, and modeled by a consortium in the domain of wheeled robots. In the toolchain, this corresponds to the action of defining services by the service designer. This includes the service-level interface of a robot motion command with a communication pattern.

A typical problem in wheeled robot navigation is given by the different types of robot motion primitives such as differential drive, omnidirectional drive, ackermann drive, and synchronous drive. While there can be service-level interfaces for collision avoidance to support all of these, the specific motion type plays a particular role in composing the components. Not all components support all variants of motion types. Composing two components with an incompatible motion type results in an invalid composition. Such motion types can be expressed via service properties as part of service definitions in composition Tier 2. All components that follow the service definition for collision avoidance are considered "component alternatives". They use the service property to distinguish themselves by the supported robot motion type towards the system compositor.

Using the domain structures from Tier 2 is a typical task of the component supplier on Tier 3. He has sufficient freedom to implement the internals of his component. For example, he may implement special selling points that are not considered at Tier 2 and makes them accessible and parameterizable via variation points. A typical task of the system builder is to find a component that supports the motion type of the robot base platform.

All steps are described from the viewpoint of the according role (Fig. 6.9). The viewpoint thus will change through the linear structure of the remaining chapter and the corresponding role is explicitly marked in each example or screenshot. This is to visualize the separation of roles. Depending on the scope of how the approach is applied, only a subset of these steps might be necessary. The order in which the steps are applied may vary as well (see also section 4.3.2).

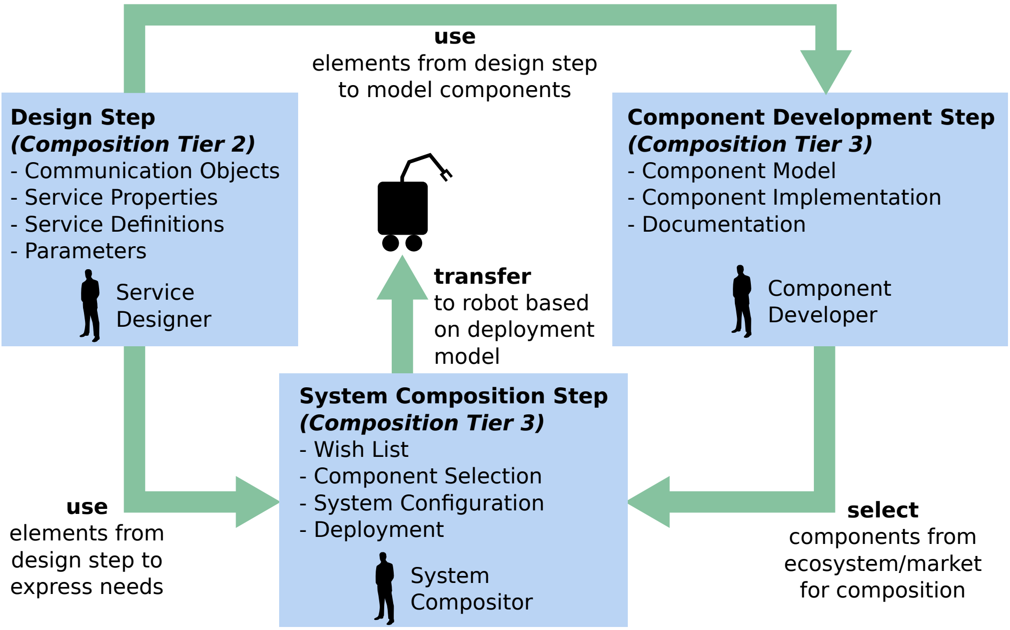


Figure 6.9: An overview on the individual steps taken during the complete composition workflow as supported by the SmartMDSD Toolchain. This figure includes a list of the models that are created in each step and are listed here for reference and overview.

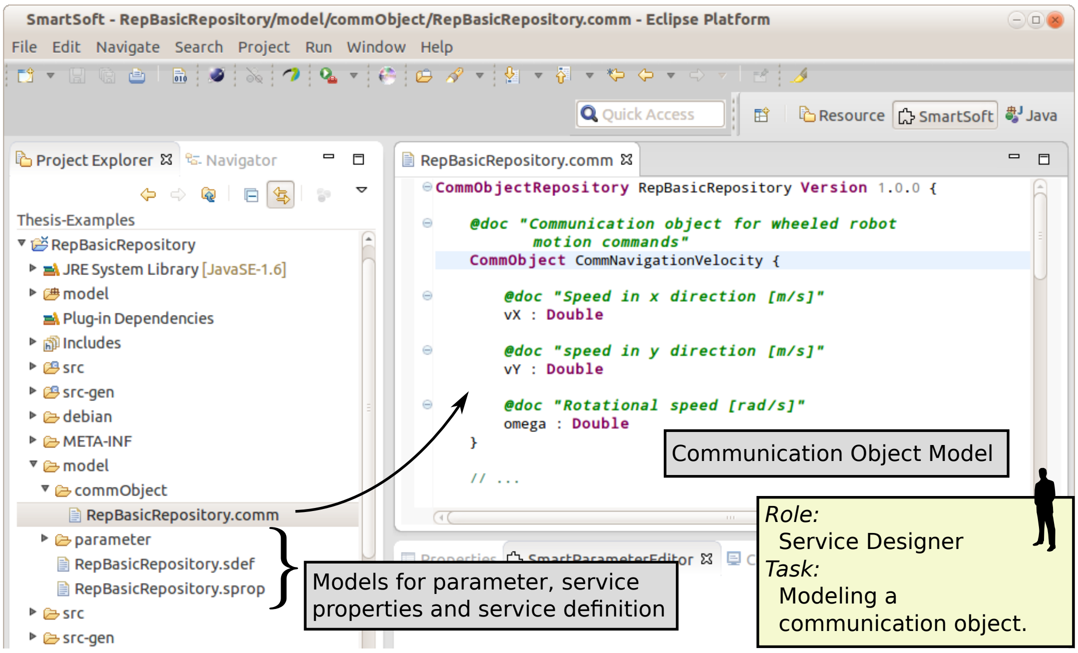
6.3.2 System Design Step

The main building blocks for reuse in components and compositions are defined in the system design step. In this step, the user models communication objects, service properties, service definitions, and parameters using textual DSLs.

Communication Objects

Figure 6.10 shows a typical view of the SmartMDSD Toolchain while creating a communication object to define the data structure that is communicated between service endpoints (*CommNavigationVelocity*). Communication objects are used in service definitions. Optional documentation might be annotated (*@doc*) to include prose text in the generated documentation.

CommNavigationVelocity holds the data structure to communicate motion commands of wheeled robots: the attributes *vX* and *vY* represent the velocity in x- and y-direction; *omega* represents the rotational velocity.



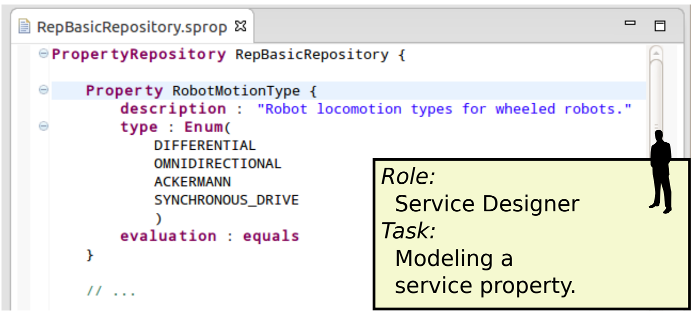
```
CommObjectRepository RepBasicRepository Version 1.0.0 {

    @doc "Communication object for wheeled robot
          motion commands"
    CommObject CommNavigationVelocity {

        @doc "Speed in x direction [m/s]"
        vX : Double

        @doc "speed in y direction [m/s]"
        vY : Double

        @doc "Rotational speed [rad/s]"
        omega : Double
    }

    // ...
```

Figure 6.10: An example of using the communication object DSL during system design to define the data structure that is communicated between service endpoints. The service definition is later composed from communication objects, among others.

Service Properties

Service properties are modeled using the service property DSL (Fig. 6.11). They provide building blocks for reuse in service definitions to express the semantics of a service on the application-level. No values are assigned yet. Values are assigned during service instantiation in components (service endpoints) and composition (wish list) when values are relevant to specify the need and offer of service endpoints. A prose text description provides information on the use and on the semantics of this element.

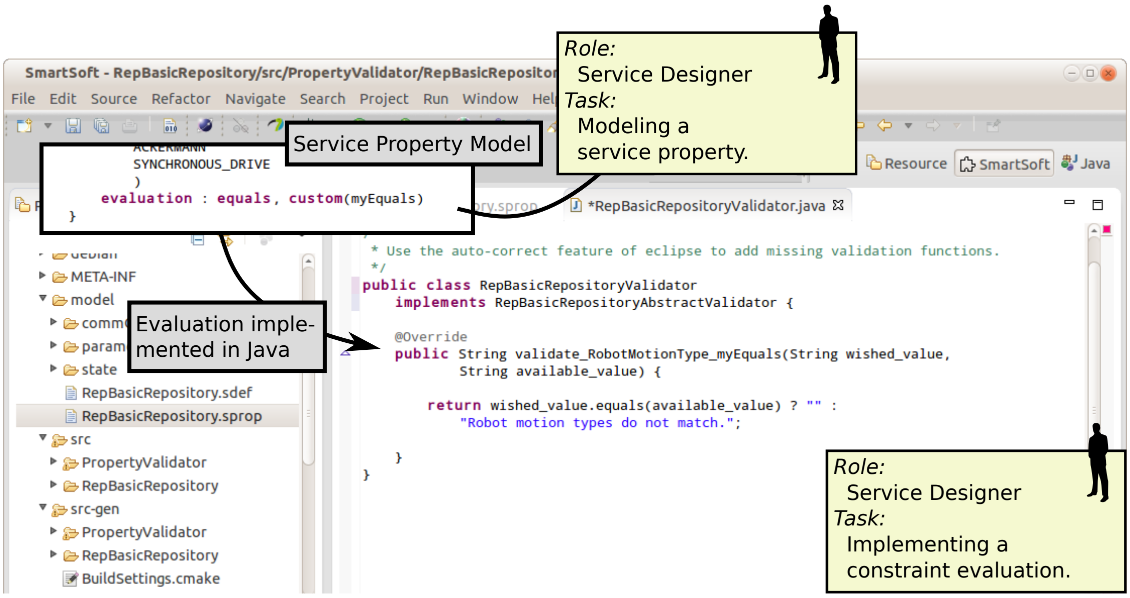
For example, the *RobotMotionType* (Fig. 6.11) will be used to distinguish between motion types of wheeled robots later. The service property thus defines possible robot motion types: differential drive, omnidirectional drive, ackermann and synchronous drive (see section 6.3.1).



```
PropertyRepository RepBasicRepository {

    Property RobotMotionType {
        description : "Robot locomotion types for wheeled robots."
        type : Enum(
            DIFFERENTIAL
            OMNIDIRECTIONAL
            ACKERMANN
            SYNCHRONOUS_DRIVE
            )
        evaluation : equals
    }

    // ...
```

Figure 6.11: An example of using the service property DSL during system design to model building blocks that express the semantics of a service on the application-level. Service definitions are later composed from service properties, among others.

Custom Constraints Evaluation Custom evaluation functions to extend the available ones can be defined in the service property model. Figure 6.12 shows the hand-made equivalent of the *equals* evaluation. From the property definition, an abstract java class is generated for implementation by the user. The evaluation function returns an empty string in case the constraint evaluation succeeded; a string containing an additional message otherwise.



```
            ACKERMANN
            SYNCHRONOUS_DRIVE
            )
        evaluation : equals, custom(myEquals)
    }
```

```
 * Use the auto-correct feature of eclipse to add missing validation functions.
 */
public class RepBasicRepositoryValidator
    implements RepBasicRepositoryAbstractValidator {

    @Override
    public String validate_RobotMotionType_myEquals(String wished_value,
            String available_value) {

        return wished_value.equals(available_value) ? "" :
            "Robot motion types do not match.";

    }
}
```

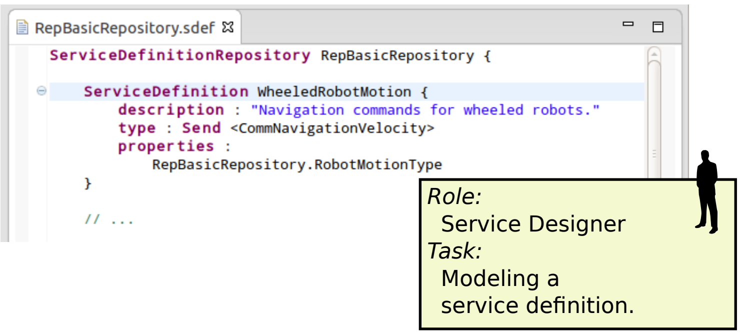
Figure 6.12: Modeling and implementing custom evaluation functions. Custom evaluation functions to extend the available ones can be defined in the service property model. It generates to a java class that is implemented by the user.

Service Definition

The service definition DSL (Fig. 6.13) during system design is a textual language that composes building blocks to describe a service: the communication data structure (communication object), the communication semantics (communication pattern), and the service properties to express the service's semantics on the application-level. Service definitions are created once for reuse in components to model the component hull and wish lists to express the needs of the application.

Figure 6.13 shows the service definition *WheeledRobotMotion* for later use in the SmartCdlServer and alternatives. It is used to command a wheeled robot to move around. It uses the *send* pattern in combination with the communication object model *CommNavigationVelocity*.

The service definition so far is generic for motion commands. However, not all components that provide or require this service are going to support all motion types. The property *RobotMotionType* will later be used to explicate with which robot motion type this service can be used with (see section 6.3.1 and paragraph on "Service Properties", page 164).



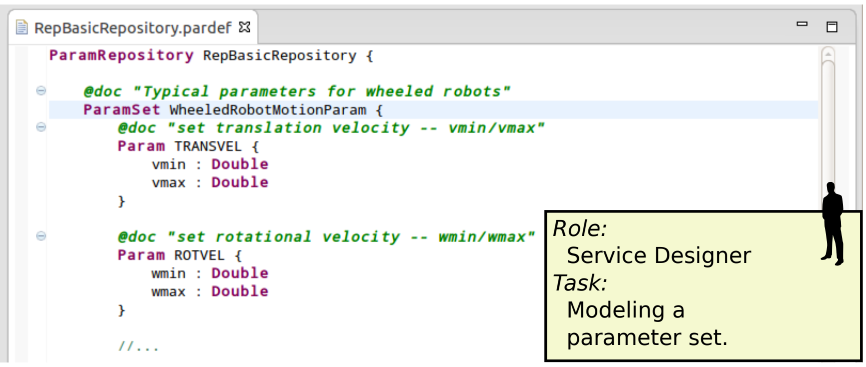
```
ServiceDefinitionRepository RepBasicRepository {

    ServiceDefinition WheeledRobotMotion {
        description : "Navigation commands for wheeled robots."
        type : Send <CommNavigationVelocity>
        properties :
            RepBasicRepository.RobotMotionType
    }

    // ...
```

Role:
Service Designer
Task:
Modeling a
service definition.

Figure 6.13: An example of using the service definition DSL to put together all building blocks to describe a service.

Parameter

The parameter DSL is a textual language to model variation points for reuse in components. Figure 6.14 shows the parameter DSL during system design while modeling the parameter set *WheeledRobotMotionParam*. It groups typical parameters that are used for wheeled robots, for example the parameters for translational and rotational velocity *TRANSVEL* and *ROTVEL*. Each of these parameters consists of two name–type attributes. They hold the minimum and maximum velocities supported by the robot.



```
ParamRepository RepBasicRepository {

    @doc "Typical parameters for wheeled robots"
    ParamSet WheeledRobotMotionParam {
        @doc "set translation velocity -- vmin/vmax"
        Param TRANSVEL {
            vmin : Double
            vmax : Double
        }

        @doc "set rotational velocity -- wmin/wmax"
        Param ROTVEL {
            wmin : Double
            wmax : Double
        }

        //...
```

Role:
Service Designer
Task:
Modeling a parameter set.

Figure 6.14: An example of using the parameter DSL, a textual language to model variation points for reuse in components.

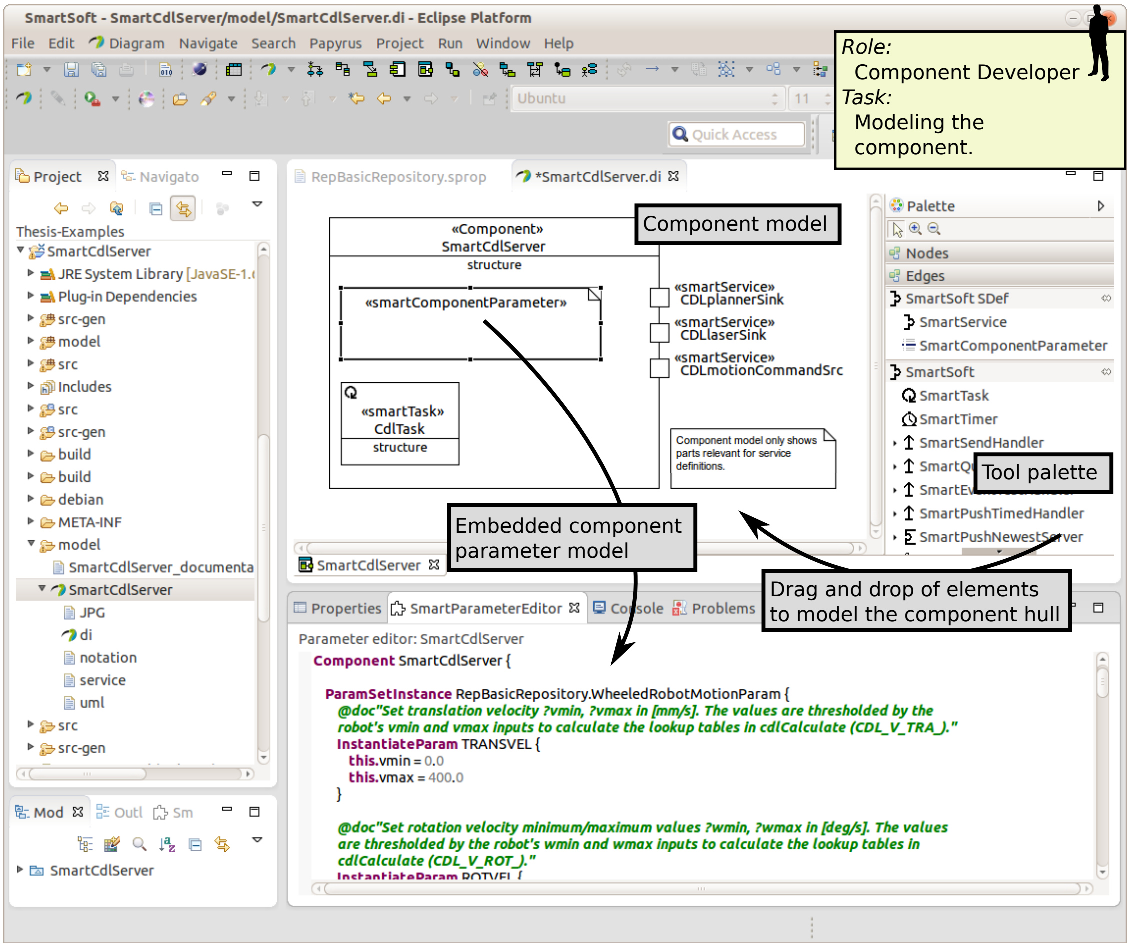
6.3.3 Component Development Step

Component development first models the component by reusing the structural building blocks from system design: communication objects, service properties, service definitions, and parameters. Once the component model exists, code skeletons are generated for adding user-implementations. Finally, the component is documented using the documentation DSL.

Component Model

Figure 6.15 shows a typical view of the SmartMDSD Toolchain while modeling a component graphically based on UML profiles in Papyrus. Stereotyped UML Ports represent service endpoints in components. A stereotyped UML comment within the component holds the parameter model that is created using the component parameter DSL. The component service DSL refines the component's service endpoints.

The component model in Fig. 6.15 shows the SmartCdlServer component for the navigation use-case. It shows the extensions of the SmartMARS meta-model [SSS09a; Ser] that are relevant for component modeling using service definitions. The modeled *CdlTask* task represents a SmartSoft SmartTask [Sch04a]. It is used to implement the business logic of the component.



Role:
Component Developer
Task:
Modeling the component.

Component model

Tool palette

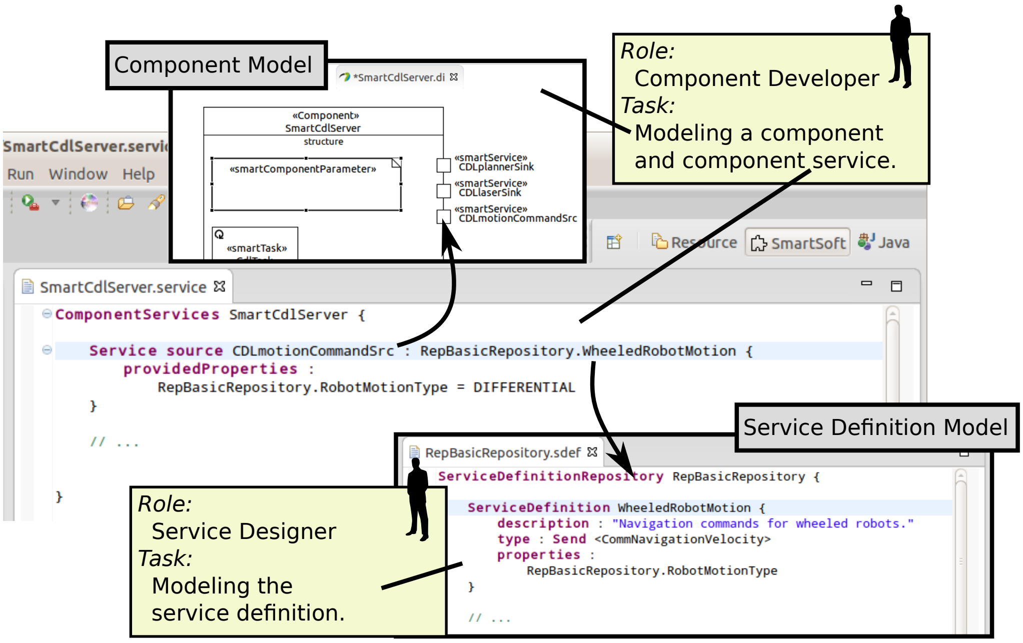
Embedded component parameter model

Drag and drop of elements to model the component hull

```
Component SmartCdlServer {

  ParamSetInstance RepBasicRepository.WheeledRobotMotionParam {
    @doc"Set translation velocity ?vmin, ?vmax in [mm/s]. The values are thresholded by the
    robot's vmin and vmax inputs to calculate the lookup tables in cdlCalculate (CDL_V_TRA_)."
    InstantiateParam TRANSVEL {
      this.vmin = 0.0
      this.vmax = 400.0
    }

    @doc"Set rotation velocity minimum/maximum values ?wmin, ?wmax in [deg/s]. The values
    are thresholded by the robot's wmin and wmax inputs to calculate the lookup tables in
    cdlCalculate (CDL_V_ROT_)."
    InstantiateParam ROTVEL {
```

Figure 6.15: A typical view of the SmartMDSD Toolchain while modeling a component graphically based on UML profiles using Papyrus.

Component Service The component service DSL refines service endpoints that were modeled as ports in the component model (Fig. 6.16). The DSL instantiates service definitions and refines their service properties (see previous section). The refinement of *CDLmotionCommandSrc* instantiates the service definition *WheeledRobotMotion* and assigns values to properties: the component implementation at hand can only work with differential drives. An example for a counterpart of this component service endpoint is shown in Fig. 6.17. A service endpoint of the SmartPioneerBaseServer component [Sma] instantiates the same service definition as service sink and assigns one of the existing evaluation functions. In this case, it must support a differential drive since the Pioneer P3DX, for which the SmartPioneerBaseServer was developed, is powered by a differential drive.



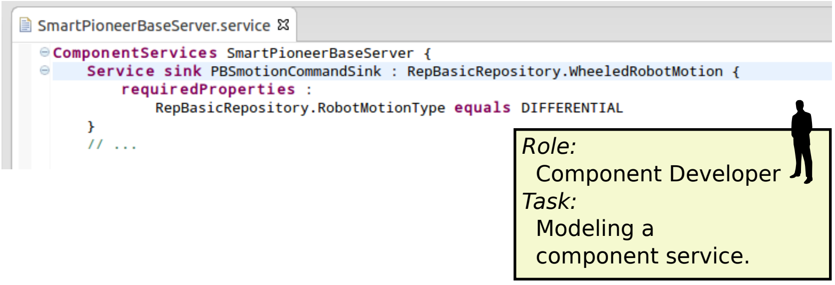
Role:
Component Developer
Task:
Modeling a component and component service.

```
ComponentServices SmartCdlServer {

    Service source CDLmotionCommandSrc : RepBasicRepository.WheeledRobotMotion {
        providedProperties :
            RepBasicRepository.RobotMotionType = DIFFERENTIAL
    }

    // ...

}
```

```
ServiceDefinitionRepository RepBasicRepository {

    ServiceDefinition WheeledRobotMotion {
        description : "Navigation commands for wheeled robots."
        type : Send <CommNavigationVelocity>
        properties :
            RepBasicRepository.RobotMotionType
    }

    // ...
```

Role:
Service Designer
Task:
Modeling the service definition.

Figure 6.16: An example of using the component service DSL to refine services of the SmartCdlServer component model (large screenshot in the background). Each modeled service refers to graphical ports (representing a component service in the component model, top of figure) to instantiate a service definition model (bottom right).

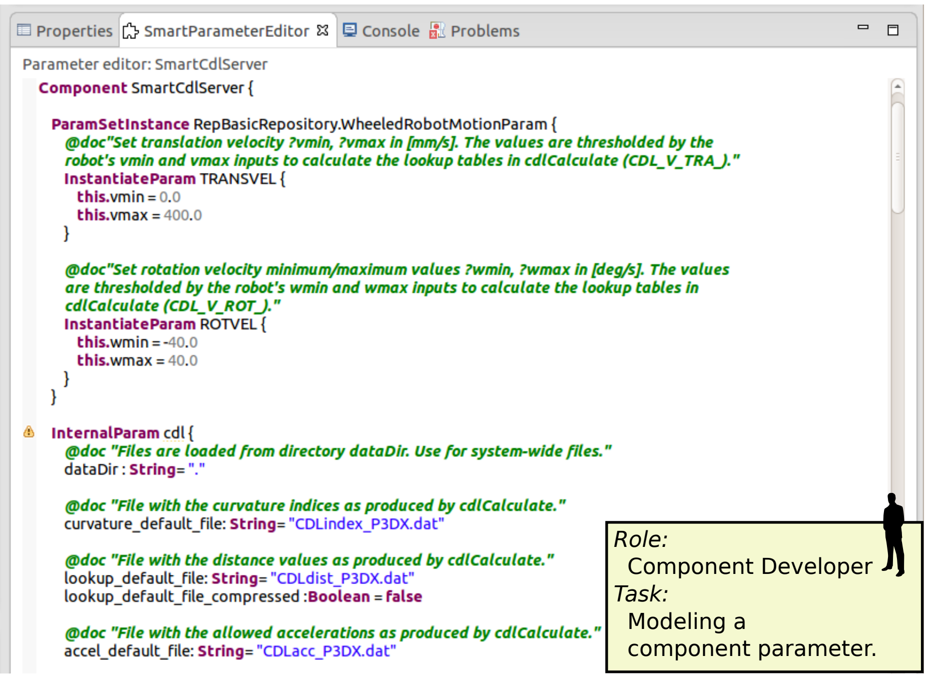


```
ComponentServices SmartPioneerBaseServer {
    Service sink PBSmotionCommandSink : RepBasicRepository.WheeledRobotMotion {
        requiredProperties :
            RepBasicRepository.RobotMotionType equals DIFFERENTIAL
    }
    // ...
```

Role:
Component Developer
Task:
Modeling a component service.

Figure 6.17: An example counterpart of the component service illustrated in Fig. 6.16: An example from SmartPioneerBaseServer.

Component Parameter Figure 6.18 illustrates the component parameter DSL while modeling the variation points for the SmartCdlServer component. By instantiating the parameter set *WheeledRobotMotionParam*, the component selects the variation points it can deal with during later system composition; it also sets their default values. In addition to reusing the parameter sets, the component may define custom component-internal parameters that are specific to the component or implementation of the SmartCdlServer. For example, the CDL implementation uses offline-generated look-up files to lower calculation effort at run-time. The component model thus defines variation points for these files and settings. Other components that also use a *WheeledRobotMotion* service source might not use these files and might need different component-specific parameters.



Parameter editor: SmartCdlServer

```
Component SmartCdlServer {

  ParamSetInstance RepBasicRepository.WheeledRobotMotionParam {
    @doc"Set translation velocity ?vmin, ?vmax in [mm/s]. The values are thresholded by the
    robot's vmin and vmax inputs to calculate the lookup tables in cdlCalculate (CDL_V_TRA_)."
    InstantiateParam TRANSVEL {
      this.vmin = 0.0
      this.vmax = 400.0
    }

    @doc"Set rotation velocity minimum/maximum values ?wmin, ?wmax in [deg/s]. The values
    are thresholded by the robot's wmin and wmax inputs to calculate the lookup tables in
    cdlCalculate (CDL_V_ROT_)."
    InstantiateParam ROTVEL {
      this.wmin = -40.0
      this.wmax = 40.0
    }
  }

  InternalParam cdl {
    @doc "Files are loaded from directory dataDir. Use for system-wide files."
    dataDir : String= "."

    @doc "File with the curvature indices as produced by cdlCalculate."
    curvature_default_file: String= "CDLindex_P3DX.dat"

    @doc "File with the distance values as produced by cdlCalculate."
    lookup_default_file: String= "CDLdist_P3DX.dat"
    lookup_default_file_compressed :Boolean = false

    @doc "File with the allowed accelerations as produced by cdlCalculate."
    accel_default_file: String= "CDLacc_P3DX.dat"
```

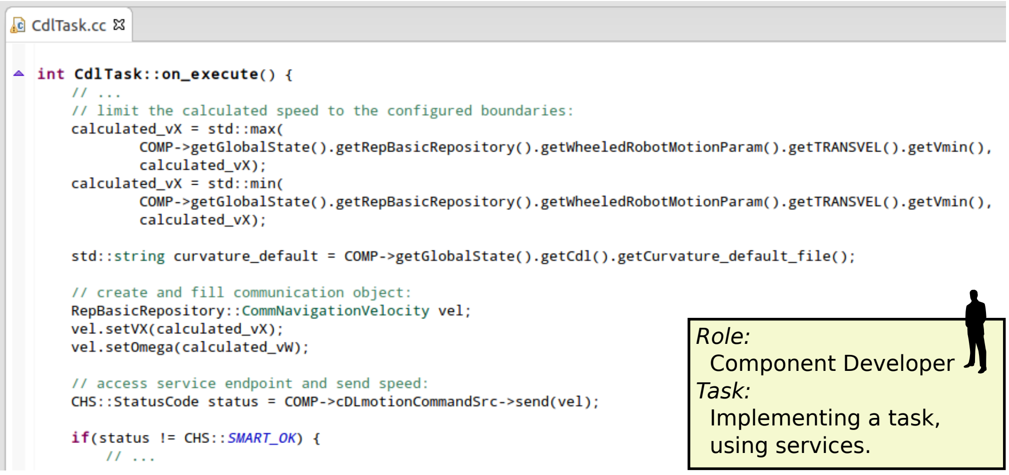
Role:
Component Developer
Task:
Modeling a
component parameter.

Figure 6.18: The component parameter DSL is used to model the component parameters of the SmartCdlServer component. The component parameter instantiates parameter sets as defined by the service designer (Fig. 6.14).

Component Implementation

From the component model, the toolchain generates skeletons to implement the business logic using the Eclipse CDT plugin for C++ development. The component developer adds own algorithms or writes glue-code for libraries.

The developer can access component parameters, communication objects and service endpoints from within the user-code (Fig. 6.19). The method in this example is executed in cycles while the SmartCdlServer component is running. In the example, a threshold is applied to the calculated translational and rotational speed of the robot as specified by the component parameters. Values of these variation points will be assigned later during system configuration. In case they are not, the default values as specified in the component model will be used. Parameters are accessed via the component infrastructure *COMP->getGlobalState()*; from there, the nested accessors (getters) follow the hierarchy of the initial parameter definition (the parameter set, section 6.3.2 or component-internal parameter, section 6.3.3). Figure 6.19 also illustrates accessing attributes of communication objects and using the service endpoint as provided by the SmartSoft framework.



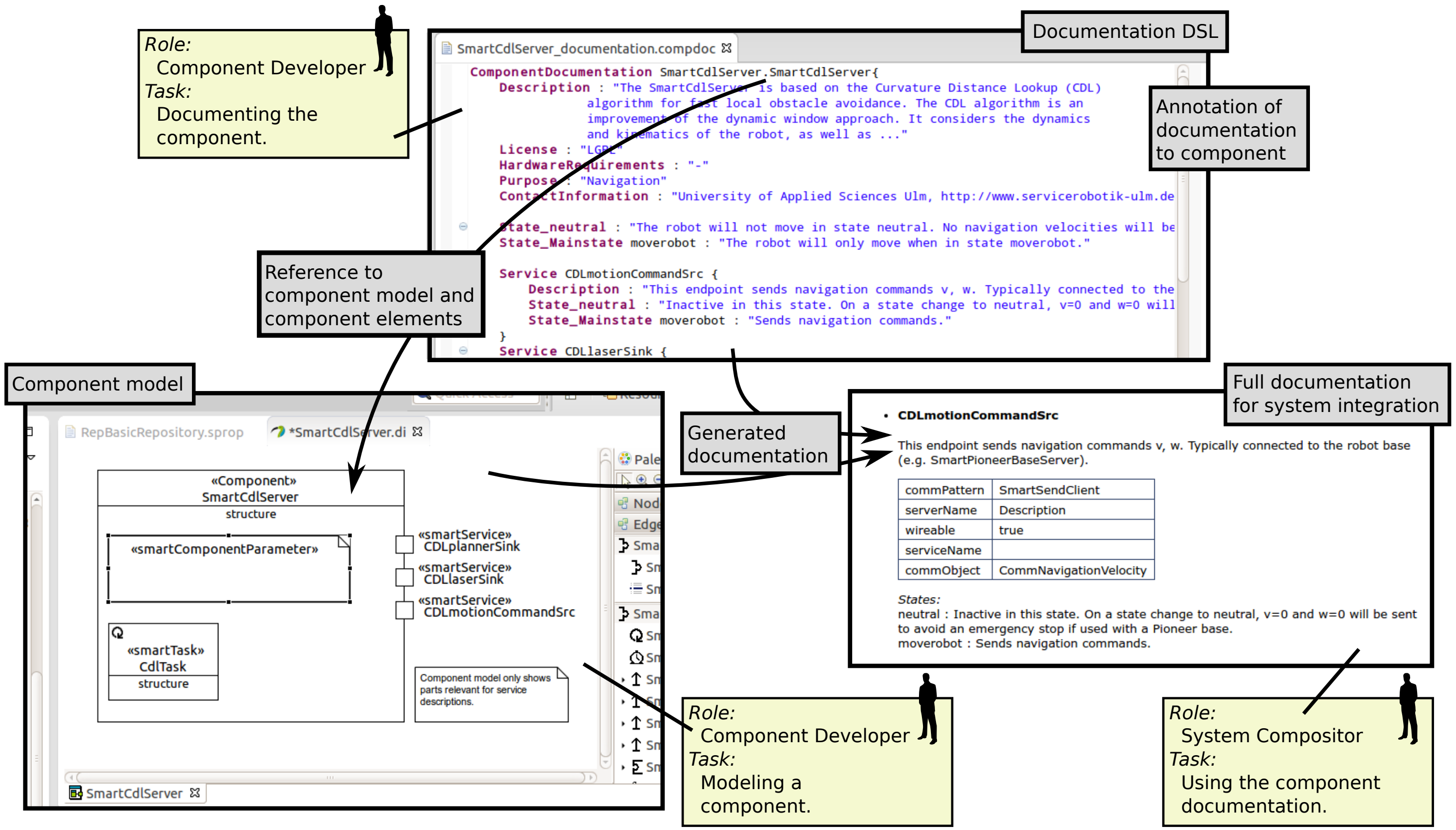
```
int CdlTask::on_execute() {
    // ...
    // limit the calculated speed to the configured boundaries:
    calculated_vX = std::max(
            COMP->getGlobalState().getRepBasicRepository().getWheeledRobotMotionParam().getTRANSVEL().getVmin(),
            calculated_vX);
    calculated_vX = std::min(
            COMP->getGlobalState().getRepBasicRepository().getWheeledRobotMotionParam().getTRANSVEL().getVmin(),
            calculated_vX);

    std::string curvature_default = COMP->getGlobalState().getCdl().getCurvature_default_file();

    // create and fill communication object:
    RepBasicRepository::CommNavigationVelocity vel;
    vel.setVX(calculated_vX);
    vel.setOmega(calculated_vW);

    // access service endpoint and send speed:
    CHS::StatusCode status = COMP->cDLmotionCommandSrc->send(vel);

    if(status != CHS::SMART_OK) {
        // ...
```

Role:
Component Developer
Task:
Implementing a task,
using services.

Figure 6.19: Implementing a component: accessing parameters, communication objects, and service endpoints from the component implementation source code.

Component Documentation

The documentation DSL (Fig. 6.20) is used to annotate elements of the graphical component model with prose text descriptions. The purpose is to provide details from the outside view on the component during the system composition step. The full component documentation is generated from the documentation model, the component model, and all models referenced from there. The documentation is thus always up-to-date.



```
ComponentDocumentation SmartCdlServer.SmartCdlServer{
    Description : "The SmartCdlServer is based on the Curvature Distance Lookup (CDL)
                  algorithm for fast local obstacle avoidance. The CDL algorithm is an
                  improvement of the dynamic window approach. It considers the dynamics
                  and kinematics of the robot, as well as ..."
    License : "LGPL"
    HardwareRequirements : "-"
    Purpose : "Navigation"
    ContactInformation : "University of Applied Sciences Ulm, http://www.servicerobotik-ulm.de

    State_neutral : "The robot will not move in state neutral. No navigation velocities will be
    State_Mainstate moverobot : "The robot will only move when in state moverobot."

    Service CDLmotionCommandSrc {
        Description : "This endpoint sends navigation commands v, w. Typically connected to the
        State_neutral : "Inactive in this state. On a state change to neutral, v=0 and w=0 will
        State_Mainstate moverobot : "Sends navigation commands."
    }
    Service CDLlaserSink {
```

- **CDLmotionCommandSrc**

This endpoint sends navigation commands v, w. Typically connected to the robot base (e.g. SmartPioneerBaseServer).

commPattern	SmartSendClient
serverName	Description
wireable	true
serviceName	
commObject	CommNavigationVelocity

States:
neutral : Inactive in this state. On a state change to neutral, v=0 and w=0 will be sent to avoid an emergency stop if used with a Pioneer base.
moverobot : Sends navigation commands.

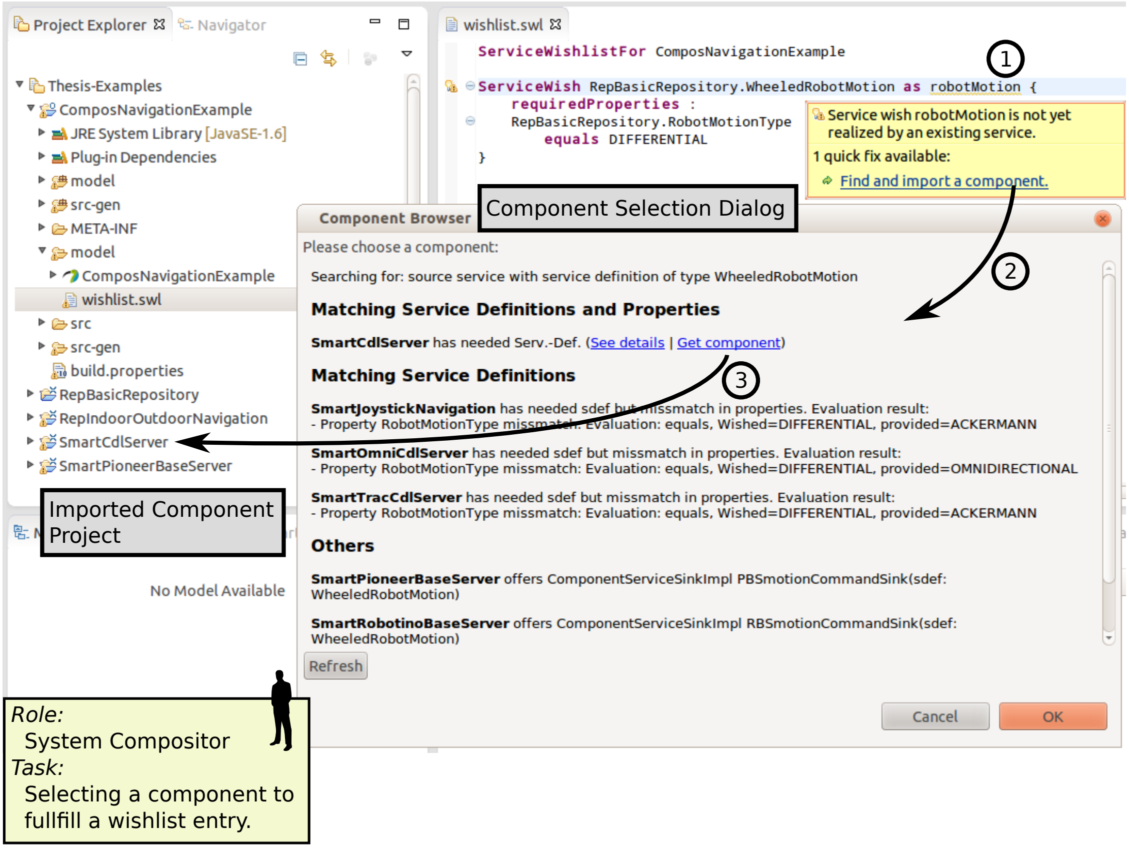
Figure 6.20: The documentation DSL is used to annotate prose text to elements of the graphical component model. The full component documentation is generated by using the documentation model but also by collecting information from all referenced models.

6.3.4 System Composition Step

System composition is the main step to put together components to applications. The system compositor uses service definitions to create the wish list model. Components are then selected and imported to the composition project based on the wish list. System configuration includes wiring and parameterizing the components. The deployment is modeled in the last step.

Wish List and Component Selection

The textual wish list model references the service definitions and instantiates properties. Figure 6.21 shows a service wish ① for *WheeledRobotMotion*. The example uses a Pioneer P3DX robot base with a differential drive. Therefore, the wish for *WheeledRobotMotion* uses service properties to express the needs for a service offering motion commands supporting a differential drive.



```
ServiceWishlistFor ComposNavigationExample

ServiceWish RepBasicRepository.WheeledRobotMotion as robotMotion {
    requiredProperties :
    RepBasicRepository.RobotMotionType
        equals DIFFERENTIAL
}
```

Service wish robotMotion is not yet realized by an existing service.

1 quick fix available:

Find and import a component.

Please choose a component:

Searching for: source service with service definition of type WheeledRobotMotion

Matching Service Definitions and Properties

SmartCdlServer has needed Serv.-Def. (See details | Get component)

Matching Service Definitions

SmartJoystickNavigation has needed sdef but missmatch in properties. Evaluation result:
- Property RobotMotionType missmatch: Evaluation: equals, Wished=DIFFERENTIAL, provided=ACKERMANN

SmartOmniCdlServer has needed sdef but missmatch in properties. Evaluation result:
- Property RobotMotionType missmatch: Evaluation: equals, Wished=DIFFERENTIAL, provided=OMNIDIRECTIONAL

SmartTracCdlServer has needed sdef but missmatch in properties. Evaluation result:
- Property RobotMotionType missmatch: Evaluation: equals, Wished=DIFFERENTIAL, provided=ACKERMANN

Others

SmartPioneerBaseServer offers ComponentServiceSinkImpl PBSmotionCommandSink(sdef: WheeledRobotMotion)

SmartRobotinoBaseServer offers ComponentServiceSinkImpl RBSmotionCommandSink(sdef: WheeledRobotMotion)

Role:
System Compositor
Task:
Selecting a component to fullfill a wishlist entry.

Figure 6.21: Component selection in the SmartMDSD Toolchain: Based on the service wish list ①, the component selection dialog ②, lists components for selection by the user. The selected component is then retrieved and imported to the workspace ③.

To this point, there is no wish realization for this service, so the user can use the component selection dialog (②, Fig. 6.21) to find a suitable component. For each component, the component repository holds the component service model of the component and an archive file (Fig. 6.22). The full Eclipse component project (including the component model and all source files) is stored in a *.zip* archive. As a separate file, the component service model holds the component services as defined in the component model (see Fig. 6.16, in the background). They are stored in a *.service* file to allow accessing them via the toolchain during component selection without downloading the full component archive. The *.service* file acts as kind of a digital data sheet and is used as input for matchmaking. The toolchain applies the matchmaking algorithm and lists the results. The toolchain also lists components with failed matches to report the reasons to the system compositor. Eventually, he may adapt the application's needs (section 5.4.2). The user then chooses one of the components to retrieve it. The toolchain then downloads the archive from the component market and automatically imports it to the toolchain workspace (Figs. 6.21 and 6.22). The component is then available for system configuration and parameterization (③, Fig. 6.21).

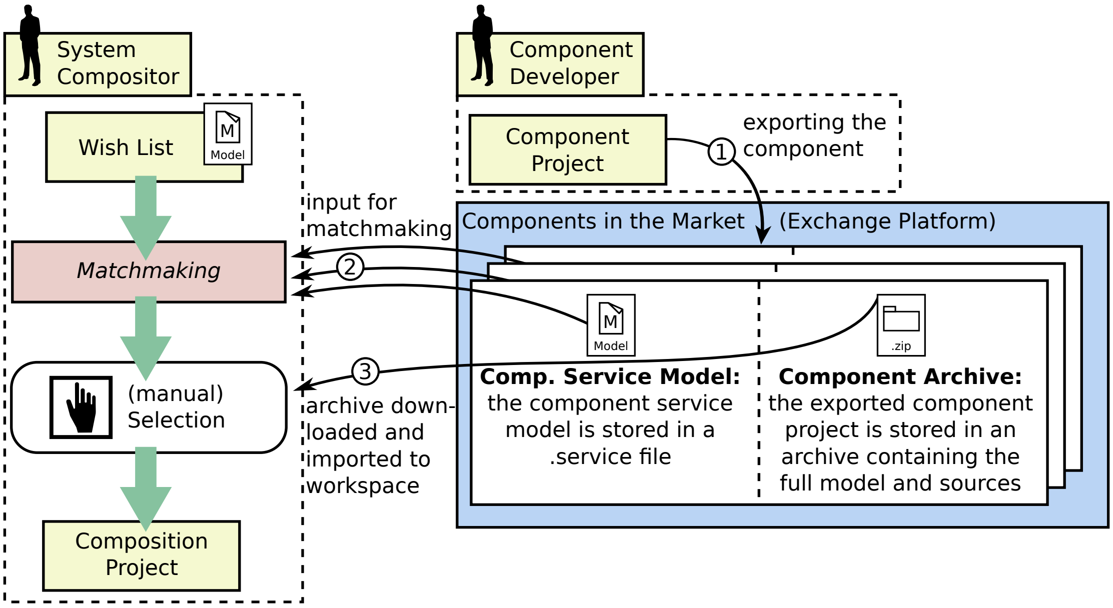

System Compositor
Wish List
M Model
Matchmaking
(manual) Selection
Composition Project
Component Developer
Component Project
① exporting the component
input for matchmaking
②
③
archive downloaded and imported to workspace
Components in the Market (Exchange Platform)
M Model
.zip
Comp. Service Model: the component service model is stored in a .service file
Component Archive: the exported component project is stored in an archive containing the full model and sources

Figure 6.22: A technical view on component retrieval. Components are exported and kept in a *.service* file representing the software artifacts that are stored in a *.zip* archive. The numbers indicate the sequence. (See also: Fig. 5.20)

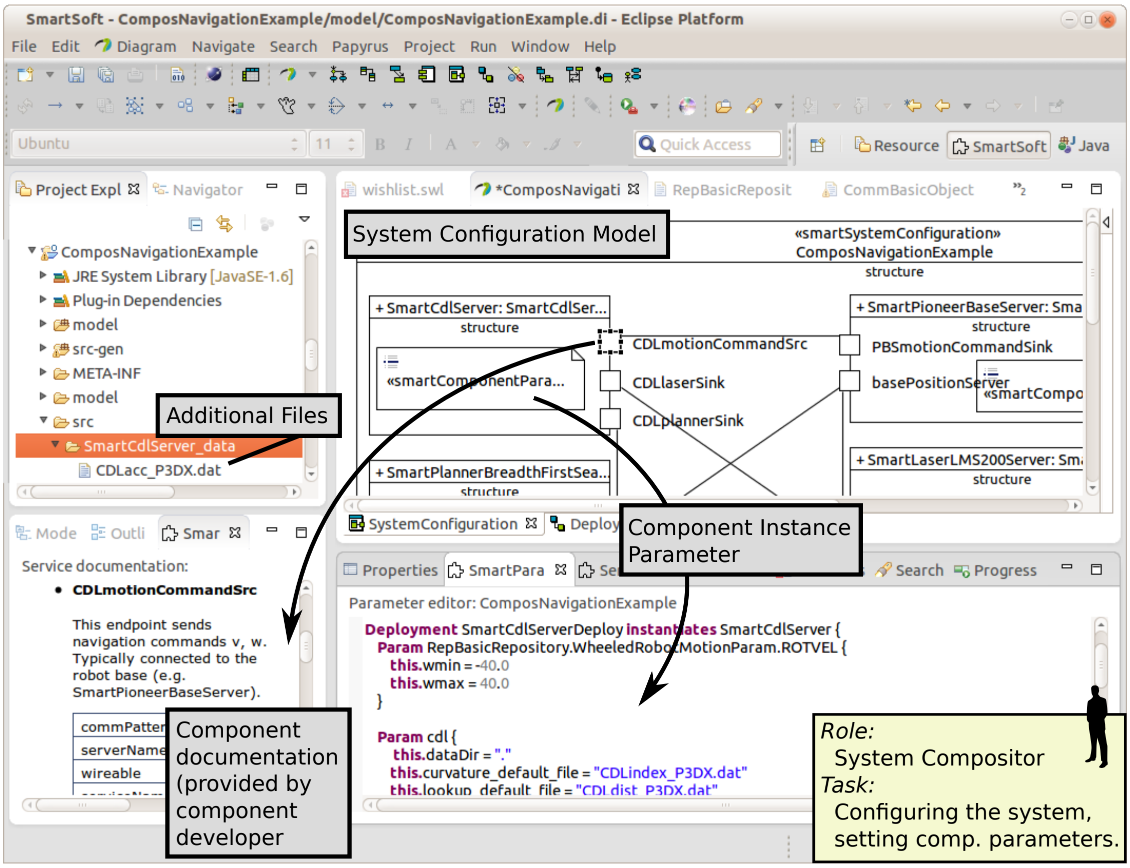
System Configuration and Parametrization

The user instantiates components and creates the initial wiring between their service endpoints in the system configuration model. The system configuration model and the according view presents the outside view of components. It shows only the component hull and the service endpoints. The user can only instantiate components and work with these instances, but cannot modify the component or component model itself. System configuration is modeled graphically using Papyrus and UML profiles (Fig. 6.23). For this purpose, a Papyrus "composite structure diagram" is used. Instances are modeled as UML Properties and are contained in a stereotyped class that represents the application (*ComposNavigationExample*, Fig. 6.23). The UML Property type refers to the component model, the property name is used to assign a name to the instance. Clicking an element of the component hull brings up its documentation (Fig. 6.23, lower left).

The textual composition parameter DSL is used to parameterize the component and to assign values to variation points as specified in the component model. The composition parameter model is stored in stereotyped UML comments. They are accessible within an editor by clicking on the comment element (Fig. 6.23). Figure 6.23 (bottom) shows an excerpt of the refinement of composition parameters. The first parameter assigns values to attributes of the parameter set *ROTVEL*. The second parameter assigns values to attributes of the component-internal parameter *cdl*.

For each instance, a unique directory is created ("<instance-name>_data") to store additional files that the component might need for execution, for example static maps or object databases. Files in the data directories will be deployed with the component instance. In the given example, the data directory holds the offline CDL look-up-table (Fig. 6.23, top left). The code-generator also creates skeletons to hook commands into the startup procedure of the application that launches all instances. Additional commands can be executed before/after starting/stopping each component instance (Fig. 6.24).

The user models service wish realizations using the wish list DSL. A wish realization links a service wish with the concrete service endpoint of a component instance. The example in Fig. 6.25 shows the wish realization of *robotMotion* that is realized by the component instance of SmartCdlServer and its service endpoint *CDLmotionCommandSrc*. The service wish's property was temporarily modified from *DIFFERENTIAL* to *ACKERMANN*, to provoke an error during verification of the composition: the error is then triggered since the SmartCdlServer no longer fulfills the wish. The toolchain supports with other checks during system composition, e.g. incompatible services, different service definitions, properties do not match, src–src or sink–sink connected services, dangling (unconnected) service endpoints, etc.



Service documentation:

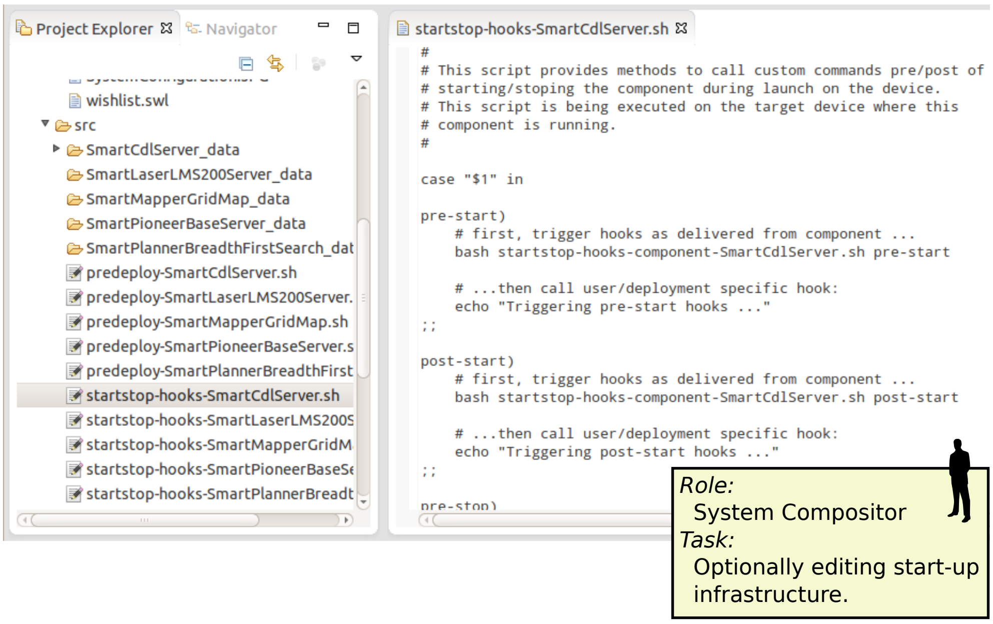
- **CDLmotionCommandSrc**

  This endpoint sends navigation commands v, w. Typically connected to the robot base (e.g. SmartPioneerBaseServer).

Parameter editor: ComposNavigationExample

```
Deployment SmartCdlServerDeploy instantiates SmartCdlServer {
  Param RepBasicRepository.WheeledRobotMotionParam.ROTVEL {
    this.wmin = -40.0
    this.wmax = 40.0
  }

  Param cdl {
    this.dataDir = "."
    this.curvature_default_file = "CDLindex_P3DX.dat"
    this.lookup_default_file = "CDLdist_P3DX.dat"
```

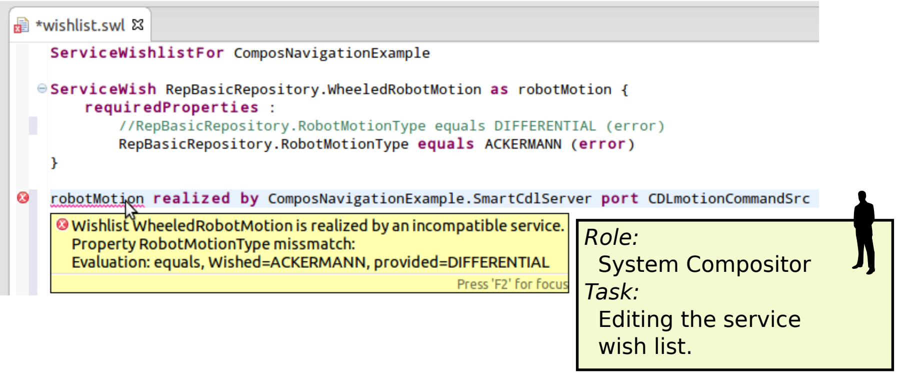
Figure 6.23: The system configuration model while editing parameters of a component instance and viewing the component documentation.


Project Explorer
Navigator
wishlist.swl
src
SmartCdlServer_data
SmartLaserLMS200Server_data
SmartMapperGridMap_data
SmartPioneerBaseServer_data
SmartPlannerBreadthFirstSearch_dat
predeploy-SmartCdlServer.sh
predeploy-SmartLaserLMS200Server.
predeploy-SmartMapperGridMap.sh
predeploy-SmartPioneerBaseServer.s
predeploy-SmartPlannerBreadthFirst
startstop-hooks-SmartCdlServer.sh
startstop-hooks-SmartLaserLMS200S
startstop-hooks-SmartMapperGridM
startstop-hooks-SmartPioneerBaseSe
startstop-hooks-SmartPlannerBreadt
startstop-hooks-SmartCdlServer.sh

```
#
# This script provides methods to call custom commands pre/post of
# starting/stoping the component during launch on the device.
# This script is being executed on the target device where this
# component is running.
#

case "$1" in

pre-start)
    # first, trigger hooks as delivered from component ...
    bash startstop-hooks-component-SmartCdlServer.sh pre-start

    # ...then call user/deployment specific hook:
    echo "Triggering pre-start hooks ..."
;;

post-start)
    # first, trigger hooks as delivered from component ...
    bash startstop-hooks-component-SmartCdlServer.sh post-start

    # ...then call user/deployment specific hook:
    echo "Triggering post-start hooks ..."
;;

pre-stop)
```

Role:
System Compositor
Task:
Optionally editing start-up infrastructure.

Figure 6.24: Editing hooks of the start-up infrastructure (optional). A generated shell script provides a skeleton to hook into the application start-up infrastructure for customization, e.g. to execute commands before/after starting/stopping instances.



```
ServiceWishlistFor ComposNavigationExample

ServiceWish RepBasicRepository.WheeledRobotMotion as robotMotion {
    requiredProperties :
        //RepBasicRepository.RobotMotionType equals DIFFERENTIAL (error)
        RepBasicRepository.RobotMotionType equals ACKERMANN (error)
}

robotMotion realized by ComposNavigationExample.SmartCdlServer port CDLmotionCommandSrc
```

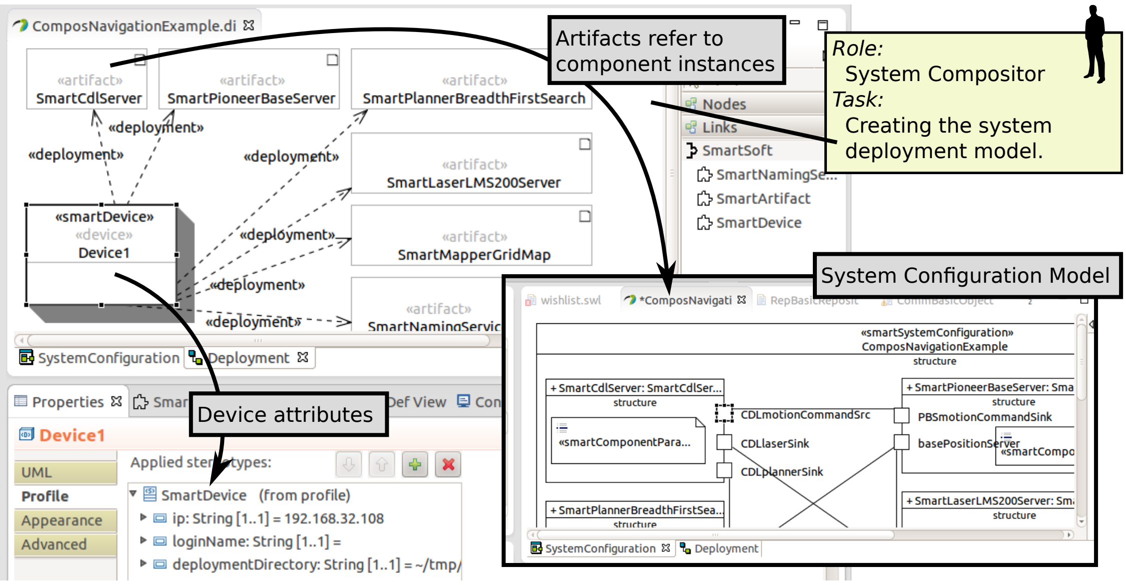
Wishlist WheeledRobotMotion is realized by an incompatible service.
Property RobotMotionType missmatch:
Evaluation: equals, Wished=ACKERMANN, provided=DIFFERENTIAL

Role:
System Compositor
Task:
Editing the service wish list.

Figure 6.25: The realization of a service wish using the wish list DSL. The service's wish was temporarily modified to provoke an error during verification.

Deployment

The deployment model maps component instances to execution units to prepare transfer and execution of the application. It is modeled graphically using a Papyrus UML "deployment diagram". Devices map to stereotyped UML Devices with additional attributes to cover the elements of the deployment meta-model. Stereotyped UML Artifacts are used as helper-elements to represent and refer to component instances to link them to devices using UML Deployment connectors (Fig. 6.26). During deployment modeling, the user can only use instances but cannot further modify them. In the given example, all instances that are created in the system configuration model map to a single device in the deployment model (Fig. 6.26).



Artifacts refer to component instances

Role: System Compositor
Task: Creating the system deployment model.

System Configuration Model

Device attributes

Applied stereotypes:
- SmartDevice (from profile)
  - ip: String [1..1] = 192.168.32.108
  - loginName: String [1..1] =
  - deploymentDirectory: String [1..1] = ~/tmp/

ComposNavigationExample.di
«artifact» SmartCdlServer
«artifact» SmartPioneerBaseServer
«artifact» SmartPlannerBreadthFirstSearch
«artifact» SmartLaserLMS200Server
«artifact» SmartMapperGridMap
«artifact» SmartNamingServic
«deployment»
«smartDevice» «device» Device1
Nodes
Links
SmartSoft
SmartNamingSe...
SmartArtifact
SmartDevice
SystemConfiguration
Deployment
Properties
Smar
Def View
Con
Device1
UML
Profile
Appearance
Advanced
wishlist.swl
*ComposNavigati
RepBasicReposit
CommBasicObject
«smartSystemConfiguration» ComposNavigationExample
structure
+ SmartCdlServer: SmartCdlSer...
«smartComponentPara...
CDLmotionCommandSrc
CDLlaserSink
CDLplannerSink
+ SmartPioneerBaseServer: Sma
PBSmotionCommandSink
basePositionServer
«smartCompo
+ SmartPlannerBreadthFirstSea...
+ SmartLaserLMS200Server: Sm

Figure 6.26: The deployment is modeled using Papyrus and maps component instances to execution units to prepare transfer and execution of the application.

Transfer to the Robot and Starting the Application

Creating the deployment model is the last step in the workflow. Based on the deployment model and the system configuration model, the toolchain collects all artifacts and transfers them to the target devices. The application is then ready to start.

6.4 Summary

This chapter discussed considerations and alternatives to implement the meta-structure for system composition in tooling. It has described one possible mapping of the composition structures to tooling. It has presented the concrete implementation as a contribution to the Eclipse-based SmartMDSD Toolchain, an Integrated Development Environment (IDE) for robotics software. The SmartMDSD Toolchain supports ecosystem participants in applying the composition workflow. It can be used by domain experts on composition Tier 2 to create domain structures. It can be used by participants of the ecosystem on Tier 3 to provide and use actual content in the ecosystem—that is, software components that are developed and provided by component developers and systems that are composed from these components by system compositors. The chapter has described the full user perspective of the composition workflow via the SmartMDSD Toolchain, starting from a service definition model (domain experts, Tier 2), moving on to the development of components (Tier 3), and completing with the system composition and finally the deployment of the system to the robot (Tier 3).

The chapter described a use-case to present the look and feel from an ecosystem participant's point of view. The next chapter evaluates the approach to system composition and the SmartMDSD Toolchain using real-world robotics systems that were built in several activities.

7 Demos, Application, and Evaluation

This chapter presents concrete robot systems ("demonstrators") and the initiatives that have built them. It describes the benefit of using the contributions of the thesis to the SmartMDSD Toolchain for systematic engineering of software based on system composition. A user study was conducted to access the benefit and experience of the SmartMDSD Toolchain from a user's point of view. The chapter closes with a discussion to reflect the presented approach in relation to the overall research goal and questions.

7.1 Application Fields

When considering all demonstrators and projects that have applied the approach in this thesis, the field of application covers a broad spectrum as illustrated in Fig. 7.1. It ranges from custom developments to establishing a project-internal ecosystem, from users with no robotics expertise to highly skilled robotics experts, from systems in simulation to real-world state of the art systems, and even from service robotics to the smartphone domain. Section 7.3 will elaborate the demonstrators and associated projects in detail. Section 7.5 will evaluate them in the context of a conducted user study.

In all the demonstrators and in all the activities described in this chapter, the SmartMDSD Toolchain was used to develop building blocks and to compose them to applications.

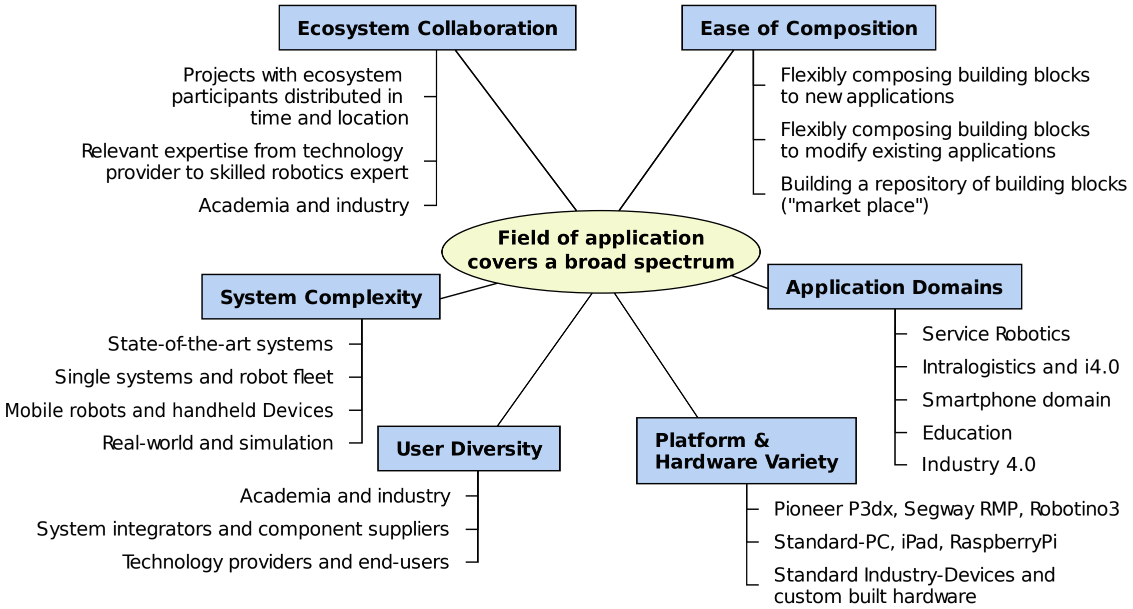

Field of application covers a broad spectrum
Ecosystem Collaboration
Projects with ecosystem participants distributed in time and location
Relevant expertise from technology provider to skilled robotics expert
Academia and industry
Ease of Composition
Flexibly composing building blocks to new applications
Flexibly composing building blocks to modify existing applications
Building a repository of building blocks ("market place")
System Complexity
State-of-the-art systems
Single systems and robot fleet
Mobile robots and handheld Devices
Real-world and simulation
Application Domains
Service Robotics
Intralogistics and i4.0
Smartphone domain
Education
Industry 4.0
User Diversity
Academia and industry
System integrators and component suppliers
Technology providers and end-users
Platform & Hardware Variety
Pioneer P3dx, Segway RMP, Robotino3
Standard-PC, iPad, RaspberryPi
Standard Industry-Devices and custom built hardware

Figure 7.1: The fields of application of the SmartMDSD Toolchain and the approach cover a broad spectrum.

7.2 Maturity of the SmartMDSD Toolchain

The SmartMDSD Toolchain has been made available in 20 public releases under an open source license comprising three major generations in the last six years. About 64 software components developed with the SmartMDSD Toolchain are available for immediate composition. About 36 components are publicly available [Ser]. The toolchain was used in several projects and collaborations and thus "has been demonstrated in operational environments", which corresponds to technology readiness level (TRL) 6 according to [euR16] (acknowledged in [Fio15]).The overall implementation of the publicly available SmartMDSD Toolchain is a joint effort of the Service Robotics Research Center at the Ulm University of Applied Sciences (Hochschule Ulm). The underlying structures and toolchain architecture of the "SmartMDSD Toolchain v2" were contributed by this thesis. They are described in the next sections.

The service definitions, service properties and wish lists are so far available in non-public demo releases only, they are planned to be publicly released with the next generation of the SmartMDSD Toolchain "v3". These parts were not yet in productive use by the presented projects, since these were in critical or final phases when demo releases were available. The demo releases were not taken up by the projects for productive use to not interfere with the existing models and implementations for stability and timely reasons. All other parts that are described in this thesis are included in the public releases and were used as described in this chapter.

Even though service definitions, service properties and wish lists were not part of the productive releases used in the projects, their concept still was applied. Especially within the FIONA research project (section 7.3.2). FIONA applied the exact workflow with service definitions as presented in this thesis. However, the service definitions were only partially modeled in the

SmartMDSD Toolchain. The communication objects were modeled in the toolchain, but the overall service definitions were "modeled" in documents. The roles followed the workflow as if the service definitions were modeled in the toolchain: The roles directly referenced communication objects and communication patterns from their components, thus adhering to service properties by policy as documented.

Applying the concept of service definitions in the project even without the underlying tool support brought immediate benefits when discussing service definitions and writing them down. Service definitions structured the project collaboration and enabled the variety of demonstrators and provided a real testbed for system composition, separation of roles and establishing a project-internal ecosystem. This, however, also underlined the need to have them available in models and tooling for consistency, automation, etc. The "modeling" of service definitions in documents is now no longer necessary as it is covered by tooling as presented in chapter 6.

7.3 Demonstrators and Activities

This section describes demonstrations of robotic systems and the activities that developed them by applying the SmartMDSD Toolchain and the approach in this thesis. Together, the demonstrators and activities cover a very broad range of applications as illustrated in Fig. 7.1.

7.3.1 The (Collaborative) Robot Butlers

The *Robot Butler Scenario* and the *Collaborative Butler Scenario* with the robots Kate and Larry (Fig. 7.2) are an outcome of the research project *ZAFH Servicerobotik (Collaborative Center for Applied Research on Service Robotics)* [ZAFH]. The ZAFH Servicerobotik developed methodologies for building service robots that act in everyday life environments. Its major approach was to extend and to merge separated technologies while aiming for suitability for use in everyday life environments.

The ZAFH Servicerobotik has shown how the structures that this thesis formalizes as meta-models enhance the development of a service robot. It has shown the positive effect of separation of roles and freedom from choice to reducing system complexity in a—at that time—state of the art service robot application. Especially the components for (active) object recognition show the flexible composition of component alternatives based on the special needs of the application for object recognition. The project has shown how to kickstart a collection of software components and the SmartSoft ecosystem. Many components are still in use today to compose new applications, e.g. the intralogistics scenario (section 7.3.3).

The ZAFH Servicerobotik consisted of three partners from academia. All were working at different locations and were acting as component suppliers. One partner additionally took the role of a system builder to build Kate and Larry. The project is an example for straight-forward software development with the aim to build the robot Kate.

The robots and components were developed with the first version of the SmartMDSD Toolchain. This first version by Steck, **Stampfer**, and Schlegel [SSS09a] established the foundation for the toolchain series and evaluated the structures that can be found in today's productive

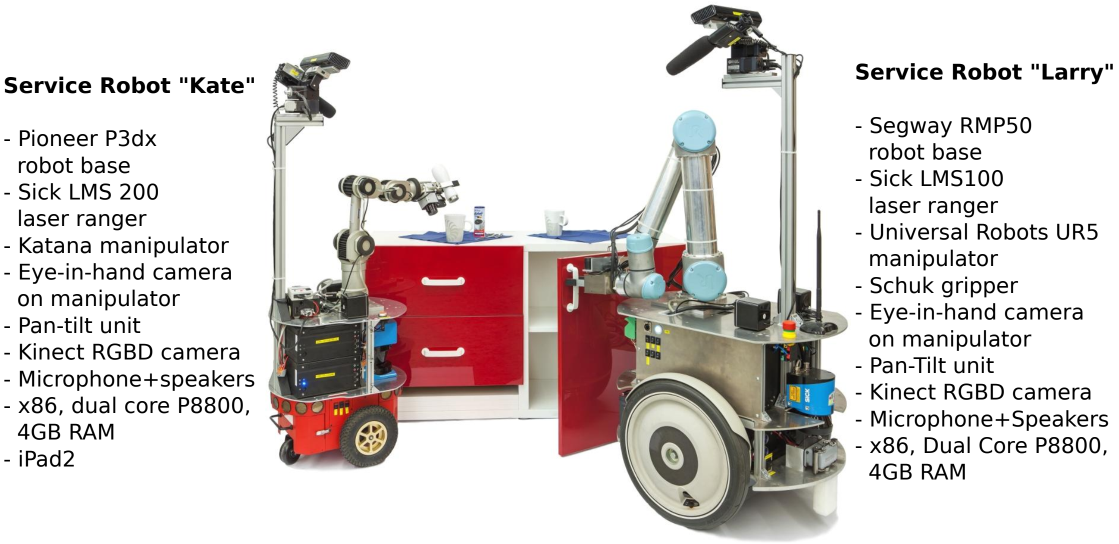

Service Robot "Kate"
- Pioneer P3dx robot base
- Sick LMS 200 laser ranger
- Katana manipulator
- Eye-in-hand camera on manipulator
- Pan-tilt unit
- Kinect RGBD camera
- Microphone+speakers
- x86, dual core P8800, 4GB RAM
- iPad2
Service Robot "Larry"
- Segway RMP50 robot base
- Sick LMS100 laser ranger
- Universal Robots UR5 manipulator
- Schuk gripper
- Eye-in-hand camera on manipulator
- Pan-Tilt unit
- Kinect RGBD camera
- Microphone+Speakers
- x86, Dual Core P8800, 4GB RAM

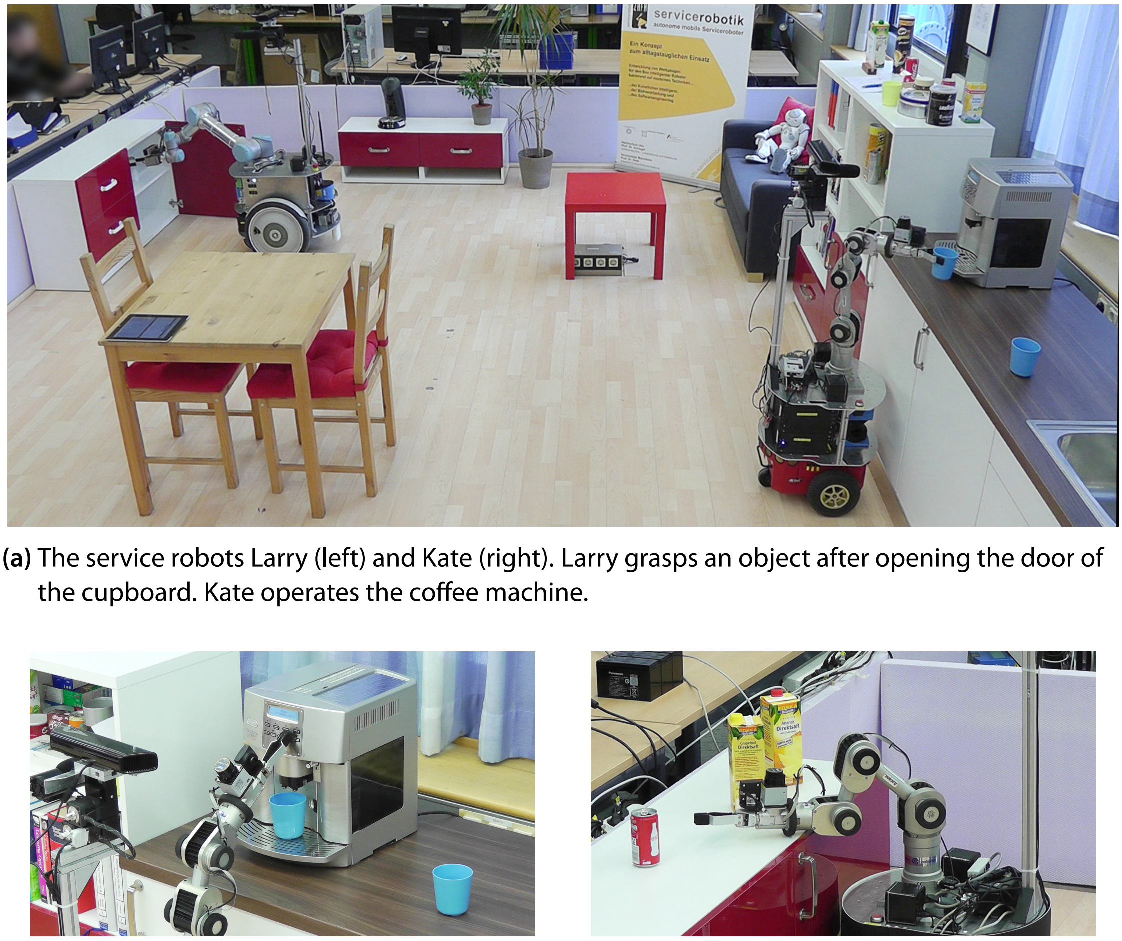
Figure 7.2: The service robots Kate (left) and Larry (right) operating in an everyday life environment.

release ("v2", see section 7.2). Parts of the structures presented in this thesis were only applied by policy as documented and "modeled on paper". During building the robots Kate and Larry, many requirements and best-practices for building a state of the art complex robot system were identified. These are now formalized in the composition structures and supported with tooling by this thesis.

The *Robot Butler Scenario* is a 30-minute scenario in which the service robot "Kate" works as Butler in a household-like environment (Fig. 7.3a, video at [Heg+12]). Users can order drinks through spoken language that the robot then fetches. Kate prepares coffee by fetching cups and operating an ordinary coffee machine (Fig. 7.3b). The service robot Kate runs a total of 21 components on two computers (Table 7.1). A second robot "Larry" was later introduced to share work with Kate in the *Collaborative Butler Scenario* (Fig. 7.3a, video at [Lut+13]). Larry is equipped with a more capable manipulator to perform more complex manipulation tasks such as opening a cupboard. An overview on the components that run onboard of Kate and Larry is given in Fig. 7.3b.

Larry was put into operation by composing the components from the ecosystem that was kick-started with the development of Kate. Larry is one example of modifying an existing system to match new needs with low effort: The component architecture of Kate was transferred to Larry by exchanging three components. These three components were selected from "component alternatives" to match Larry's hardware (robot base, laser ranger, and manipulator). Larry's full system configuration diagram is shown in Fig. 7.4 (see Schlegel, Lotz, Lutz, et al. [Sch+15]).

The butler and collaborative robot butler scenarios feature algorithms that are suitable for use in everyday life environments ("Alltagstauglichkeit"). The robots featured the integration of a whole range of capabilities that were state of the art: mobile manipulation (fetching cups, pressing buttons of the coffee machine), object recognition (recognizing cups, juices, coffee machine), person recognition (to deliver orders correctly), speech recognition, task sequencing, localization, path planning, obstacle avoidance, etc. A system that integrates such a variety of capabilities can only be developed and maintained with an appropriate development methodology and supporting tool that supports in managing system complexity. The SmartMDSD Toolchain and its structures proved to do so, even at this early stage in 2012/2013.



(a) The service robots Larry (left) and Kate (right). Larry grasps an object after opening the door of the cupboard. Kate operates the coffee machine.

(b) Kate operating the coffee machine.

(c) Active object recognition.

Figure 7.3: Excerpts of the (collaborative) butler scenario in a household-environment. Video at [Lut+13]

Component	Purpose	Description	Kate	Larry
SmartTCL	Task Coordination	Task sequencer using SmartTCL	x	x
SmartSymbolicPlanner	Task Coordination	Planning object handling and stacking using metric-FF	x	x
SmartTTSLoquendeo	HMI	Speech synthesis using Loquendo library	x	x
SmartSTTLoquendo	HMI	Speech recognition using Loquendo library	x	x
SmartFaceRecognition	HMI	Face recognition based on Verilook library	x	x
SmartWebInterface	HMI	Interface to tablet PC for the user	x	x
SmartGMapping	Localization	SLAM	x	x
SmartAmcl	Localization	Localization in a map	x	x
SmartSchunkGripperServer	Mobile Manipulation	Operating the gripper		x
SmartOpenRave	Mobile Manipulation	Manipulation planning using OpenRave	x	x
SmartURServer	Mobile Manipulation	Universal Robots Manipulator		x
SmartKatanaServer	Mobile Manipulation	Katana Manipulator	x	
SmartCdlServer	Navigation	Collision Avoidance	x	x
SmartLaserLMS1xxServer	Navigation	Laser Ranger		x x
SmartLaserLMS200Server	Navigation	Laser Ranger	x	
SmartRMPBaseServer	Navigation	Base platform of the robot		x
SmartPioneerServer	Navigation	Base platform of the robot	x	
SmartPlannerBreadthFirstSearch	Navigation	Path planning for navigation	x	x
SmartMapperGridMap	Navigation	Obstacle grid map for navigation	x	x
SmartActiveObjectRecognition	Object Recognition	Very reliable object recognition with object inspection	x	x
SmartIDSuEyeImageServer	Object Recognition	Camera on manipulator to inspect objects	x	x
SmartLaserSimpleObjectDetector	Object Recognition	Simple detection of objects on the floor	x	x
SmartObjectRecognition	Object Recognition	Object recognition utilizing a bunch of algorithms and OpenCV, MRPT, ABBY OCR, Tesseract OCR, ZBAR, PCL, MOPED	x	x
SmartKinectServer	Object Recognition	Color-Image and depth camera	x	x
SmartPTUServer	Object Recognition	Pan-Tilt-Unit for camera	x	x

Table 7.1: The 25 software components that are composed in Kate (21) and Larry (23). A "x" for each component instance.

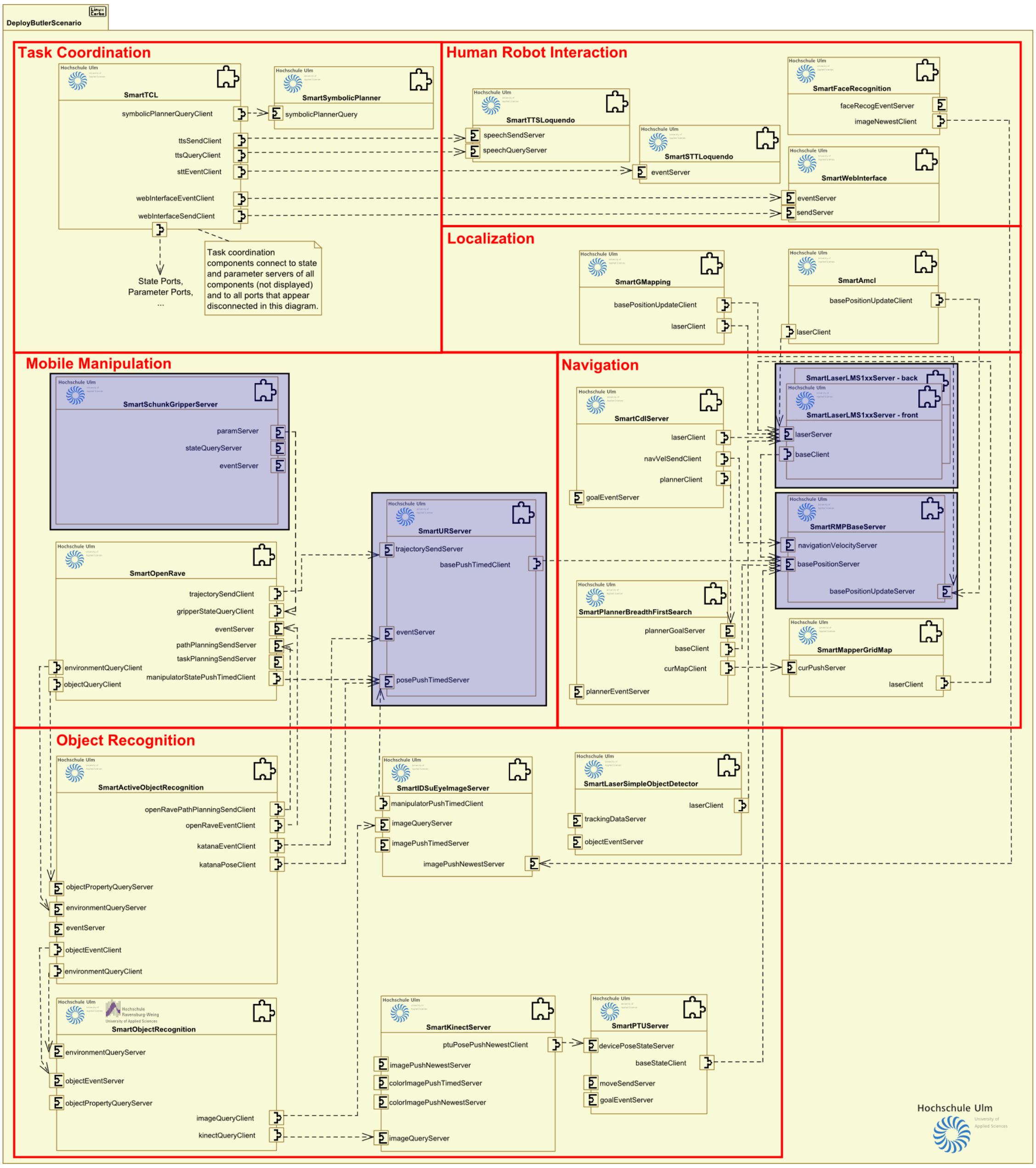


Figure 7.4: The system configuration view of robot "Larry" from the collaborative butler scenario (figure from [Sch+13]). While composing the robot Larry, most of the component architecture stays the same: Only the components in blue boxes are different from Kate. All others have been composed as-is. This is a screenshot of the SmartMDSD Toolchain. The red boxes, the blue boxes, and the logos have been added for clarification.

The scenarios were iteratively developed, modified, and enhanced by adding and removing components. One example is the object recognition that demonstrates the composability of components and also the selection of component alternatives based on the needs of the application (Fig. 7.5). An object recognition service may be provided by different components that each implement special algorithms for specific objects (e.g. color-based, feature-based, shape-based) as shown in Fig. 7.5 ("traditional way"). The system builder would choose the component based on the application's needs. A second variant is one powerful component that uses several algorithms to probabilistically fuse their results for more reliability (Fig. 7.5, "multimodal object recognition"). This has been demonstrated as *multimodal object recognition* in [Lut+12; LSS13]. The reliability of object recognition can even be improved in a third variant with an active approach as demonstrated in [SLS12a; SLS12b] (video at [LS12]; Fig. 7.5, bottom). *Active object recognition* is an example for the flexible composition of components for the dedicated purpose of object recognition. It is an example for separation of roles and dedicated expertise: The object recognition expert can focus on his expertise while the manipulation components were composed from existing ones.

In active object recognition, an object is not recognized based on an image from a single view alone. The object is actively inspected from different viewpoints. The approach uses an "eye-in-hand camera" that is mounted on the manipulator (Fig. 7.3c, bottom). Based on an initial recognition and initial guess about the object, the robot calculates the next-best-view, places the camera and recognizes the object again. The result of the new observation is probabilistically fused and the process is repeated until the necessary reliability is reached. The approach was demonstrated to reliably recognize even almost identical objects and is suitable for tasks that require high reliability, such as recognizing medicine. Giving this set of components for object recognition, the service definitions make them composable. The service properties enable the selection of the one that matches the needs of the application as expressed by the system builder.

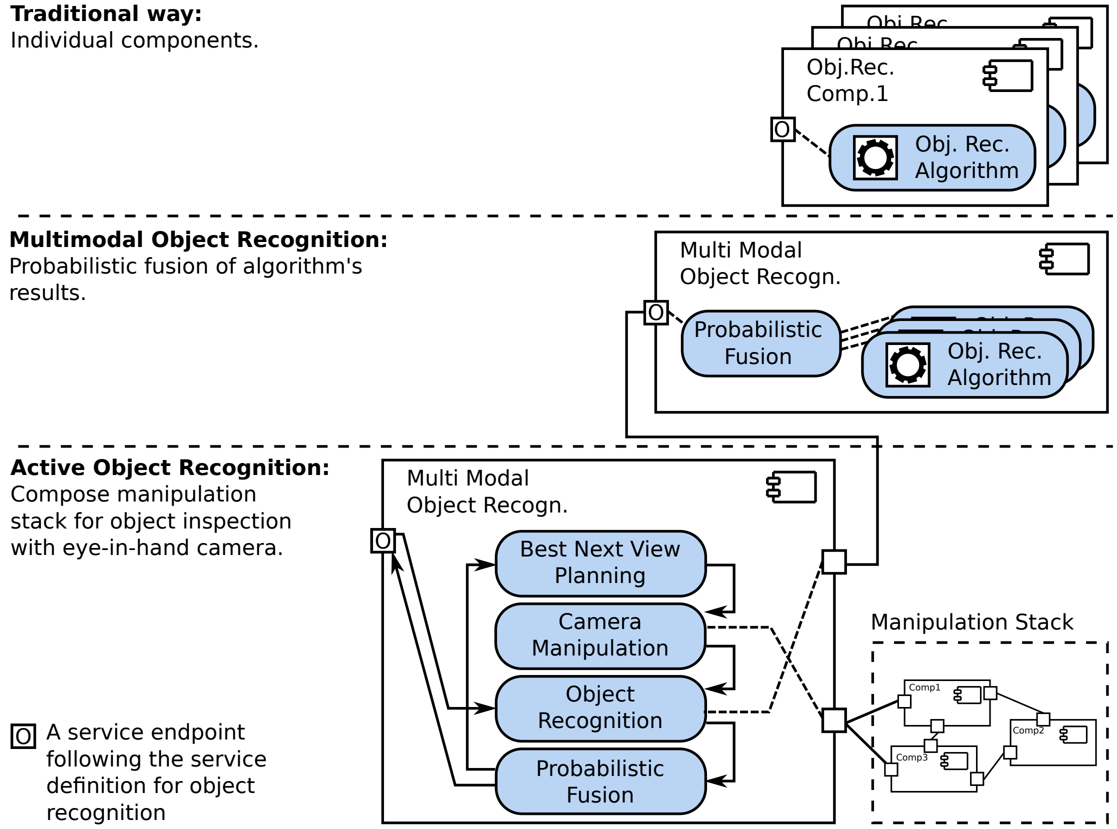

Traditional way:
Individual components.
Obj.Rec.
Obj.Rec.
Obj.Rec. Comp.1
O
Obj. Rec. Algorithm
Multimodal Object Recognition:
Probabilistic fusion of algorithm's results.
Multi Modal Object Recogn.
O
Probabilistic Fusion
Obj. Rec. Algorithm
Active Object Recognition:
Compose manipulation stack for object inspection with eye-in-hand camera.
Multi Modal Object Recogn.
O
Best Next View Planning
Camera Manipulation
Object Recognition
Probabilistic Fusion
Manipulation Stack
Comp1
Comp2
Comp3
O A service endpoint following the service definition for object recognition

Figure 7.5: Flexible combination of object recognition by composition. A variant of multimodal object recognition might only include the fusion in a dedicated component and get the results from the individual components (components as in "traditional way"). Multimodal object recognition was demonstrated in [Lut+12; LSS13], active object recognition was demonstrated in [SLS12a; SLS12b].

7.3.2 Personal Mobile Navigation

The goal of the research project *FIONA (Framework for Indoor and Outdoor Navigation Assistance)* [Fiona; Bür+16] was to establish methods and domain-specific structures that support the development of services for indoor and outdoor localization and navigation for sighted and visually impaired people. FIONA aimed at supporting the development of new applications based on these services. FIONA is an example of a consortium that developed domain-specific structures at composition Tier 2. The domain of FIONA is not the service robotics domain and therefore shows the suitability and benefit of the approach even beyond service robotics.

The FIONA project can be considered a perfect test-bed for collaboration in an ecosystem. The project consortium consisted of ten partners: Eight partners from industry versus two partners from academia. FIONA was an activity of high interest by industry looking for new approaches to system development. FIONA is a perfect example for a project with participants distributed in time and location. The partners were distributed across five countries in Europe and even new partners joined the project during its run-time. The partners covered relevant expertise from technology providers to skilled experts in the domains of smartphones and the domain of visually impaired people. The project also included end-users for evaluation.

The FIONA project impressively shows the flexibility and ease of system composition. It established a project-internal ecosystem of 29 components that were used during the project run-time to flexibly compose demonstrators or flexibly modify existing ones through composition (see [Bür+16] and Table 7.2). The project composed components to 18 demonstrators in total. The project was fully based on the composition workflow and was supported by the SmartMDSD Toolchain as presented in this thesis. Section 7.5.3 will get back to composition in FIONA when evaluating the approach and provide more details on the number of components, demonstrator variants, and effort in composing them.

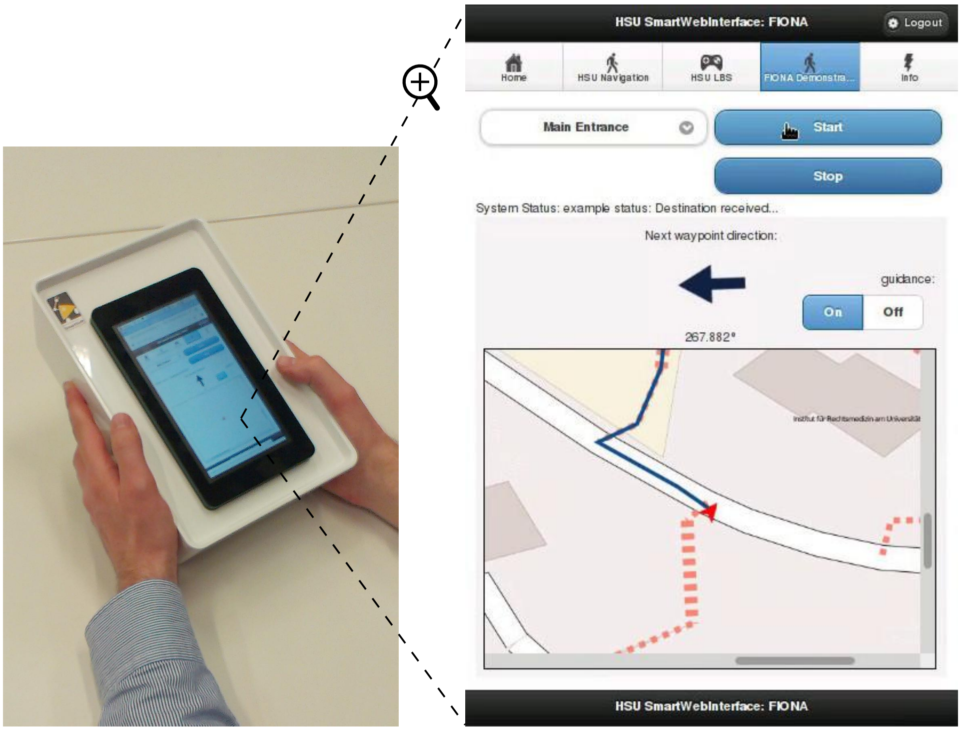
The FIONA demonstrators also illustrate a broad variety of hardware and platforms used. It includes standard x86 laptops but components were also deployed to iPad and RaspberryPi systems. FIONA used off-the-shelf commercial products (e.g. cameras, iBeacons) but also custom developments and industry prototypes (e.g. inertial measurement unit (IMU) prototypes). The demonstrators consisted of smartphones, handheld devices, and body-mounted devices.

Several mobile navigation systems were built to demonstrate the flexible composition of software components to a variety of applications. All demonstrators guide walking persons to a desired destination. The use-case is like a navigation system as known from cars or Google Maps, but focuses on pedestrian navigation based on open structures on Tier 2 (no vendor lock-in) for sighted and visually impaired people.

The most advanced demonstrator in this collection is the *"Personal Mobile Navigation Demonstrator"* (Fig. 7.6, video at [SF16]). It runs on a handheld Raspberry Pi with a touch screen. The demonstrator shows the seamless transition between indoor and outdoor navigation. This is an example for flexible composition of two different components (indoor and outdoor localization) for the same purpose (localization) that adds a benefit over using only one of them. Indoor localization uses iBeacons. Outdoor navigation is based on GPS. The demonstrator was set up to work in the closer area of Ulm University of Applied Sciences based on map data from Open Street Map (OSM). The OSM-components and the handover between indoor and outdoor was contributed by a master's thesis [Fra15]. This is an example of composable components from a technology provider. The master's thesis acted as a technology provider / component supplier. It is also an example of how separation of roles supports the collaboration of different levels of expertise. Components were contributed by a student with no initial expertise in localization, but she was able to compose a full demonstrator using also other existing components. Figure 7.7 shows the system configuration view during composition of this demonstrator. The demonstrator was tested in navigating from the train station to the Ulm University of Applied Sciences. The path was approximately 600m in length. A video can be found at [SF16].

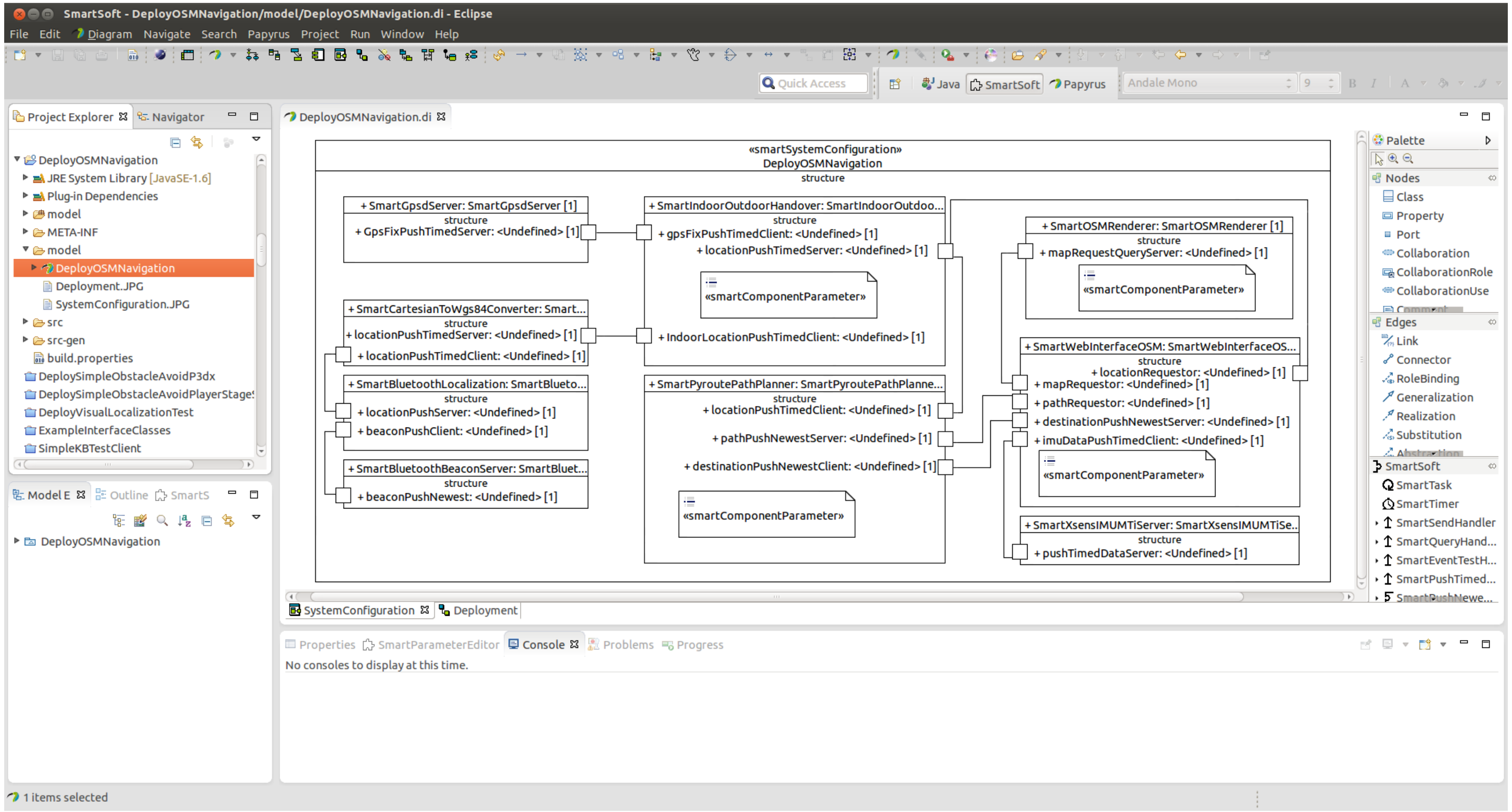
Three earlier variants of the demonstrator have been developed (in reverse order of development). They all demonstrate the composition of different components to come up with a variety of demonstrators at low effort (more variants of the personal mobile navigation demonstrator are summarized in Table 7.2).

- Seamless Indoor and Outdoor Navigation (using GPS for outdoor-localization) [FS15]
- Indoor Mobile Navigation (iBeacons for indoor localization) [Sta+14a]
- Simulator-based Mobile Navigation (using the robotics simulator "MORSE") [Sta+14b]



System Status: example status: Destination received...

Figure 7.6: The Personal Mobile Navigation Demonstrator: the handheld device (left) and its graphical user interface (right). Video at [SF16].


SmartSoft - DeployOSMNavigation/model/DeployOSMNavigation.di - Eclipse
File Edit Diagram Navigate Search Papyrus Project Run Window Help
Quick Access
Java
SmartSoft
Papyrus
Andale Mono
Project Explorer
Navigator
DeployOSMNavigation
JRE System Library [JavaSE-1.6]
Plug-in Dependencies
model
META-INF
model
DeployOSMNavigation
Deployment.JPG
SystemConfiguration.JPG
src
src-gen
build.properties
DeploySimpleObstacleAvoidP3dx
DeploySimpleObstacleAvoidPlayerStage
DeployVisualLocalizationTest
ExampleInterfaceClasses
SimpleKBTestClient
Model E
Outline
SmartS
DeployOSMNavigation
DeployOSMNavigation.di
«smartSystemConfiguration»
DeployOSMNavigation
structure
+ SmartGpsdServer: SmartGpsdServer [1]
structure
+ GpsFixPushTimedServer: <Undefined> [1]
+ SmartIndoorOutdoorHandover: SmartIndoorOutdoo...
structure
+ gpsFixPushTimedClient: <Undefined> [1]
+ locationPushTimedServer: <Undefined> [1]
«smartComponentParameter»
+ IndoorLocationPushTimedClient: <Undefined> [1]
+ SmartOSMRenderer: SmartOSMRenderer [1]
structure
+ mapRequestQueryServer: <Undefined> [1]
«smartComponentParameter»
+ SmartCartesianToWgs84Converter: Smart...
structure
+ locationPushTimedServer: <Undefined> [1]
+ locationPushTimedClient: <Undefined> [1]
+ SmartBluetoothLocalization: SmartBlueto...
structure
+ locationPushServer: <Undefined> [1]
+ beaconPushClient: <Undefined> [1]
+ SmartBluetoothBeaconServer: SmartBluet...
structure
+ beaconPushNewest: <Undefined> [1]
+ SmartPyroutePathPlanner: SmartPyroutePathPlanne...
structure
+ locationPushTimedClient: <Undefined> [1]
+ pathPushNewestServer: <Undefined> [1]
+ destinationPushNewestClient: <Undefined> [1]
«smartComponentParameter»
+ SmartWebInterfaceOSM: SmartWebInterfaceOS...
structure
+ locationRequestor: <Undefined> [1]
+ mapRequestor: <Undefined> [1]
+ pathRequestor: <Undefined> [1]
+ destinationPushNewestServer: <Undefined> [1]
+ imuDataPushTimedClient: <Undefined> [1]
«smartComponentParameter»
+ SmartXsensIMUMTiServer: SmartXsensIMUMTiSe...
structure
+ pushTimedDataServer: <Undefined> [1]
Palette
Nodes
Class
Property
Port
Collaboration
CollaborationRole
CollaborationUse
Comment
Edges
Link
Connector
RoleBinding
Generalization
Realization
Substitution
Abstraction
SmartSoft
SmartTask
SmartTimer
SmartSendHandler
SmartQueryHand...
SmartEventTestH...
SmartPushTimed...
SmartPushNewe...
SystemConfiguration
Deployment
Properties
SmartParameterEditor
Console
Problems
Progress
1 items selected

No consoles to display at this time.

Figure 7.7: Screenshot of the SmartMDSD Toolchain showing the system configuration view of the Personal Mobile Navigation Demonstrator.

The baseline and software components of these demonstrators were picked up and composed to a total of 18 demonstrators by the FIONA project in the domain of location-based services and mobile navigation for sighted and visually impaired people [Bür+16] (see section 7.5.3 and Table 7.2). One of them is the *"Visual Localization Demonstrator"* using feature-based visual localization for personal navigation in an office-space at Comland d.o.o in Slovenia (Fig. 7.8, video at [SFC15]). It is an example of separation of roles as the role contributing the localization component is a commercial technology provider for visual localization for visually impaired people. The demonstrator can be seen as an example for modifying an existing system: It was first set up using a simulator, then composing the application such that it works in a real office environment. It utilizes a different set of hardware than the previously described demonstrator. It uses a laptop in a backpack to run high-demand visual localization algorithms and an iPad to provide the final look and feel using a handheld device (Fig. 7.8). Fig. 7.9 is a screenshot from the SmartMDSD Toolchain showing the composition of components in the system configuration view. Figure 7.10 shows the mapping of component instances to the target device in the deployment view. Section 7.5.3 explains and evaluates the efforts in composing this demonstrator.

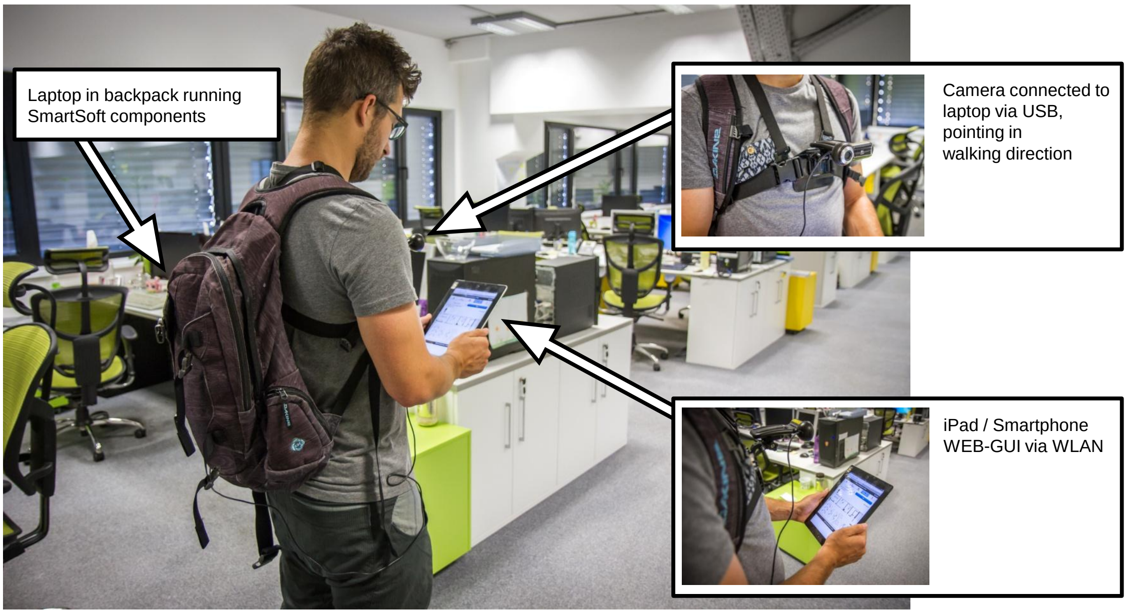


Figure 7.8: The "Visual Localization Demonstrator" for personal navigation in an office space. Video at [SFC15].

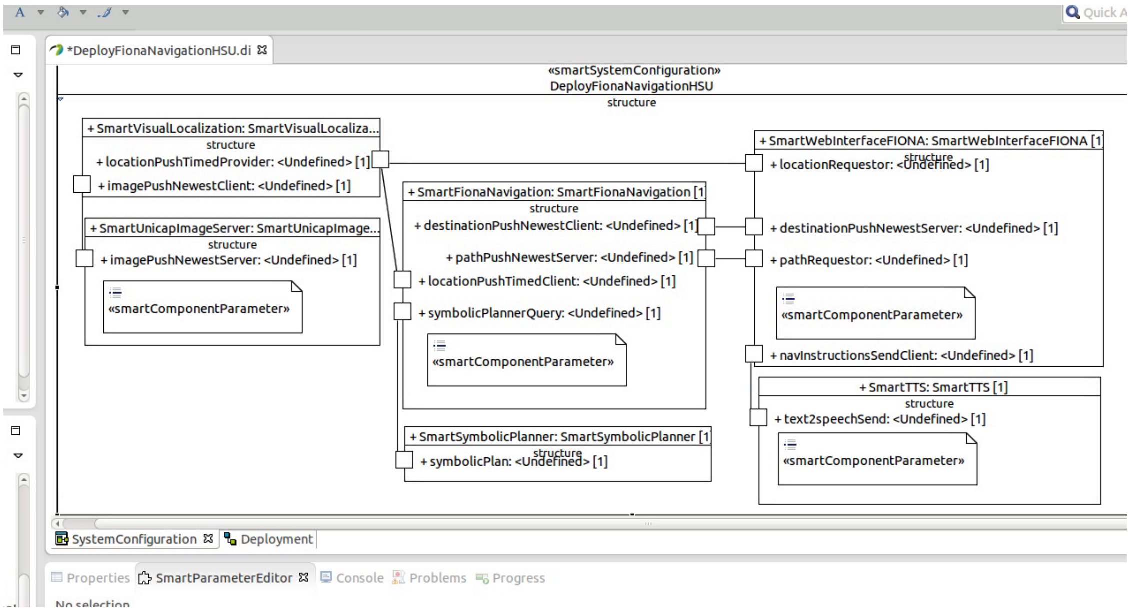

*DeployFionaNavigationHSU.di
Quick A
«smartSystemConfiguration»
DeployFionaNavigationHSU
structure
+ SmartVisualLocalization: SmartVisualLocaliza...
structure
+ locationPushTimedProvider: <Undefined> [1]
+ imagePushNewestClient: <Undefined> [1]
+ SmartUnicapImageServer: SmartUnicapImage...
structure
+ imagePushNewestServer: <Undefined> [1]
«smartComponentParameter»
+ SmartFionaNavigation: SmartFionaNavigation [1
structure
+ destinationPushNewestClient: <Undefined> [1]
+ pathPushNewestServer: <Undefined> [1]
+ locationPushTimedClient: <Undefined> [1]
+ symbolicPlannerQuery: <Undefined> [1]
«smartComponentParameter»
+ SmartSymbolicPlanner: SmartSymbolicPlanner [1
structure
+ symbolicPlan: <Undefined> [1]
+ SmartWebInterfaceFIONA: SmartWebInterfaceFIONA [1
structure
+ locationRequestor: <Undefined> [1]
+ destinationPushNewestServer: <Undefined> [1]
+ pathRequestor: <Undefined> [1]
«smartComponentParameter»
+ navInstructionsSendClient: <Undefined> [1]
+ SmartTTS: SmartTTS [1]
structure
+ text2speechSend: <Undefined> [1]
«smartComponentParameter»
SystemConfiguration
Deployment
Properties
SmartParameterEditor
Console
Problems
Progress
No selection

Figure 7.9: A screenshot from the SmartMDSD Toolchain showing the system configuration view of the “Visual Localization Demonstrator”.

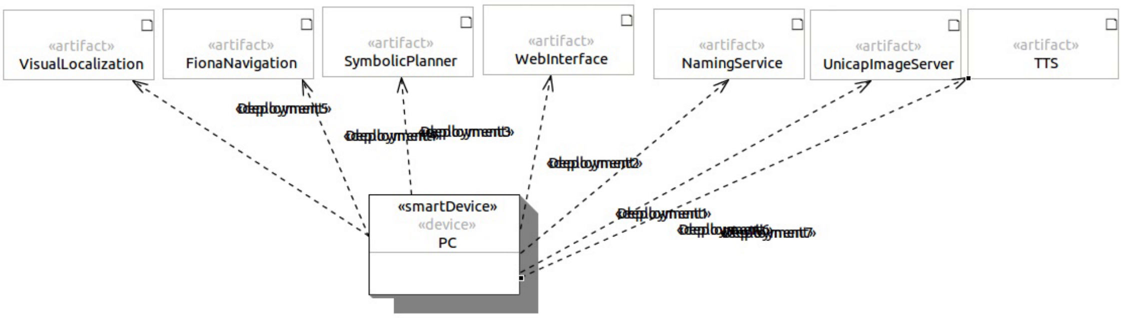


Figure 7.10: A screenshot from the SmartMDSD Toolchain showing the system deployment view of the “Visual Localization Demonstrator”.

7.3.3 Industry 4.0 Robot Commissioning Fleet in Intra-Logistics

The SmartMDSD Toolchain was used to develop a testbed for industry 4.0 and intra-logistics by the BMBF SME innovation project *LogiRob ("Multi-Robot-Transportsystem im mit Menschen geteilten Arbeitsraum")* [Bun]. The *Industry 4.0 Robot Commissioning Fleet in Intra-Logistics* scenario (Fig. 7.11, video at [Lut+17]) demonstrates the effectiveness of system composition by using existing components. It was composed out of existing building blocks, configured, and tested by only one person within only two working days. The scenario demonstrates the use of the SmartMDSD Toolchain in a state of the art and industry-relevant use-case. The scenario was built by composing building blocks supplied by academia and industry, but also by composing existing components from the SmartSoft ecosystem.

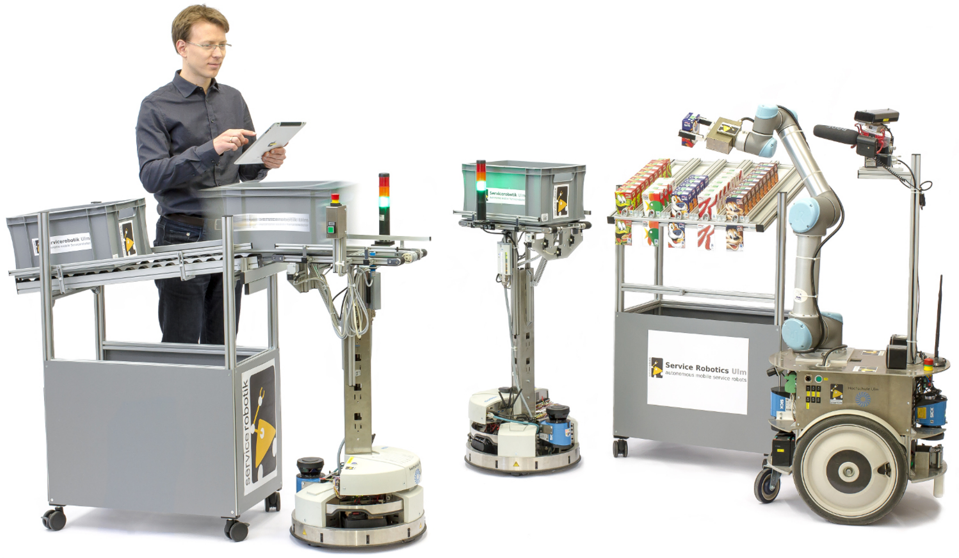


Figure 7.11: The "Industry 4.0 Robot Commissioning Fleet in Intra-Logistics" scenario. Three robots feature: navigation in a robot fleet, mobile manipulation for order picking, and handling of boxes. Video at [Lut+17].

In the scenario, a fleet of FESTO Robotino3 [REC16] robots and the UR3 equipped service robot "Larry" collaboratively execute commissioning of placed orders (Fig. 7.11). The scenario shows localization, manipulation for order picking, machine-to-machine communication, and handling of boxes (human-involved pickup and automatic delivery). A highlight in this scenario is the corridor-based navigation that organizes the robot's movements in a space shared with humans and avoids blocking situations between robots even in very narrow space[1] [LVS16]. The robots communicate directly (machine-to-machine communication). Only dispatching of orders and maintaining the reservation of navigation corridors is managed by a central component. It itself runs onboard of one of the Robotino robots with which the fleet communicates. The scenario was tested in a real logistics warehouse.

[1] The complete scenario is run in a 5x5m environment including additional shelves and a kitchen counter.

7.3.4 Transportation Tasks in a Hospital

The research project *iserveU (Intelligente modulare Serviceroboter-Funktionalitäten im menschlichen Umfeld am Beispiel von Krankenhäusern)* [Gin+16] developed an intelligent service robot for transportation tasks in a hospital (Fig. 7.12). The motivation in this project is to assist hospital staff with transportation tasks. The system was tested in a public part of a hospital during normal operation hours.

iserveU demonstrates that the approach also can be applied in a local perspective for straightforward development of a single robot or establishing a closed / intra-organizational ecosystem (see section 4.3.2). The composition workflow that is proposed by this thesis also structures the development in this kind of activities. Here, the separation of roles contributes to the robot development. The project used the SmartMDSD Toolchain for "straight-forward" system development, i.e. "team collaboration" (see section 3.1.1). The project started to define the overall system architecture by defining services. It was then implemented using the toolchain. The partner's contributions in 14 software components (Fig. 7.13) were integrated to the demonstrator in several iterations (Fig. 7.12).

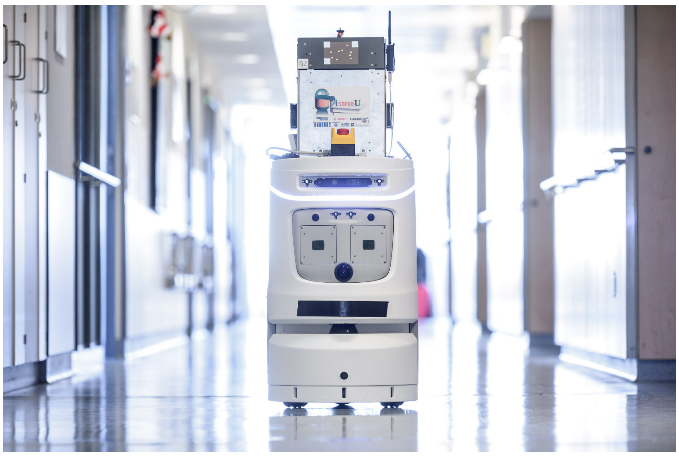


Figure 7.12: The iserveU hospital transports robot (figure from [Gin+16]).

The iserveU robot is based on a FESTO Robotino [REC16] platform. The robot showcases industry-grade radar-based localization technology and navigation in human environments. iserveU is an example of a collaboration with three partners from industry, four partners from academia, and also one end-user (the hospital). The consortium used the SmartMDSD Toolchain to model and implement the modular service-oriented and component-based architecture of the robot.

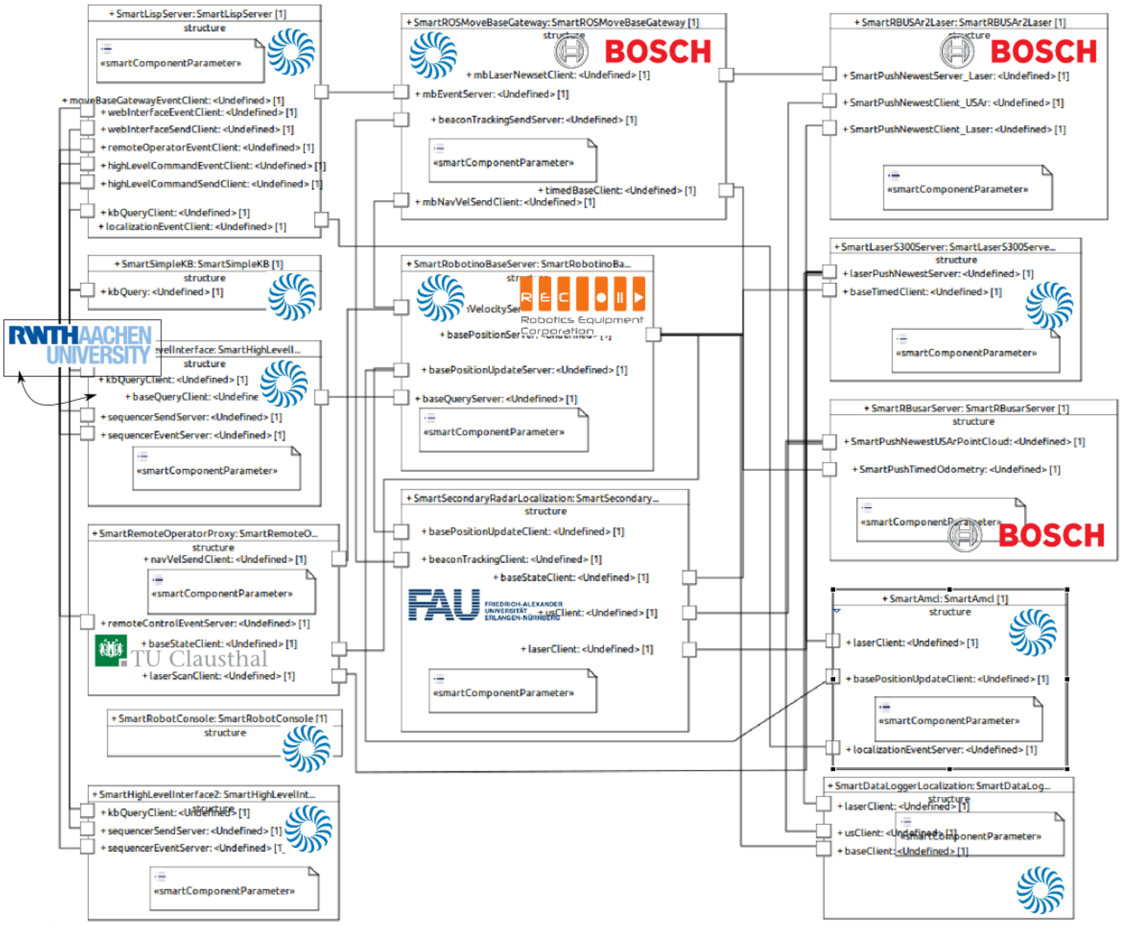

+ SmartLispServer: SmartLispServer [1]
structure
«smartComponentParameter»
+ moveBaseGatewayEventClient: <Undefined> [1]
+ webInterfaceEventClient: <Undefined> [1]
+ webInterfaceSendClient: <Undefined> [1]
+ remoteOperatorEventClient: <Undefined> [1]
+ highLevelCommandEventClient: <Undefined> [1]
+ highLevelCommandSendClient: <Undefined> [1]
+ kbQueryClient: <Undefined> [1]
+ localizationEventClient: <Undefined> [1]
+ SmartSimpleKB: SmartSimpleKB [1]
+ kbQuery: <Undefined> [1]
RWTH AACHEN UNIVERSITY
+ SmartHighLevelInterface: SmartHighLevelI...
+ kbQueryClient: <Undefined> [1]
+ baseQueryClient: <Undefine
+ sequencerSendServer: <Undefined> [1]
+ sequencerEventServer: <Undefined> [1]
+ SmartRemoteOperatorProxy: SmartRemoteO...
+ navVelSendClient: <Undefined> [1]
+ remoteControlEventServer: <Undefined> [1]
+ baseStateClient: <Undefined> [1]
TU Clausthal
+ laserScanClient: <Undefined> [1]
+ SmartRobotConsole: SmartRobotConsole [1]
+ SmartHighLevelInterface2: SmartHighLevelInt...
+ kbQueryClient: <Undefined> [1]
+ sequencerSendServer: <Undefined> [1]
+ sequencerEventServer: <Undefined> [1]
+ SmartROSMoveBaseGateway: SmartROSMoveBaseGateway [1]
BOSCH
+ mbLaserNewsetClient: <Undefined> [1]
+ mbEventServer: <Undefined> [1]
+ beaconTrackingSendServer: <Undefined> [1]
+ timedBaseClient: <Undefined> [1]
+ mbNavVelSendClient: <Undefined> [1]
+ SmartRobotinoBaseServer: SmartRobotinoBa...
VelocitySer
REC Robotics Equipment Corporation
+ basePositionSer
+ basePositionUpdateServer: <Undefined> [1]
+ baseQueryServer: <Undefined> [1]
+ SmartSecondaryRadarLocalization: SmartSecondary...
+ basePositionUpdateClient: <Undefined> [1]
+ beaconTrackingClient: <Undefined> [1]
+ baseStateClient: <Undefined> [1]
FAU FRIEDRICH-ALEXANDER UNIVERSITÄT ERLANGEN-NÜRNBERG
+ usClient: <Undefined> [1]
+ laserClient: <Undefined> [1]
+ SmartRBUSAr2Laser: SmartRBUSAr2Laser [1]
BOSCH
+ SmartPushNewestServer_Laser: <Undefined> [1]
+ SmartPushNewestClient_USAr: <Undefined> [1]
+ SmartPushNewestClient_Laser: <Undefined> [1]
+ SmartLaserS300Server: SmartLaserS300Serve...
+ laserPushNewestServer: <Undefined> [1]
+ baseTimedClient: <Undefined> [1]
+ SmartRBusarServer: SmartRBusarServer [1]
+ SmartPushNewestUSArPointCloud: <Undefined> [1]
+ SmartPushTimedOdometry: <Undefined> [1]
BOSCH
+ SmartAmcl: SmartAmcl [1]
+ laserClient: <Undefined> [1]
+ basePositionUpdateClient: <Undefined> [1]
+ localizationEventServer: <Undefined> [1]
+ SmartDataLoggerLocalization: SmartDataLog...
+ laserClient: <Undefined> [1]
+ usClient: <Undefined> [1]
+ baseClient: <Undefined> [1]

Figure 7.13: The system configuration model of the iserveU hospital transports robot (screenshot from the SmartMDSD Toolchain; system compositor role). An example of a system with components composed from different suppliers (note the organization logos). Figure from [Gin+16].

7.3.5 Application in Education

The SmartMDSD Toolchain and a set of components were used in the *lecture "autonomous mobile robots"* at Ulm University of Applied Sciences (bachelor's course). The complexity of a moving robot is already high for students with no robotics expertise at all. The SmartMDSD Toolchain allowed them to gain advantage as system builder (via system compositor role) from composing existing building blocks. The students re-used path planning, mapping, device components (laser and robot platform), and obstacle avoidance. They developed small parts of the system as component supplier, e.g. to detect and approach objects. The SmartMDSD Toolchain allowed them to focus on only one component to implement their algorithms while re-using the other components. The component was tested in a deployment for the simulator first, before the component then was deployed to the real robot. This lecture is one example where the model-driven approach reduces complexity such that even students with no robotics expertise at all can build and manage a robotics system with non-neglectable complexity. Separation of roles and the need to reduce system complexity was applied in a very realistic manner: There is no chance for the students to understand the details of the existing components—they just must rely on and use the building blocks that other roles provided.

The toolchain was used in several generations of student projects [SmartBots] (master's course) in the RoboCup team of the Ulm University of Applied Sciences. They started in the *RoboCup@Home league* with the service robots Kate, and later Larry (section 7.3.1). The robots had to interact with humans, perform robust navigation, recognize and manipulate objects. The team recently moved to the *RoboCup FESTO logistics league* using FESTO Robotino3 [REC16] robots. The focus in this league is on fleet coordination and interaction with machines in a production flow. These scenarios already bring the full system complexity of service robots that includes path planning, localization, obstacle avoidance, human–machine interaction, object recognition, mobile manipulation, and error handling in the real world. The students formed teams for only one year during their studies with very limited capacity in time. Each year, the competition rulebook changed and required modifications of the scenario that was taken over from the previous team. Modification mostly included adding new robot capabilities in new software components, thus extending the existing service architecture and modifying the overall task sequencing (robotics behavior). There was no handover in terms of overlapping time or overlapping persons between two consecutive teams. The students, being no robotics experts, of course, had no chance to understand and dig into every component. Thus, they had a practical need for separation of roles and to re-use building blocks 'as-is' and to rely on their composability. The composition of components—thanks to the toolchain and the underlying structures—were the key in this long-term project to enable the students to modify the system flexibly according to the needs of the updated rulebook and to migrate from Kate to Larry. Thanks to the toolchain, they were able to focus on their contributions and to handover the newly built components to the next team.

7.4 User Study

A user study was conducted to evaluate the perceived benefits and user's experience in applying the approach and the SmartMDSD Toolchain. It was published in [Sta+16]. This section briefly introduces the study and summarizes its results before the next section then uses the results to evaluate the approach in a project context.

The study was conducted with partners of active research collaborations. There were 18 responses to the survey. These responses are of high quality since they cover a wide range of roles with representative qualifications and level of expertise. Responses came from industry (45%) and academia (55%) including technology providers, system integrators and experts from different domains. They were all professionals with backgrounds in embedded systems, automotive, robotics, artificial intelligence, computer vision, localization, and general sensors and control. The participants and number of responses thus provides a representative and high quality contribution to the survey.

The anonymous questionnaire included 44 questions and it took 20–30 minutes to complete it. Most of the answers were formulated using five-level Likert items [Lik32] from "strongly agree" over "neutral" to "strongly disagree".

Within their company or institution, the participants are using a very heterogeneous set of development tools and integration methods. From the free-text answers, it can be concluded that an integrated IDE, such as the SmartMDSD Toolchain, is not a standard within these groups and that they apply a class-based integration and reuse philosophy.

Even though the SmartMDSD Toolchain is not a final product but a research development, 82% of the participants say that they can work productively using it. Almost all participants (94%) saw a fundamental contribution to system composition out of software building blocks to build up new applications.

Even though the service definitions were not included in the productive release of the SmartMDSD Toolchain at the time of conducting the study, they were part of the overall process that the users applied, most explicit in FIONA (see section 7.2). The results of the study in this context are thus considered valid for evaluation since the users were able to perceive their benefits even without an implementation.

Software metrics are an approach for determining the quality of a product or process. Using software metrics to compare the benefit of development tools or processes can be cumbersome with high effort as shown by Basili [BBM96]. It requires multiple applications to develop, multiple teams with comparable skills, the same definitions of goals, and evaluation with and without the approach. Furthermore, it requires meaningful metrics—which itself is subject to debates and active research since the introduction of metrics. Conducting a user study based on a survey is, thus, considered more reasonable to evaluate the approach. In contrast to software metrics, this is a subjective evaluation by the participants that is also influenced by the fact that they know the purpose of filling out the survey (cf. Hawthorne effect [ER03, p. 232]). But it is an evaluation with users of the target audience under real conditions in practice. The participants used the toolchain and the approach over an adequate period of time (1 year). When using a shorter period of time, the users will not get beyond the period of training and familiarization and, thus, will not see the benefit in reducing the problem complexity—they might instead perceive the solution complexity only.

7.5 Evaluation in Project-Context

This section highlights the experience and perceived benefits of using the approach and the SmartMDSD Toolchain in context of the presented demonstrators and projects. The results of the user study support the arguments. The numbers given relate to the overall study, unless mentioned otherwise (in few cases the numbers only relate to a specific project). The evaluation focuses on the FIONA project and is divided in three categories:

Project Collaboration describes the work and interaction within the project: how the approach helped to organize and structure the collaboration towards functional demonstrators.

Software Development describes the user-view: how the approach supported the individual project, partner or user during development.

System Composition describes the combination of both on a higher level: how the approach supported to put the pieces together and compose demonstrators.

7.5.1 Benefits for Project Collaboration

From a non-ecosystem perspective, the workflow based on service definitions structured the collaboration between project partners. At a very early stage in the project, the partners had to think about the technical details of provided or needed functionality. Explicitly writing down service definitions triggered the need to think and agree on data structures, communication semantics and assumptions, i.e. service properties. This supported in identifying white spots (e.g. a component service is needed by someone, but not provided by anyone) and sort out mismatching assumptions at an early project stage. The benefit of service definitions is not only visible in the result but also in the process of finding them, because questions are asked and options are evaluated. The very same benefit is also visible in architectural design [Sta15; BCK12]. In iserveU and FIONA, for example, the project phase of defining services was a key activity towards the successful project demonstrators. Especially in FIONA service definitions fundamentally shaped the project collaboration and set the basis for flexible composition of demonstrators. The result of this activity are the domain structures on composition Tier 2 and those were one of the project goals in FIONA.

The study showed that the benefit of structuring the project collaboration is also perceived by the participants: All (93%) agreed that the overall development workflow, the service definitions and the SmartMDSD Toolchain structured the project collaboration towards functional demonstrators (31% said it was a great help, 62% said it helped). The benefit of explicit service definitions in system design was seen by the participants (94%). Most participants (65%) agreed that writing down service definitions improved the early identification of gaps in the overall functionality and enhanced the agreements between components.

FIONA was a realistic testbed for an ecosystem approach. Due to different funding schemes and schedules even within one project, the project consortium was very dynamic. Planned partners did not get funding and never started working. New partners joined very late during project runtime. Thanks to composition Tier 2 and service definitions, the late joining partners followed the domain structures and contributed components on Tier 3. This made up a project-internal ecosystem. Their contributions were "component alternatives" with various levels of performances (e.g. for use in other environments, better quality). Having a set of component alternatives to choose from immediately allowed to build up demonstrators with increasing performance (localization accuracy) or for use in new environments (indoor iBeacons vs. indoor feature-based localization vs. outdoor GPS). This demonstrates the need for assisted component selection based on the explicated application needs.

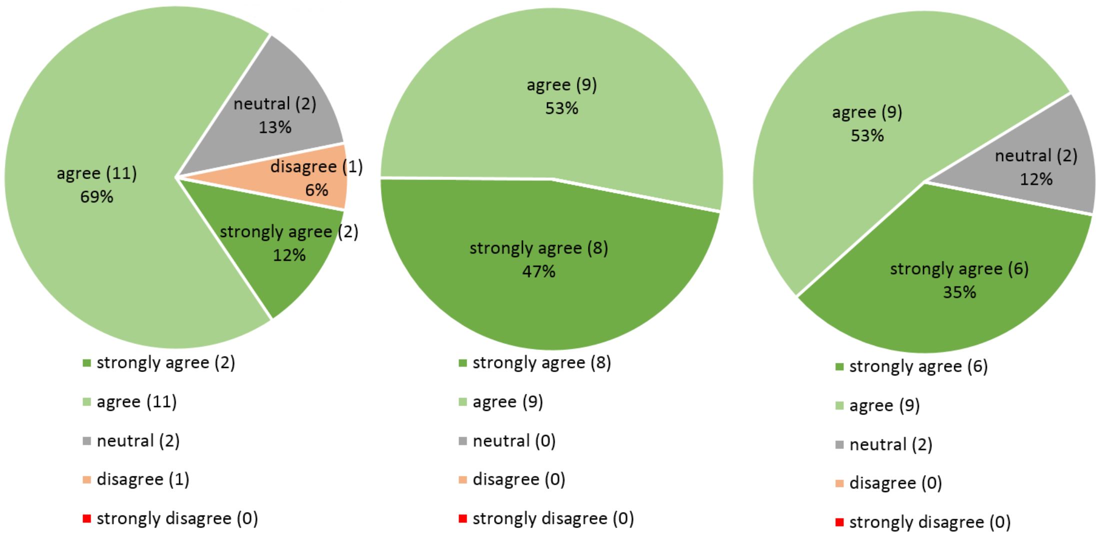
7.5.2 Benefits for Software Development

The SmartMDSD Toolchain makes the underlying concepts of the SmartSoft World accessible to its users, so they can immediately benefit from the advantages. This was beneficial to the mentioned activities. None of the users could spend time and effort to dive into the deep concepts and code-structures of component development. But still, this would be required to use them in order to come up with integrated results. All benefits of the approach are immediately available through the SmartMDSD Toolchain. Domain-specific structures were immediately accessible and component suppliers were able to start implementing business logic into generated skeletons. The SmartMDSD Toolchain provided the necessary workflow-support and guided through the steps of the composition workflow. Most participants (81% of the replies) shared this perception (Fig. 7.14a).

Service definitions decouple and separate the roles between component developers as well as between component developer and system compositor. This was a key factor in FIONA, where partners were distributed over countries, working on separate contributions with their own schedule. Component developers followed service definitions to ensure integration of their component in the demonstrators and to describe the needs of their component towards others. They were able to focus on their contribution and technical challenge without worrying about how it integrates into the overall demonstrators. All responses of the user study confirm that service definitions supported the distributed development both in time and space (Fig. 7.14b). Most participants (88%) experienced that they were able to focus on their role of expertise and contribution only, that is: separation of roles (Fig. 7.14c).

Changes in service definitions at component development or composition time trigger the need for mutual agreement between all involved stakeholders. The participants reported, that it was immediately clear to them who is affected in case mutual agreements were needed (supported by 94% of the replies).

Using the SmartMDSD Toolchain, the users can gain immediate advantage of code-generators. Users can immediately gain advantage from expert knowledge expressed in code-generators. Without using the SmartMDSD Toolchain, experience shows that components are set up very differently (e.g. organization of files, pattern instantiation, communication object access methods, internal component structure). Using the toolchain, the structure and content of user-files becomes uniform through the described integration of user-code with skeletons. They are thus easier to understand and navigate. This significantly improves maintenance of components and shared component development within different persons in one role.



(a) The SmartMDSD Toolchain provides the necessary workflow-support and guides users through the overall development process.

(b) Service Definitions supported the separation of development in time and space.

(c) I was able to focus on my field of expertise and contribution to the project (separation of roles).

Figure 7.14: An excerpt of the user study as published in [Sta+16].

7.5.3 Benefits for System Composition

FIONA has come up with 29 components that were composed to a total of 18 applications (Table 7.2). This demonstrates the capability of the approach to system composition. Even partners that developed only a small component could immediately build own demonstrators by composing and reusing other components from the project-internal ecosystem.

Most of the participants of the user study perceived a benefit of using the toolchain for system composition (56% with high benefit, 45% with benefit). Almost all (94%) see a fundamental contribution to system composition out of software building blocks to effectively build new applications.

Approximately 960 variants of the main project demonstrator might be composed, when considering all component alternatives based on service definitions (Table 7.2). These variants, however, distinguish in their performance (diversity of performance). There are, for example, five components that provide localization services. But each comes with different localization quality as they use different localization technologies or work indoors (using iBeacons), outdoors (GPS) or both (using MORSE simulator). Expressing this information and aligning it with the application's needs will identify the subset of the variants that can be considered for composition in a certain demonstrator. This is an example in which expressing and using properties supports the user: Otherwise, one would have to look for the documentation, even source code or testing and might recognize a mismatch late.

18 Demonstrators (functional Implementation)

29 Components (functional implementation)	Component supplier	Number of re-uses	Indoor simulation, native iPad GUI	HSU iBeacon Indoor Navigation	HSU iBeacon Museum Example	Indoor navigation (Morse simulator)	HSU/Comland	HSU/Bosch Integration	HSU indoor/outdoor navigation	GUI and Security	Bosch Beacon, ESK Fusion	Visual Localization	Belt Simulator	Havelsan Nav. Simulation	Authentication	Full Demonstrator	Security Door Demonstrator	Czech IMU Demonstrator	Integr. Concatel	HVS Multifloor
SmartBluetoothLocalization	HSU	4		x	x			x	x											
SmartBluetoothLocalization	Bosch	5									x					x	x	x	x	
SmartGpsdServer	HSU	1							x											
SmartMorseLocalization	HSU	7	x			x	x	x					x	x						x
SmartVisualLocalization	COM	2					x					x								
SmartTTS	HSU	4				x	x		x										x	
SmartTTSLoquendo	HSU	0																		
SmartTTSMary	HSU	0																		
SmartRBXsensOrientation	Bosch	0																		
SmartSensorDataFusionESK	ESK	4									x					x	x		x	
SmartXsensIMUMTiServer	HSU	1							x											
SmatIMUComponentCZ	CZ	1																x		
SmartFionaNavigation	HSU	12		x	x	x	x	x	x		x		x			x	x	x	x	
SmartPathPlanning	HVL	2												x						x
SmartSecurity (software)	HSU	2				x									x					
SmartSecurity (hardware TPM)	Infineon	5								x						x	x	x	x	
SmartBluetoothBeaconServer	HSU	7		x	x			x	x		x					x	x			
SmartRBBluetoothBeaconServer	Bosch	3																x	x	x
SmartWebInterface	HSU	16		x	x	x	x	x	x	x	x	x	x	x	x	x	x	x		x
SmartiPadNativeGui	HSU	1	x																	
SmartHapticBelt	Bosch	6						x					x			x	x	x	x	
SmartContextProvider	CCTL	1																	x	
SmartIndoorOutdoorHandover	HSU	1							x											
SmartMapProvider	HVL	2												x						x
SmartProfileProvider	HVL	2												x						x
SmartSecureDoor	Infineon	1															x			
SmartSymbolicPlanner	HSU	12		x	x	x	x	x	x		x		x			x	x	x	x	
SmartUnicapImageServer	HSU	1					x													
SmartWgs84ToCartesianConverter	HSU	1							x											
	Components used:		1	4	4	4	4	5	7	2	5	1	4	4	2	7	8	7	7	5

Co-Existence of component alternatives:
Tot. variants: 960

5x Localization
2x Credentials
3x Speech
2x Beacon
4x Pose Inform.
2x GUI
2x Path Planning

Table 7.2: A total of 18 demonstrators was composed from 29 components from a FIONA project-internal ecosystem. Colors indicate groups of component alternatives. Multiplying the number of alternatives in each group results in 960 possible variants of the project demonstrator—each with different performance (diversity of performance).

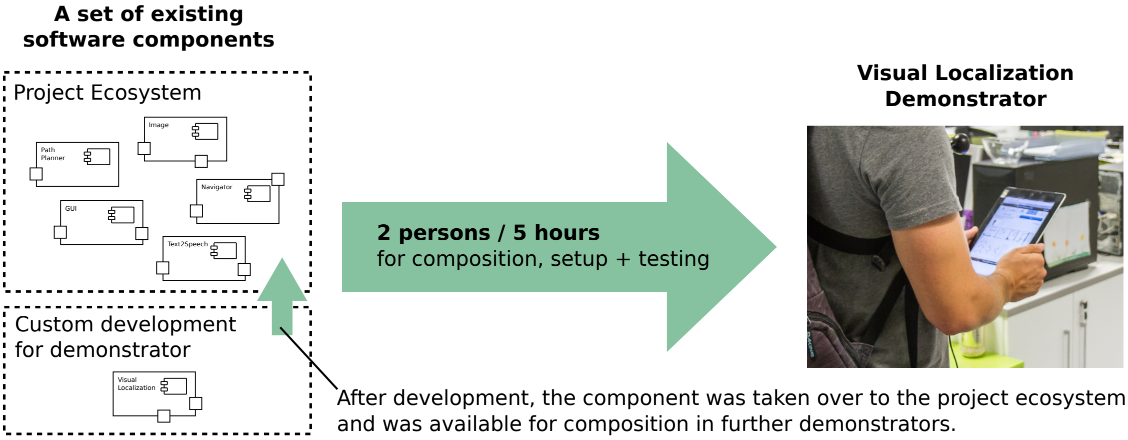
Service definitions reduce the pressure towards a "big bang integration" in late project stages. They provide the foundation for building early prototypes in which components can continuously be extended and improved, as it was applied in FIONA: For example, a basic implementation of a localization component with moderate localization quality based on iBeacons was used in early demonstrators. Later, a new component with a more advanced localization implementation was provided by a different partner. It offered much better localization quality and enabled scenarios that need more accuracy, e.g. for personal navigation for visually impaired people. A similar use-case can be found in the navigation-context of FIONA: A complete sub-architecture was exchanged by new components when a late-joining partner provided more powerful path planning and guidance solutions. A third example was already described with the butler scenario: exchanging object recognition by active object recognition.

Service definitions and properties supported to find and identify the service structure very early in the project, thus prevented problems at a later stage. In FIONA, for example, partners were asked to provide details of inputs and outputs of their components: many of them had the same purpose but different interfaces, both in syntax and semantic. If it were implemented this way, there would have been big integration effort in the end. Agreeing and expressing service definitions helped to overcome these differences. This is the purpose of composition Tier 2.

Perceived Composition Effort at Integration Workshops

To grasp the effort of system composition, three examples of integration workshops in FIONA are given.

Workshop 1 In an early integration workshop, the *"Visual Localization Demonstrator"* for personal navigation in an office space (section 7.3.2) was put in operation in one day by two people. Five of six components were taken from the project-internal ecosystem. One component for visual localization was developed prior to the workshop by one project partner. It adhered to the domain structures from composition Tier 2. The component was put into the project-internal ecosystem afterwards and was used for composition in other demonstrators. Besides setting up hardware, maps and testing, integration and composition itself was the work of few hours (approximately 6 hours for two persons): "We were able to integrate different components with an ease and in the matter of hours," as the chief technology officer of Comland d.o.o reported [SFC15] (Fig. 7.15). The "Visual Localization Demonstrator" is one example of successful system composition with low effort in context of the FIONA project.


A set of existing software components
Project Ecosystem
Image
Path Planner
Navigator
GUI
Text2Speech
2 persons / 5 hours
for composition, setup + testing
Visual Localization Demonstrator
Custom development for demonstrator
Visual Localization

After development, the component was taken over to the project ecosystem and was available for composition in further demonstrators.

Figure 7.15: The Visual Localization Demonstrator was put in operation by composition of five existing components. One component for visual localization was developed prior to the workshop by one project partner. It adhered to the domain structures from composition Tier 2. The component was put into the project-internal ecosystem afterwards and was used for composition in other demonstrators.

Workshop 2 During a midterm integration workshop, ten participants built up five demonstrators in three days. The total effort for system composition from the project-internal ecosystem using the SmartMDSD Toolchain was very low and took about three hours. This is confirmed by the user study that was conducted shortly after the workshop: Considering only the replies from FIONA, the participants (71%) perceived the composition effort "low" or "very low" in comparison to other activities of the integration workshop (Fig. 7.16). Thanks to the low effort for composition of software components, the participants were able to focus (separation of roles, taking the role of the system builder) on the actual and time-consuming tasks during such workshops: setting up custom hardware with the integrated demonstrator platform, configuration of components to the new environment (e.g. creating maps and waypoints), and testing the demonstrator.

Most of the scenario configuration was done using the component parameters of the SmartMDSD Toolchain and most of the participants (57%) perceived a low effort during the workshop. When explicitly being asked for parameterization in the overall study, the participants rated the parameterization mechanism as adequate (94%). Some components unfortunately did not follow the service definitions as defined in documents (see section 7.2), thus adjustments to components were necessary (Fig. 7.16, component development). Having service definitions fully supported in the toolchain would have prevented this, but even without them, the users perceived the necessary changes as low effort, thanks to the model-driven toolchain.

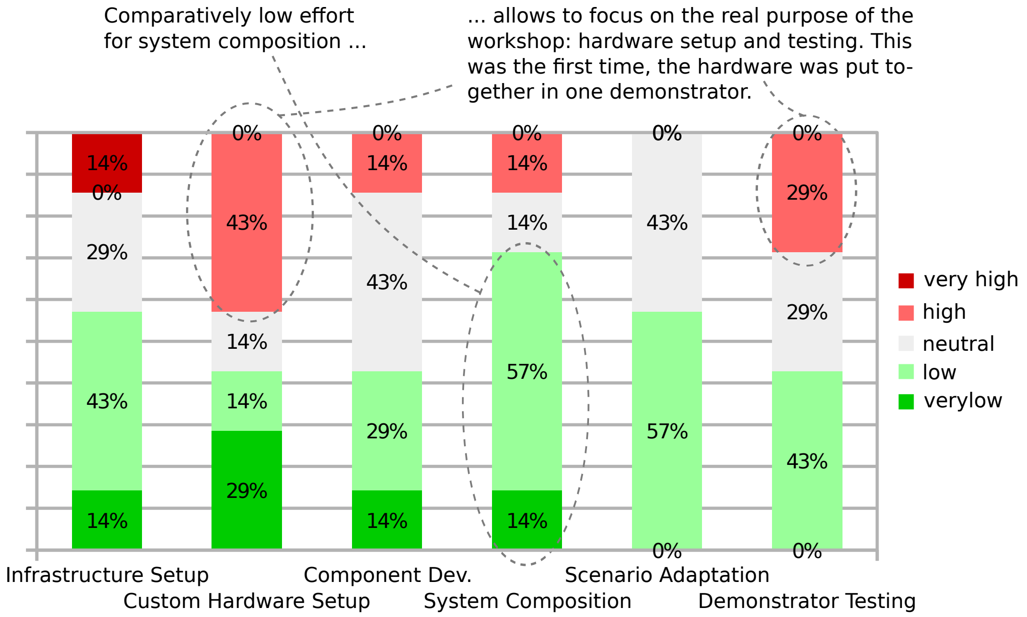


Figure 7.16: The efforts for system composition during a FIONA workshop as perceived by the participants. The categories show: Infrastructure Setup (e.g. networking, version control, OS, libraries), Custom Hardware Setup (e.g. bluetooth setup, serial drivers), Component Development (e.g. bug fixing, extending functionality), System Composition, Scenario Configuration (parameterization, creating maps, waypoints), and Demonstrator Testing. There are many neutral answers because the partners had individual responsibilities and were not active in all tasks.

Workshop 3 One workshop took place at a late stage in the project where the project-internal ecosystem has been established. At this workshop, two new versions of the existing project demonstrator for personal navigation have been composed within an hour to integrate contributions of two partners in the demonstrator: The demonstrator was extended by a more powerful navigation solution.

7.6 Discussion

This section reflects and discusses the presented approach in relation to the overall research goal and questions (see section 1.1).

> *How can software development for service robotics be improved in order to advance from handmade integration and crafting to systematic engineering of software based on system composition in a software ecosystem?*

The presented approach contributes to system composition by introducing a meta-structure and according workflow to create and use domain-specific structures based on service definitions. It builds on the concept of services of the service-oriented component-based SmartSoft framework. This raises the starting point of this thesis to an adequate conceptual level. The concepts behind the SmartSoft framework already provide basic means for system composition (e.g. a set of communication patterns and service orientation) and, thus, the presented approach also benefits from these. Successful robotics applications might be built without the approach presented in this thesis (asking developers to write clean code, use frameworks, best-practices and guidelines as documented). It is, however, argued that the level of composability in this case can only be realized with much more effort and more errors underway. At some point in time, this is not manageable anymore. This thesis provides means to structure domains in composition Tier 2 to overcome this.

The results showed the applicability of the approach for system composition to build new applications from existing building blocks. Composing a complete application (100%) from existing parts, admittedly, is theory. As Fröberg [Frö02] concludes, it is more practical to compose generic components and to develop custom solutions. Transferred to robotics, this means that basic skills can be composed and application-specific solutions are subject to custom development. Regarding robotics, several basic skills (navigation, localization, object recognition, speech, etc.) can be identified and it can, thus, be considered realistic to compose these basics according to the needs and develop others to solve custom skills. This is a huge step forward in the overall vision of an ecosystem for robotics since technology immediately becomes accessible. To address system composition at large, other areas of composition besides components and services need to be addressed as well. This includes both composing other kinds of building blocks (e.g. task sequencing and action plots as motivated in [SS14b], hardware models) and other aspects of composition and composability of building blocks (e.g. timing).

> *1. How to improve the **composability** of building blocks (software components) such that they not only fit together technically, but also work together in a meaningful way in the overall application?*

Service definitions are a key in the proposed meta-structure and workflow. Their definition on composition Tier 2 improves the composability of services offered by software components towards system composition (Tier 3). They include service properties that allow to express and use the semantics of services on the application-level, which otherwise remain hidden within code and documentation or must be managed manually. Thanks to constraints, properties can be used to express the needs and offer between the component's service endpoints and set the basis for assisted component selection.

Restricting the use of properties and constraints to single property attributes only might not suit all use-cases. Enabling constraints to consider multiple properties might thus be considered. This would bring an advantage when relations between properties exist or a certain set of properties usually occurs in combination. The approach does not realize this so far, but the service definition as an existing model provides the foundation to do so, for example by defining the constraints on the level of the service definition.

The list of properties that are available when instantiating a service definition is a fixed set; only the properties that are included in the service definition can be used. This might be perceived as restricting, and users might wish to extend the set of properties. However, this goes back to weighing freedom from choice versus freedom of choice. It would be possible not to be bound to a fixed set of properties in the service definition, but the properties must be modeled externally for reuse in service definitions. Modeling properties during instantiation of the service definition will remove their grounding and hinder their evaluation. Limiting the list of properties in the service definition, however, also provides guidance. The user knows that these properties are available as opposed to an open and rather infinite set of properties that he does not know about.

The expressiveness of static service properties with respect to their use for qualities is rather limited as the quality might depend on factors beyond the service. It might depend on the quality of an input-service or even on runtime-factors (the current situation of the robot or the environment). Extending service properties to more dynamic and interlinked properties is thus one of the major directions to be addressed in future research. Expressing static properties, however, contributed a necessary step in this direction as the properties are now accessible and integrated in the structure and workflow—it is now a matter of where their values originate from. Expressing qualities with static properties improves composability and assist in component selection, as static properties allow for expressing qualities that can be achieved in principle.

> *2. How to organize the building blocks (software components) in an overall **composition workflow** that decouples and manages both the stakeholders and the parts that they supply or use to collaborate in an ecosystem?*

An overall composition workflow was presented. It is based on service definitions as the key model to decouple between component development and system composition. Several metamodels have been presented to cover the workflow.

Service definitions contribute to the separation of roles in the overall workflow since they decouple between component development and system composition. The user study confirms that the involved roles can focus on their task only. One of the reasons behind successful separation of roles in the approach lies in the separation of the models which proved to be effective. As soon as different people contribute, the separation of roles cannot be realized with separated views (problem-specific views) or designated working areas (restricting access, user and rights management) within a single model only. Separation of roles can be improved through these. They are, however, not sufficient for the separation of roles for system composition towards an ecosystem. Organizing the models in separate, harmonizing but well-linked elements separates them physically and provides the necessary distance. Regarding the roles in a workflow for system composition, there are more roles involved than are presented in the approach. These roles, the corresponding models and views must be integrated in the workflow. Depending on the particular case, this might influence the presented models since this cannot be done in isolation. Integrating roles for task sequencing, for example, influences the component development step and the step of building the application where the overall action plot of the application is modeled.

The key for separation of roles is to manage the handover and connection between them. The approach provides service definitions to realize this handover with respect to services between the workflow steps. The service definition model thus is an element that is integrated in the workflow and can be extended towards new requirements with respect to services.

The approach was applied in the FIONA research project and several demonstrators showed its suitability for system composition. Flexible system composition as shown is possible thanks to the meta-structure that organizes the ecosystem in three composition tiers. The domain structures on Tier 2, and the service definitions therein, cannot be modified by component suppliers and system builders. However, the missing flexibility when using service definitions was not perceived as a disadvantage: Most participants of the study (88.3%) recognized the benefit in identification of service definitions and in the agreement on service definitions. They felt that the stability that was introduced at the cost of flexibility was a good deal.

The proposed workflow covers all steps up to deployment and running the application. The deployment model covers enough information for the workflow to be complete to demonstrate the full picture for system composition. It needs, however, extension of important information that is necessary to fully model the execution of software components, e.g. CPU architecture of the device or networking between devices.

Oster and Wade [OW13] argue that the discovery of building blocks is a major hurdle towards establishing a successful ecosystem. Service definitions support in expressing the needs of the application and to select suitable software components. How these service definitions are discovered in the first place, however, needs to be addressed as well. Components can be found based on their services, but finding the service definitions to express the needs might happen more freely. Eventually, the user is not yet looking for a specific service definition because he has not yet decided what service he is going to implement for the custom component. An approach to discover service definitions might, thus, be realized through simple searches, e.g. based on tags and free-text search or utilizing ontologies to categorize service definitions.

If any structure becomes established to serve an ecosystem also depends on aspects beyond the technical suitability. It also depends on its acceptance, widespread dissemination and even political aspects. The proposed meta-structure therefore does not provide a single fixed structure but a tool to define custom structures. Structures thus might co-evolve, might converge to de-facto standards and thus achieve high-impact. This, however, is beyond technical aspects. It needs dissemination and even political activities.

> 3. *How to design an integrated tool that* ***supports users*** *in modeling and composition of models and corresponding software artifacts through the workflow for system composition?*

The Eclipse-based SmartMDSD Toolchain applies model-driven techniques and provides DSLs for graphical and textual modeling in an IDE to support users in creating and using the meta-structure for system composition. The toolchain was used in research projects and the study showed that it significantly improves system development.

The study confirmed that tool-support is a necessary part for successful system composition and that the SmartMDSD Toolchain can be used productively for this purpose. The mixture between graphical and textual models in the SmartMDSD Toolchain is adequate according to the study. The realization of graphical modeling in Unified Modeling Language (UML) was considered reasonable for a toolchain for productive use thanks to low effort through existing graphical tools and notations that are familiar to users. Drawbacks, however, were experienced in using

UML for this purpose. UML profiles are suitable when the semantic and notation of the original UML elements are reasonable for and compatible with the elements that are modeled for the approach. There is, however, a high modeling effort when the elements of UML do not directly map to elements of the approach. There is even the danger of modifying the original semantics of UML. It was observed, that users follow their own knowledge when modeling in UML. The result sometimes does conform to and sometimes does not conform to the UML standard. Since UML is very open, this leads to models that represent what the user thinks, but these models might not correspond to the described approach. There is, thus, a high effort for a tool-provider to restrict the usage to match the approach to get valid models. Interestingly, the SmartMDSD Toolchain was more familiar in its use for users that only have few or no UML knowledge. Initially, using a known modeling approach was thought to be an advantage. Notably, Bonnet, Voirin, Exertier, et al. [Bon+16] come to the same overall conclusion and lessons learned in UML profiling with the Capella tool that is inspired by UML and SysML (see section 2.3.1). It is, finally, about the structure that sets the basis for and is independent of the implementation in any technology. The use of UML profiles in the SmartMDSD Toolchain for graphical modeling together with Xtext for textual modeling was low effort due to existing tooling. It was reasonable for a research toolchain to come up with tooling that actually can be used to build systems. It showed its benefit despite these drawbacks. They can be addressed when turning the toolchain into a commercial product.

Component selection and matchmaking eases the access to a component market. Examples of component selection have been shown, but there is room for more user support in tooling. For example, the component selection process only considers one service wish against one component service: The selection mechanism will not suggest a component that might provide two wished services over another that only provides one. Furthermore, component selection should not only consider match and mismatch as Kritikos and Plexousakis [KP08] argue. In case the application's needs are very tough and no component meets the needs, is desirable to assist the user in suggesting which application needs shall be relaxed such that a suitable component can be found. This is not about "overwriting" the needs, but about knowing what will happen when a component is selected which does not meet the needs. Relaxing the needs might be an option over developing an in-house component from scratch.

7.7 Summary

This chapter described several real-world robotics systems that were built using the approach. They demonstrate the benefit of the approach to address system complexity and flexibility in composition. Several research projects and collaborations were involved in building these real-world robots. The chapter evaluated the positive influence of system composition, separation of roles, and ecosystem collaboration on these projects. A user study was presented that evaluates the perceived benefits as well as the user's experience in applying the approach and the SmartMDSD Toolchain. The chapter closed with a discussion of the contributions of the thesis in the context of the research questions.

The next chapter concludes the thesis and provides an outlook for future work.

8
Conclusion

This chapter concludes the thesis. It summarizes the contributions and explains their applicability and relevance. The thesis finally ends with an outlook on future work.

Current practice in software development for service robotics resembles crafting and uses integration-centric development approaches: A robotics application is designed and then broken down into smaller pieces that are solved individually at a lower level of complexity. What follows is the integration of these pieces to a system using adapters. While this approach is adequate to reduce the complexity of a single robotics application, it hinders the reuse of parts in other applications. It also hinders the collaboration of experts in specific domains. In particular, such collaboration is essential in an interdisciplinary domain such as robotics. It also hinders the flexible reuse of existing (third party) building blocks to come up with new applications in an efficient way.

The approach presented in this thesis achieves the step change from handmade crafting and integration of software to a systematic engineering approach based on system composition in the context of a software business ecosystem for robotics. The approach organizes an ecosystem for robotics software in three composition tiers: a tier for general structures that enables composition[1], a tier for domain-specific structures, and a last tier for building blocks and their use. The approach applies separation of roles to manage the interaction and collaboration as a key factor to support the transition from fixed value-chains to flexible value-networks. The approach utilizes model-driven techniques to overcome crafting and to enable a composition-oriented approach that lowers effort, cost, and the time to market.

Structures are needed to enable system composition in an ecosystem. This thesis has presented a meta-structure that provides the necessary framework for enabling composition in accordance with the needs of service robotics. The meta-structure is based on Component-Based

[1] A meta-structure for composition, also called the "composition structure" in short

Software Engineering (CBSE) and Service-Oriented Architecture (SOA). It allows explicating and using knowledge which is relevant for composition but which usually remains hidden. Using the meta-structure and the presented composition workflow (composition Tier 1), domain experts can express (model) recurring concepts (structures) of their domain to set the framework for composition (composition Tier 2). Component suppliers and system builders (composition Tier 3) adhere to these concepts to supply and compose building blocks independently. The composition structures ease the development of composable software components.

The composition workflow is based on service definitions that are modeled using a Service Definition Language (SDL) on composition Tier 2. Service definitions provide basic reusable elements in the workflow to realize separation of roles as they form a bridge between component supply and system composition. They increase the composability of services. Service properties were presented to express the semantics of a service on the application-level. Expressing constraints on service properties ensures the composability of components that provide or require services. The system builder is supported in selecting adequate components according to the application's needs.

In addition to the core meta-models for service definitions, service properties, and components, several other meta-models have been introduced to cover the necessary infrastructure to complete the composition workflow. Overall, the following meta-models have been presented:

- the *service definition* meta-model and its building blocks: the meta-model for modeling *communication data structures* and the meta-model for *service properties*
- the *parameter* meta-models to define, instantiate, and use variation points for components
- an extension of the SmartMARS *component* meta-model to use it with service definitions
- the *wish list* meta-model to express the needs of an application based on service definitions
- the *system configuration* meta-model to compose an application from existing components
- the *deployment* meta-model to map instances of software components to execution units

The approach is implemented in the model-driven SmartMDSD Toolchain. It provides an Integrated Development Environment (IDE) for software development in service robotics to support users in applying the composition workflow. The toolchain plays a key role in providing access to the structures and in providing guidance for successful ecosystem collaboration. The toolchain covers the design of services on composition Tier 2, the development of components on composition Tier 3, and the composition of components to systems and finally their deployment. Several Domain-Specific Languages (DSLs) have been developed that allow to seamlessly create and use the necessary models between the involved roles in the composition workflow. Based on the needs of the application under development, the SmartMDSD Toolchain provides support in selecting components from the envisioned component market in the software ecosystem.

The results and evaluation of the approach show that the proposed organization of an ecosystem in three composition tiers, the composition structure, and the implementation within the SmartMDSD Toolchain contribute to the composition of new applications by using existing building blocks (system composition). A user study was conducted to evaluate the benefit and experience as perceived by users of the SmartMDSD Toolchain. The study confirmed that the concept of service definitions contributes to the separation of roles by structuring their collaboration, and allowing them to focus on their contribution and expertise. The approach was applied to build a number of systems in a broad field of domains and projects. This ranges from custom developments to establishing a project-internal ecosystem, from users with no robotics expertise to highly skilled robotics experts, from systems in simulation to real-world systems, and from service robotics to the smartphone domain. The approach was used in various projects and activities ranging from the goal of developing a specific robot application from scratch up to using the approach in an ecosystem. The ecosystem approach was specifically applied and evaluated in the European research project FIONA (Framework for Indoor and Outdoor Navigation Assistance) [Fiona]. FIONA established a project-internal ecosystem with 29 software components that were composed to form 18 different project demonstrators. Thanks to the service definitions and several component alternatives, the ecosystem enables the composition of 960 demonstrator variants. The integration workshops in FIONA and the user study conducted for this thesis showed that the approach enables ecosystem collaboration with separation of roles and reduces the effort required to compose new applications to just a few hours.

8.1 Applicability and Relevance

The approach is implemented in the SmartMDSD Toolchain. The toolchain and the proposed workflow were used in practice in several research activities to develop and compose many applications in the domain of service robotics, intralogistics, and also in domains unrelated to service robotics—that is, in personal navigation for indoor and outdoor environments. This demonstrates the practicability of system composition and demonstrates the contributions of the thesis: The demonstrators show that the approach is suitable and that it has been used to build a number of systems. The range of applications shows that the approach is neither limited to a single application nor to a single domain. This thesis contributed to:

- system composition in general
- applying Model-Driven Software Development (MDSD) in robotics
- organization of a robotics software ecosystem using three composition tiers
- structures for robotics to enhance the separation of roles and handover as required for ecosystem collaboration
- raising the composability of building blocks and supporting component suppliers to provide them for reuse by others

- the effective composition of new applications from existing building blocks, thus developing new innovation potential through combination, especially for Small and Medium-Sized Enterprises (SMEs)

The contributions listed above are in line with the current challenges and needs of robotics as identified by the European SPARC Robotics initiative within the *Strategic Research Agenda (SRA)* [euR13] and the *Multi-Annual Roadmap (MAR)* [euR16]. Furthermore, the contributions are in line with the current challenges in software engineering in general—for example, as identified by the *ITEA Roadmap for Software-Intensive Systems and Services* [ITE09]. This shows the importance and relevance of the topics to which this thesis contributes.

The overall motivation of this thesis has been and is being addressed in recent (e.g. FIONA [Fiona] and ReApp [ReApp]) and ongoing (RobMoSys EU Horizon 2020 [RobMoSys] until December 2020 and German BMWi PAiCE SeRoNet [Bun17] until February 2021) initiatives. This thesis has contributed to the vision of the *RobMoSys* project, which is now funded by the *European Horizon 2020 research and innovation programme.* Its funding on a European level underlines the relevance of a model-driven and composition-oriented approach that is addressed by this thesis. RobMoSys is a part of the effort towards establishing an *EU Digital Industrial Platform for Robotics* with a focus on sound modeling structures. The service-based composition approach and ecosystem organization in three composition tiers, as presented in this thesis, is currently being adopted by the RobMoSys project. The *SeRoNet* project, whose major focus is on establishing a brokerage and market platform, addresses similar goals as RobMoSys but on a German national level. It too was influenced by this thesis during preparation. Thus, the topics addressed in this thesis and the contributions made by this thesis are considered a relevant contribution to systematic engineering of software for service robots.

8.2 Future Work

The approach contributed to the systematic engineering of software for robots. The presented structures provide a basis for promising future work.

The modeling in this thesis addressed an important but narrow area of robotics applications. Applying modeling to other areas of the system is necessary—for example, in safety validation, in certification, in managing qualities, and in managing additional non-functional properties. Linking the models of these areas with the models proposed in this approach is promising to enable a consistent interplay between these models. For example, other models might inject values for the so far fixed values in service properties. The accuracy of a laser scan might be retrieved from the hardware model or data sheet, from the component instance parameterization, or from other service properties to make them depend on input data and on chains of input data. At the same time, information expressed in service properties might be propagated to other models. For example, accessing them from the component implementation would enable the alignment of certain algorithmic settings. Accessing them from action plots (task plots for task sequencing) would allow reacting to them at run-time.

Another promising extension of design-time usage involves extending the concept of expressing offers and needs to other models, and even other areas of system composition. This would further improve the separation of roles, the composability, and the selection process in an ecosystem. For example, to select and compose existing action plots or composite components.

Since service definitions express compatible service endpoints, they can be used to identify mappings to different service definitions in order to make them compatible, thus further increasing the composability of these components. This would require defining a mapping and transformation between the service definitions and automating this through code-generation. Ontologies can help to structure the service definitions. So far, connecting two incompatible services requires an additional (manually implemented) gateway component (an adapter) that covers this transformation. It is plausible to transform one data structure into another, assuming that all the information is available to do so. More research is required with respect to the communication and service semantics.

Introducing run-time usage of the approach would allow using the information explicated in the models beyond design-time. Design-time focuses on building a consistent application where, for example, service properties provide information about qualities in principle. Robots, however, work in highly dynamic environments and many properties cannot be foreseen at design-time until the current situation of the robot is known at run-time. Using the information modeled in service definitions at run-time would ensure that the system adheres to the explicated qualities or semantics—or notice that they are not met. Using design-time information at run-time would allow for more robust applications and may even contribute to using resources adequately. For example, algorithms might no longer run at best effort but only with the required effort to match the needed quality. Monitoring at run-time, for example, is an important issue. Information from service definitions can be used to feed monitoring approaches in order to observe them. Active approaches, such as active perception or active localization, might use the information expressed in service definitions in order to actively acquire the expressed needs at run-time. Components can adjust their usual maximum performance to an adequate performance based on what is needed and expressed in the service definitions in order to save resources.

Glossary

Domain-Specific Language (DSL)

A “modeling language dedicated to a particular problem domain that offers specific notations and abstractions, which, at the same time, decrease the coding complexity and increase programmer productivity within that domain” [SSS16].

DSLs provide a way of modeling that is tailored to and adequate to the role or problem in focus. They can be textual or graphical.

application

See: robot application

architecture (software architecture)

“The structure or structures of the system, which comprise software elements, the externally visible properties of those elements, and the relationships among them.” [BCK12]

collaboration (ecosystem collaboration)

“Collaboration” of participants in an ecosystem refers to complementing each other and sharing independent and self-contained development artifacts.

Collaboration is not meant in the sense of close collaboration as in working in a team or collaborative editing of documents. *See also:* ecosystem

communication object

A self-contained entity to hold and access information that is being exchanged between components in SmartSoft.

Communication objects are C++-like objects that define the data structure and implement framework internal access methods and optional user methods (getter and setter) for convenient access. *See also:* service

communication pattern

A set of few but sufficient means that define the way in which components exchange information over services in the SmartSoft framework.

SmartSoft provides communication patterns for the sake of composability, for example one-way “send”, two-way “request-response”, and publish/subscribe mechanisms on a timely basis or based on availability of new data. *See also:* service

component (software component)

The unit of composition and unit of exchange in the ecosystem that provides functionality to the system through formally defined services at a certain level of abstraction (cf. Szyperski [Szy02]).

A software component is defined through a component model. A component can realize one or more services and interacts with others through services only. When speaking of components, this thesis refers to explicit software components as in the SmartSoft World: a service-oriented component. In contrast to that, the general term “component” is a synonym for an arbitrary piece or element of something.

component alternative

A building block (software component) that satisfies the same needs as another one.

For example, a histogram-based object recognition and a feature-based object recognition building block can be alternatives if they both provide a service for object recognition. Alternatives might differ in quality, implementation standards, maturity level, performance, or functionality (diversity of performance). Composition requires to choose from alternatives according to the needs of the application. *See also:* component (software component)

composability

The ability to flexibly combine and recombine building blocks (software components) [PW03] “as-is” into different systems for different purposes in a meaningful way. It comes with composability as the property of parts that makes them become “building blocks”.

It is the basic prerequisite for system composition. Composability has aspects both between components and between application and components. It comprises syntactic and semantic aspects. The thesis improves composability of software components by explicating (modeling) syntactic and semantic information, that otherwise remains hidden within building blocks. The approach manages this information through the composition workflow. *See also:* system composition, system integration

composition

See: system composition

composition workflow

The order and connection of steps, the involved roles, and necessary models to build new applications from existing building blocks.

Usually, “process” refers to “workflow” on the conceptual level; “workflow” refers to a more concrete and technical level, for example as realized in a tool. However, as the workflow of the SmartMDSD Toolchain is very close to the conceptual process, the terms are used as synonyms.

diversity of performance

A distinguishing property of a component alternative.

For example: quality, implementation standard, performance, or maturity level. *See also:* component alternative

ecosystem

A collaboration model (cf. [BB10; IL04]), which describes the many ways and advantages in which stakeholders (e.g. experts in various fields or companies) network, collaborate, share efforts and costs around a domain or product.

This thesis envisions a robotics software business ecosystem: Robotics is a diverse and interdisciplinary field, and contributors will have dedicated experience and can contribute software building blocks using their expertise for use by others. Participants in an ecosystem do not necessarily know each other. The challenge is to organize the contributions without bilateral negotiation of agreements between participants: An ecosystem approach must enable collaboration by structure rather than collaboration by management. Ecosystem and software business ecosystem are used as synonyms. *See also:* collaboration (ecosystem collaboration)

freedom from choice (versus freedom *of* choice)

Freedom *from* choice is a principle that positively limits the number of available options to provide guidance via selected structures, thereby removing unnecessary degrees of freedom.

Freedom *from* choice supports separation of roles and is a mandatory prerequisite for system composition. Selecting the appropriate structures comes with a high responsibility to identify where guidance is needed and how it can be accomplished without limiting the use of the structures for system design. In contrast, freedom *of* choice does not limit the design space, thus, leaving the whole variety of options open for use (one problem can be solved in numerous ways); gaining flexibility at the cost of guidance, composability, and system level conformance.

integration

See: system integration

interface

The “boundary across which two independent entities meet and interact or communicate with each other” [Bac+02].

"Interface" is used as a very broad term, not to be confused with the narrow interpretation of "interface" as in application programming interface (API). The interface between components is realized on a service level. *See also:* service

parameter, parameter set

See: variation point

participant (in an ecosystem)

Synonym for stakeholder. *See also:* role

robot

Synonym for service robot. *See also:* service robot

robot application

Application as in "software application": The software for the service robot that is being developed.

Building blocks (software components) are put together (composed) to applications. Robot application is used as a synonym for the "system" that is being developed. Systems can be used as a building block for other systems.

role

A certain task or activity with associated responsibilities that someone (individual, group, or organization) covers in the composition workflow, for example the component developer role or the system compositor role.

Someone that covers a particular role typically is an expert in a particular field (e.g. object recognition). A role takes a particular perspective or view on the overall workflow or application. It is associated with certain tasks, duties, rights, and permissions which do not overlap with other roles. All participants of the ecosystem cover a particular role. *See also:* stakeholder, separation of roles

separation of concerns

A principle in computer science and software engineering: It identifies and decouples different problem areas into distinct parts such that one can look at them and solve them independently resulting in a lower complexity.

Separation of concerns is the basis for separation of roles. It is a necessary prerequisite for system composition in a robotics business ecosystem. *See also:* separation of roles

separation of roles

A principle that enables and supports different groups of stakeholders in playing their role in an overall composition workflow without being required to become an expert in every field (in what other roles cover).

Separation of roles is closely related to separation of concerns. It is a necessary prerequisite for system composition in a robotics business ecosystem. *See also:* role, separation of concerns

service

A system-level entity that shapes the architecture and via which components exchange information at a proper level of abstraction.

Services follow a service contract. A service separates the internal and external view of a component. They describe the functional boundary between components. Services are described via service definitions. A service in the system consists of at least two service endpoints associated with components (providing part / requiring part). Components create a service endpoint by instantiating a service definition. *See also:* service endpoint, service definition

service definition

The common structure for a class of services in a reusable and formal description to ensure that components offering or using such a service can be used together; a service definition can be considered the "type definition" of a service, i.e. the formal definition of a service on composition Tier 2.

The service definition is the main element in the composition structure. Component models instantiate service definitions to create a service endpoint. Service definitions explicate information that is important for composition: data structure, communication semantics and service properties. Service definitions are reusable elements in the composition workflow and are considered the entities that enable separation of roles, e.g. to decouple component development from system composition. Service definitions describe the service as a whole and do not distinguish between service endpoints (providing part or requiring part). *See also:* service, service endpoint

service endpoint

The part of the component that realizes the service: the only access point of a component, i.e. the entry point of a component to communicate with others through a service.

When modeling a component, the component developer models a service endpoint by instantiating a service definition (the "type" of the service endpoint). The component developer also decides whether the endpoint is the requiring part or the providing part of the service ("source" or "sink" with respect to the flow of information via the service). The graphical notation corresponds to the UML port. *See also:* service, service definition

service property

A reusable name–value pair to explicate the semantics of a service on the application-level that is otherwise hidden within an implementation or only included in documentation.

Service properties contribute to building a domain-specific vocabulary on composition Tier 2. They are used by service definitions and improve the composability of software components and component selection. Component developers assign values to service properties to express what their service provides or requires. Using constraints, the property's value is used to support component selection and to validate the composition.

service robot

An autonomous system that performs useful tasks for humans [Int12] in a shared environment, e.g. co-workers at work, household assistants, delivery robots and farming robots.

"Robotics" and "robot" in this thesis generally refer to "service robotics" and "service robot".

service wish list

A collection of services to express the needs of an application.

An entry of the wish list is referred to as a "service wish" or "wish". It instantiates a service definition and refines the service properties to express the needs towards that service. The wish list is input and basis to assist in selecting suitable components from a market or repository of components. The wish list serves as input to verify the composed system.

skill (of a robot)

A basic capability or function of a robot.

A collection of skills is required for the robot to do a certain task. For example, a butler robot requires skills for localization, object recognition, mobile manipulation, speech input, speech output, etc. A component often implements a certain skill, but skills might also be realized by multiple components. Following the three tier architecture [Bon+97; Fir89], this thesis uses the term skill for components that are coordinated by the sequencer. In the context of SmartSoft and its use of robotics behavior, "skill" is defined in a narrower context and refers to the link between robotics behavior tasks (or task plots) and services of components.

SmartSoft (SmartSoft World)

An umbrella term for concepts, principles, tools, and content that are developed at the Service Robotics Research Center Ulm (Service Robotics Ulm).

The SmartSoft World extends and carries on the core motivation of the "SmartSoft Framework" [SW99b]: A framework that applies a component-based approach for robotics software development. This thesis contributes to the SmartSoft World.

stakeholder

An individual, a group of individuals, or organizations that share a particular interest in the robotics business ecosystem and collaborate therein. Synonym for participant in an ecosystem.

The term "stakeholder" is a rather general term used in the context of the ecosystem and its participants (e.g. structural drivers, suppliers, and system builders). In contrast to that, a "role" is a more narrow and technical term in the context of the composition workflow. Each stakeholder covers one or several roles. *See also:* role, participant (in an ecosystem)

system composition

The activity of putting together a service robotics application from existing building blocks (here: software components) in a meaningful way, that is, flexibly combining and recombining them depending on the application's needs.

System composition puts a focus on the new whole that is created from existing parts rather than on making parts work together only by gluing them together: The whole still consists of its parts, the parts are still visible and isolated as entities and they are thus still exchangeable. They can be split again. Software components that are subject to composition shall be taken "as-is", only being adapted within modeled boundaries. They thus must be built with this intention right from the beginning. The context in which they will later be composed is unknown, which puts special requirements on their composability and the overall workflow.

System composition is about adhering to a composition structure. It is about putting in effort once to comply with the composition structure and gain immediate composability with other building blocks. In contrast, integration is about building individual adapters between mutual parts or even modifying the parts themselves. High effort comes through the high number of combinations between building blocks that each require a new adapter each time they are integrated.

See also: system integration

system configuration

A step in the workflow and according (meta-)model to wire (connect), parameterize, and assemble component instances.

system integration

The activity and effort of combining components, requiring modification or additional action to make them work with others. System integration is the opposite of system composition.

A distinction between integration and composition can be drawn by the effort [PW03]: The ability to readily combine and recombine composable components distinguishes them from integrated components. The latter are modified with high effort to make them work with others, essentially by writing adapters. The integrated part amalgamates into the whole (i.e. becomes one part; mixes, as red and green water will mix), thus making it hard to be removed or exchanged—and if they are removed, it requires new adapters. In software engineering in general, "integration" is the process or step of combining or assembling components into an overall/whole system [ISO15; ISO10].

See also: system composition

variation point

An explicated variable of a component that is subject to modification from outside of the component for parameterization/configuration.

System composition aims at using components "as-is" but components might need configuration via variation points to match the current application or context. Modification of a component, e.g. modifying an open-source component, contrasts with that. Using a variation point is a modification that is intended by the component supplier. It does not break the component in any way. Modifying the source code for the purpose of configuration is not intended by the component supplier and may trigger side effects that are unwanted and unknown.

verification

The check whether the system composed from components meets the expressed application's needs.

Verification is the "test of a system to prove that it meets all its specified requirements at a particular stage of its development" [ISO15]. Boehm [Boe79] considers verification as the correspondence between the software and its specification. In context of this thesis, this means to check the "fulfillment" of all wishes in the wish list.

wish list

See: service wish list

workflow

See: composition workflow

Abbreviations

List of abbreviations that are used in more than one paragraph.

API	Application Programming Interface
BCM	BRICS Component Model
CBSE	Component-Based Software Engineering
DSL	Domain-Specific Language
DSPL	Dynamic Software Product Line
GUI	Graphical User Interface
IDE	Integrated Development Environment
IDL	Interface Definition Language
MDSD	Model-Driven Software Development
OCL	Object Constraint Language
OMG	Object Management Group
OPC-UA	OPC Unified Architecture
OWL	Web Ontology Language
OWL-S	Web Ontology Language for Web Services
ROS	Robot Operating System
SDL	Service Definition Language
SLAM	Simultaneous Localization and Mapping
SME	Small and Medium-Sized Enterprise
SOA	Service-Oriented Architecture
SoaML	Service Oriented Architecture Modeling Language

SPL	Software Product Line
SysML	Systems Modeling Language
UML	Unified Modeling Language
WSDL	Web Services Description Language

List of Figures

1.1 Software development in service robotics resembles crafting 2
1.2 The relation of changes to development efforts 3
1.3 The step change from integration to composition 4
1.4 A robotics business ecosystem and its participants 5
1.5 The need and the benefit for system composition 6
1.6 Research questions 7
1.7 The organization of an ecosystem in three composition tiers 10
1.8 Thesis outline 12

2.1 The eclipse ecosystem: collaboration and competition 21
2.2 Component communicate through services 42
2.3 The 3T architecture in robotics 45

3.1 The need for structures for system composition in an ecosystem 47
3.2 A robotics business ecosystem and its participants 53
3.3 Use-cases: developing and maintaining a service robot application 54
3.4 Use-cases: the ecosystem perspective 56
3.5 Collaboration by management vs. collaboration by structure 58
3.6 Superordinate structures for system composition 58
3.7 The granularity of components 60
3.8 The consequences on system composition 61
3.9 Occurrence of composability 62
3.10 The composition workflow 65
3.11 Stakeholders collaborating and interacting in an ecosystem 66
3.12 Horizontal, vertical, and lateral separation of roles 69
3.13 Support via tooling for ecosystem participants 71
3.14 Overview of the consequences on the approach 75
3.15 Overview of the consequences on the approach refined: considerations 78
3.16 The SmartSoft framework in relation to this thesis 80

4.1 The three tiers for system composition in an ecosystem 92

4.2 Consequences and roles in relation to the composition tiers 94
4.3 The approach for system composition based on service definitions 95
4.4 Service definitions establish the link between suppliers and users 97
4.5 The meta-structure for system composition 98
4.6 Influence of considerations to elements of the composition structure 99
4.7 The roles of the workflow and their interaction with the composition structure 100
4.8 Components provide or require services as defined in service definitions 101
4.9 The workflow view on the approach 105
4.10 Applying the approach in a local, non-ecosystem perspective 106

5.1 The structure of chapter 5 109
5.2 An overview on the meta-models of the composition structure in Tier 1 111
5.3 The service definition meta-model 112
5.4 The communication object meta-model 116
5.5 Service properties along the composition workflow 118
5.6 The meta-model for service properties 119
5.7 Mapping of constraints in service properties to evaluation methods 121
5.8 The parameter set meta-model 123
5.9 Illustration of parameter sets 124
5.10 Documentation in the meta-models 124
5.11 The component meta-model 126
5.12 Service endpoints: the service level 127
5.13 Service endpoints: the technical level 128
5.14 The parameter meta-model 131
5.15 Component-internal parameters in comparison to parameter sets 133
5.16 The component documentation meta-model 134
5.17 Generation and source of documentation 135
5.18 Service wish list, market, and system configuration 136
5.19 The service wish list meta-model 137
5.20 The component selection process 138
5.21 The matchmaking process 139
5.22 Wish list realization explained 142
5.23 The service realization meta-model 143
5.24 The system configuration meta-model 144
5.25 The instance association problem 145
5.26 The composition parameter meta-model 146
5.27 Verification of the composition along composition axes 147
5.28 Verification of the composition with respect to services 147
5.29 The deployment meta-model 148

6.1 The consequences on the approach: integrated tooling (C7) 152
6.2 Realization variants: UML profiling vs. direct realization 153
6.3 SmartMDSD Toolchain implementation technologies 154

6.4 Workflow of the SmartMDSD Toolchain 156
6.5 The modeling perspective and the model relations in the SmartMDSD Toolchain 157
6.6 Code generation: applying the generation-gap pattern 159
6.7 Artifacts in the composition workflow 160
6.8 Use-Case: navigation stack 161
6.9 Toolchain introduction: example scenario 163
6.10 Using the communication object DSL 164
6.11 Using the service property DSL 165
6.12 Modeling and implementing custom evaluation functions 165
6.13 Using the service definition DSL 166
6.14 Using the parameter DSL 167
6.15 Modeling a component 168
6.16 Using the component service DSL in the SmartCdlServer 169
6.17 Using the component service DSL in the SmartPioneerBaseServer 169
6.18 Using the component parameter DSL 170
6.19 Implementing a component 171
6.20 Using the documentation DSL 172
6.21 Component selection in the SmartMDSD Toolchain 173
6.22 A technical view on component retrieval 174
6.23 The system configuration model 176
6.24 Editing the start-up infrastructure 177
6.25 Editing the service wish list 177
6.26 Deployment modeling 178

7.1 Fields of application of the approach and the SmartMDSD Toolchain 182
7.2 The service robots Kate and Larry 184
7.3 Excerpts of the (collaborative) butler scenario 185
7.4 System configuration view: migration from Kate to Larry 187
7.5 Flexible combination of object recognition by composition 189
7.6 The Personal Mobile Navigation Demonstrator: handheld device and GUI . . . 191
7.7 The Personal Mobile Navigation Demonstrator: system configuration view/model 192
7.8 The Visual Localization Demonstrator for personal navigation in an office space 193
7.9 The Visual Localization Demonstrator: system configuration view/model 194
7.10 The Visual Localization Demonstrator: deployment view/model 194
7.11 Industry 4.0 Robot Commissioning Fleet in Intra-Logistics 195
7.12 The iserveU hospital transports robot 196
7.13 The system configuration model of iserveU hospital transports robot 197
7.14 An excerpt of the user study 202
7.15 The Visual Localization Demonstrator: composition effort 205
7.16 Efforts in the FIONA integration workshop 206

List of Tables

5.1 The severity of service properties 130
5.2 Mapping of service endpoints to SmartSoft communication patterns 131

6.1 DSLs, models, and implementation technology of the SmartMDSD Toolchain . 158

7.1 Components in Kate and Larry 186
7.2 Demonstrator variants in the FIONA research project 203

References

[Ame11] Editors of the American Heritage Dictionaries. *The American Heritage Dictionary of the English Language*. Vol. 5. Houghton Mifflin Harcourt, Nov. 2011. ISBN: 9780547041018.

[Ams12] Jim Amsden. *Using SoaML services architecture*. IBM developer Works. Apr. 2012.
URL: https://www.ibm.com/developerworks/rational/library/soaml-services-architecture/index.html (visited: Apr. 1, 2017).

[And+05] Noriaki Ando, Takashi Suehiro, Kosei Kitagaki, Tetsuo Kotoku, and Woo-Keun Yoon. "RT-Middleware: Distributed Component Middleware for RT (Robot Technology)". In: *Proc. IEEE/RSJ International Conference on Intelligent Robots and Systems 2005 (IROS'05)*. Edmonton, Alta., Canada, Aug. 2005, pp. 3933–3938.
DOI: 10.1109/IROS.2005.1545521.

[Autosar] AUTOSAR development cooperation. *AUTOSAR Website*.
URL: http://www.autosar.org (visited: Apr. 20, 2017).

[Awa+16] Ramez Awad, Georg Heppner, Arne Roennau, and Mirko Bordignon. "ROS Engineering Workbench based on semantically enriched App Models for improved Reusability". In: *IEEE 21st International Conference on Emerging Technologies and Factory Automation 2016 (ETFA)*. Berlin, Germany, Sept. 2016. ISBN: 978-1-5090-1314-2.
DOI: 10.1109/ETFA.2016.7733581.

[AZ06] Uwe Aßmann and Steffen Zschaler. "Ontologies, Meta-models, and the Model-Driven Paradigm". In: *Ontologies for Software Engineering and Software Technology*. Ed. by Coral Calero, Francisco Ruiz, and Mario Piattini. 1st ed. Springer-Verlag Berlin Heidelberg, 2006. Chap. 9, pp. 249–273. ISBN: 978-3-540-34517-6.
DOI: 10.1007/3-540-34518-3.

[BA99] Barry Boehm and Chris Abts. "COTS Integration: Plug and Pray?" In: *Computer* 32.1 (Jan. 1999), pp. 135–138. ISSN: 0018-9162.
DOI: 10.1109/2.738311.

[Bac+02] Felix Bachmann, Len Bass, Paul Clements, David Garlan, James Ivers, Reed Little, Robert Nord, and Judith Stafford. *Documenting Software Architecture: Documenting Interfaces*. Technical Note CMU/SEI-2002-TN-015. Pittsburgh, PA: Carnegie Mellon Software Engineering Institute, June 2002.

[Bar16] John S. Baras. *The Next Wonder - MBSE/MBE: From Ideas to "Making Products and Services"*. 21st IEEE Conference on Emerging Technologies and Factory Automation (ETFA). Keynote Talk. Berlin, Germany, Sept. 2016.

[Bas+14] Ana Sasa Bastinos, Peter Haase, Georg Heppner, Stefan Zander, and Nadia Ahmed. "ReApp Store – a semantic AppStore for applications in the robotics domain". In: *Proceedings of the Industry Track at the International Semantic Web Conference 2014, co-located with the 13th International Semantic Web Conference (ISWC 2014)*. Riva del Garda, Italy, Oct. 2014.
URL: http://ceur-ws.org/Vol-1383/.

[Bau+13] Johannes Baumgartl, Thomas Buchmann, Dominik Henrich, and Bernhard Westfechtel. "Towards Easy Robot Programming: Using DSLs, Code Generators and Software Product Lines". In: *Proceedings of the 8th International Conference on Software Paradigm Trends (ICSOFT-PT'13)*. 2013, pp. 147–157.

[BB10] Jan Bosch and Petra Bosch-Sijtsema. "From integration to composition: On the impact of software product lines, global development and ecosystems". In: *Journal of Systems and Software* 83.1 (June 2010). SI: Top Scholars, pp. 67–76. ISSN: 0164-1212.
DOI: 10.1016/j.jss.2009.06.051.
URL: http://www.sciencedirect.com/science/article/pii/S0164121209001617.

[BBM96] Victor R. Basili, Lionel C. Briand, and Walcélio L. Melo. "A Validation of Object-Oriented Design Metrics as Quality Indicators". In: *IEEE Transactions on Software Engineering* 22.10 (Oct. 1996), pp. 751–761. ISSN: 00985589.
DOI: 10.1109/32.544352.

[BCK12] Len Bass, Paul Clements, and Rick Kazman. *Software Architecture in Practice*. 3rd ed. SEI Series in Software Engineering. Addison Wesley, 2012. ISBN: 978-0-321-81573-6.

[BCW12] Marco Brambilla, Jordi Cabot, and Manuel Wimmer. *Model-Driven Software Engineering in Practice*. 1st ed. Synthesis Lectures on Software Engineering. Morgan & Claypool Publishers, 2012. ISBN: 978-1-60845-882-0.
DOI: 10.2200/S00441ED1V01Y201208SWE001.

[BHA12] Nelly Bencomo, Svein Hallsteinsen, and Eduardo Santana de Almeida. "A View of the Dynamic Software Product Line Landscape". In: *Computer* 45.10 (Oct. 2012). ISSN: 0018-9162, pp. 36–41. ISSN: 0018-9162.
DOI: 10.1109/MC.2012.292.

[Bla] Jose-Luis Blanco-Claraco. *Mobile Robot Programming Toolkit (MRPT).* URL: http://www.mrpt.org/ (visited: Mar. 1, 2017).

[Boe79] Barry Boehm. "Guidelines for Verifying and Validating Software Requirements and Design Specifications". In: *Proceedings of the European Conference on Applied Information Technology of the International Federation for Information Processing.* London: North Holland, Sept. 1979, pp. 711–719. URL: http://csse.usc.edu/TECHRPTS/1979/1979_main.html.

[Boh+11] Jonathan Bohren, Radu Bogdan Rusu, E. Gil Jones, Eitan Marder-Eppstein, Caroline Pantofaru, Melonee Wise, Lorenz Mösenlechner, Wim Meeussen, and Stefan Holzer. "Towards Autonomous Robotic Butlers: Lessons Learned with the PR2". In: *Proc. 2011 IEEE International Conference on Robotics and Automation (ICRA'11).* Shanghai, China, May 2011, pp. 5568–5575. DOI: 10.1109/ICRA.2011.5980058.

[Bon+16] Stéphane Bonnet, Jean-Luc Voirin, Daniel Exertier, and Véronique Normand. "Not (strictly) relying on SysML for MBSE: Language, tooling and development perspectives: The Arcadia/Capella rationale". In: *2016 Annual IEEE Systems Conference (SysCon).* Orlando, FL, USA, Apr. 2016. DOI: 10.1109/SYSCON.2016.7490559.

[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), pp. 237–256. DOI: 10.1080/095281397147103.

[Bos09] Jan Bosch. "From Software Product Lines to Software Ecosystems". In: *Proceedings of the 13th International Software Product Line Conference (SPLC '09).* SPLC '09. San Francisco, California, USA: Carnegie Mellon University, 2009, pp. 111–119. URL: http://dl.acm.org/citation.cfm?id=1753235.1753251.

[BRIa] BRICS Consortium. *BRIDE: the BRIcs Development Environment.* URL: http://www.best-of-robotics.org/bride (visited: May 2, 2017).

[BRIb] BRICS Consortium. *BROCRE Website.* URL: http://www.best-of-robotics.org/brocre/ (visited: May 1, 2017).

[Bro+07] Alex Brooks, Tobias Kaupp, Alexei Makarenko, Stefan Williams, and Anders Orebäck. "Orca: A Component Model and Repository". In: *Software Engineering for Experimental Robotics.* Ed. by Davide Brugali. Vol. 30. Springer Tracts in Advanced Robotics. Springer Berlin Heidelberg, 2007, pp. 231–251. ISBN: 978-3-540-68951-5. DOI: 10.1007/978-3-540-68951-5_13. URL: http://www.cas.edu.au/content.php/237.html?publicationid=340.

[Bro+98] Manfred Broy, Anton Deimel, Juergen Henn, Kai Koskimies, František Plášil, Gustav Pomberger, Wolfgang Pree, Michael Stal, and Clemens Szyperski. “What characterizes a (software) component?” In: *Software - Concepts & Tools* 19.1 (1998), pp. 49–56. ISSN: 1432-2188.
DOI: 10.1007/s003780050007.

[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: *Proceedings of the 28th Annual ACM Symposium on Applied Computing*. SAC ’13. Coimbra, Portugal: ACM, 2013, pp. 1758–1764. ISBN: 978-1-4503-1656-9.
DOI: 10.1145/2480362.2480693.

[Bru+14] Davide Brugali, Andrea Fernandes da Fonseca, Andrea Luzzana, and Yamuna Maccarana. “Developing Service Oriented Robot Control System”. In: *2014 IEEE 8th International Symposium on Service Oriented System Engineering*. Oxford, UK, Apr. 2014, pp. 237–242.
DOI: 10.1109/SOSE.2014.28.

[Bru15] Davide Brugali. “Model-Driven Software Engineering in Robotics”. In: *IEEE Robotics & Automation Magazine* 22.3 (Sept. 2015), pp. 155–166.
DOI: 10.1109/MRA.2015.2452201.

[BS02] Manfred Broy and Johannes Siedersleben. “Objektorientierte Programmierung und Softwareentwicklung: Eine kritische Einschätzung”. In: *Informatik Spektrum* 25.1 (Feb. 2002), pp. 3–11.

[BS09] Davide Brugali and Patrizia Scandurra. “Component-Based Robotic Engineering (Part I)”. In: *IEEE Robotics Automation Magazine* 16.4 (Dec. 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 (Mar. 2010), pp. 100–112. ISSN: 1070-9932.
DOI: 10.1109/MRA.2010.935798.

[BTR05] David Benavides, Pablo Trinidad, and Antonio Ruiz-Cortés. “Automated Reasoning on Feature Models”. In: *Proceedings of Advanced Information Systems Engineering: 17th International Conference, CAiSE 2005, Porto, Portugal, June 13-17, 2005*. Springer Berlin Heidelberg, 2005, pp. 491–503. ISBN: 978-3-540-32127-9.
DOI: 10.1007/11431855_34.

[Bun] Bundesministerium für Bildung und Forschung. *LogiRob: Multi-Robot-Transportsystem im mit Menschen geteilten Arbeitsraum.*

URL: http://www.bmbf-softwareforschung.de/media/content/Infoblatt_LogiRob.pdf (visited: Aug. 7, 2017).

[Bun17] Bundesministerium für Wirtschaft und Energie (BMWi). *SeRoNet — Eine Plattform zur arbeitsteiligen Entwicklung von Serviceroboter-Lösungen*. 2017. URL: http://www.digitale-technologien.de/DT/Redaktion/DE/Standardartikel/PAICEProjekte/paice-projekt_seronet.html (visited: May 5, 2017).

[Bür+16] Mathias Bürger, Irene Süßemilch, **Dennis Stampfer**, Christian Schlegel, Florian Schreiner, Stefan Rueping, Christopher Zimmer, and Ali Golestani. *ITEA 2 Projekt: Framework for Indoor and Outdoor Navigation Assistance (FIONA): Abschlussbericht*. 2016. DOI: 10.2314/GBV:880711841.

[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 (May 2014), pp. 3–23. ISSN: 0164-1212. DOI: 10.1016/j.jss.2013.12.038.

[CodeCloud] *Code Your Cloud: A free, open-source code editor for Google Drive and oneDrive*. URL: https://codeyourcloud.com (visited: Aug. 1, 2017).

[Cou+10] Steve Cousins, Brian Gerkey, Ken Conley, and Willow Garage. "Sharing Software with ROS [ROS Topics]". In: *IEEE Robotics Automation Magazine* 17.2 (June 2010), pp. 12–14. ISSN: 1070-9932. DOI: 10.1109/MRA.2010.936956.

[CSS11] Ivica Crnkovic, Judith Stafford, and Clemens Szyperski. "Software Components beyond Programming: From Routines to Services". In: *IEEE Software* 28.3 (May 2011), pp. 22–26. ISSN: 0740-7459. DOI: 10.1109/MS.2011.62.

[Del+08] Didier Delanote, Stefan Van Baelen, Wouter Joosen, and Yolande Berbers. "Using AADL to Model a Protocol Stack". In: *13th IEEE International Conference on Engineering of Complex Computer Systems (ICECCS'08)*. Belfast, Northern Ireland, Mar. 2008, pp. 277–281. DOI: 10.1109/ICECCS.2008.12.

[Del14] Ian Delaney. *Why HERE mapping cars are basically cyborgs*. Mar. 2014. URL: http://360.here.com/2014/03/18/open-source-robotics-foundation/ (visited: Aug. 1, 2016).

[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: *Proc. Third international conference on Simulation, Modeling, and Programming for Autonomous Robots (SIMPAR'12), Nov 2012, Tsukuba, Japan*. Vol. 7628. Lecture Notes in Computer Science. <hal-00995125>. Springer-

Verlag, 2012, pp. 149–160. ISBN: 978-3-642-34327-8.
DOI: 10.1007/978-3-642-34327-8_16.
URL: https://hal.archives-ouvertes.fr/hal-00995125.

[Dia10] Rosen Diankov. "Automated Construction of Robotic Manipulation Programs". CMU-RI-TR-10-29. PhD thesis. Carnegie Mellon University, Robotics Institute, Aug. 2010.
URL: http://www.programmingvision.com/rosen_diankov_thesis.pdf.

[Die02] Andreas Dietzsch. *Systematische Wiederverwendung in der Software-Entwicklung*. Deutscher Universitätsverlag, 2002. ISBN: 978-3-663-11580-9.
DOI: 10.1007/978-3-663-11580-9.

[Dij82] Edsger W. Dijkstra. "On the Role of Scientific Thought". In: *Selected Writings on Computing: A personal Perspective*. Monographs in Computer Science. ISBN: 0-387-90652-5. Springer, 1982, pp. 60–66.
URL: http://www.cs.utexas.edu/users/EWD/ewd04xx/EWD447.PDF.

[Dör13] Stephan Dörner. "Kombination ist die neue Innovation". In: *The Wall Street Journal* (2013).
URL: http://blogs.wsj.de/wsj-tech/2013/12/11/innovation/ (visited: Mar. 22, 2016).

[DSLRob] *Workshop on Domain-Specific Languages and Models for Robotic Systems (DSLRob)*.
URL: http://www.doesnotunderstand.org/public/DSLRob (visited: Feb. 21, 2017).

[Ecla] Eclipse Foundation. *Eclipse CDO Model Repository*.
URL: http://www.eclipse.org/cdo/ (visited: Sept. 1, 2017).

[Eclb] Eclipse Foundation. *Eclipse Che: Developer workspace server and cloud IDE*.
URL: http://www.eclipse.org/che/ (visited: Aug. 1, 2017).

[Eclc] Eclipse Foundation. *Eclipse Modeling Project Website*.
URL: http://eclipse.org/modeling (visited: Jan. 25, 2016).

[Ecld] Eclipse Foundation. *Eclipse Wiki: Xbase*.
URL: https://wiki.eclipse.org/Xbase (visited: Nov. 8, 2016).

[Ecle] Eclipse Foundation. *EcoreTools – Graphical Modeling for Ecore*.
URL: http://www.eclipse.org/ecoretools/ (visited: Dec. 27, 2016).

[Eclf] Eclipse Foundation. *Research @ Eclipse Foundation*.
URL: http://www.eclipse.org/org/research/ (visited: June 1, 2017).

[Eclg] Eclipse Foundation. *Sirius Website*.
URL: https://eclipse.org/sirius/ (visited: May 2, 2017).

[Eclh] Eclipse Foundation. *Xtend Website*.
URL: http://www.eclipse.org/xtend (visited: Jan. 25, 2016).

[Ecli] Eclipse Foundation. *Xtext Website.* URL: http://www.eclipse.org/Xtext/ (visited: Jan. 25, 2016).

[ER03] Albert Endres and Dieter Rombach. *A Handbook of Software and Systems Engineering: Empirical Observations, Laws and Theories.* Addison-Wesley, 2003. ISBN: 032-1-154207.

[Erl08] Thomas Erl. *SOA: Principles of Service Design.* Prentice Hall, 2008. ISBN: 0-13-234482-3.

[ES12] Ayssam Elkady and Tarek Sobh. "Robotics Middleware: A Comprehensive Literature Survey and Attribute-Based Bibliography". In: *Journal of Robotics* 2012 (2012). Article ID 959013, pp. 1–15. DOI: 10.1155/2012/959013.

[euR] euRobotics aisbl. *Topic Group on Software Engineering, System Integration, Systems Engineering.* URL: http://www.i6.in.tum.de/Main/TG-Software-Systems-Engineering (visited: Mar. 1, 2017).

[euR13] euRobotics aisbl. *Strategic Research Agenda for Robotics in Europe 2014–2020 (SRA).* 2013.

[euR16] euRobotics aisbl. *Robotics 2020 Multi-Annual Roadmap (MAR).* Release B. Dec. 2016.

[FGH06] Peter H. Feiler, David P. Gluch, and John J. Hudak. *The Architecture Analysis & Design Language (AADL): An Introduction.* Technical Note CMU/SEI-2006-TN-011. Pittsburgh, PA: The Software Engineering Institute (SEI), Carnegie Mellon University, 2006. URL: http://resources.sei.cmu.edu/library/asset-view.cfm?AssetID=7879.

[Fio15] *FIONA Project Review: Framework for Indoor and Outdoor Navigation Assistance.* (non-public document). Oct. 2015.

[Fiona] *FIONA - Framework for Indoor and Outdoor Navigation Assistance.* URL: http://www.fiona-project.eu (visited: Jan. 25, 2016).

[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.* Addison-Wesley, 2011. ISBN: 978-0-321-71294-3.

[Fra15] Sandra Frank. "Seamless transition between indoor and outdoor localization technologies by the example of a pedestrian navigation demonstrator". Master Thesis. Ulm University, 2015.

[Frö02] Joakim Fröberg. "Software Components and COTS in Software System Development". In: *Extended Report for Building Reliable Component-Based Systems.*

Ed. by I. Crnkovic and M. Larsson. Artech House, June 2002, pp. 59–67. ISBN: 1-58053-327-2.
URL: http://www.idt.mdh.se/cbse-book/extended-reports/15_Extended_Report.pdf.

[FS15] Sandra Frank and **Dennis Stampfer**. *Demonstrating System Integration by Composition: Seamless Indoor and Outdoor Navigation*. Video. Oct. 2015.
URL: https://www.youtube.com/watch?v=fS3PJMswlH4 (visited: Dec. 27, 2016).

[GAO95] David Garlan, Robert Allen, and Joh Ockerbloom. "Architectural mismatch: why reuse is so hard". In: *IEEE Software* 12.6 (Nov. 1995), pp. 17–26. ISSN: 0740-7459.
DOI: 10.1109/52.469757.

[GB11] Luca Gherardi and Davide Brugali. "An eclipse-based Feature Models toolchain". In: *Proc. of the 6th Workshop of the Italian Eclipse Community (Eclipse-IT 2011)*. Milano, Italy, Sept. 2011.

[GB14] Luca Gherardi and Davide Brugali. "Modeling and Reusing Robotic Software Architectures: The HyperFlex Toolchain". In: *2014 IEEE International Conference on Robotics and Automation (ICRA)*. Hong Kong, China, May 2014, pp. 6414–6420.
DOI: 10.1109/ICRA.2014.6907806.

[Ger14] Brian Gerkey. *ROS user survey: the results are in*. Apr. 2014.
URL: http://www.ros.org/news/2014/04/ros-user-survey-the-results-are-in.html (visited: Feb. 21, 2017).

[Ger15] Brian Gerkey. *Why ROS 2.0?* 2015.
URL: http://design.ros2.org/articles/why_ros2.html (visited: Mar. 20, 2017).

[Gin+16] Tobias Gindele, Bernhard Rumpe, Andreas Rausch, Martin Vossiek, Peter Gulden, Christian Schlegel, and Christian Verbeek. *Intelligente modulare Serviceroboter-Funktionalitäten im menschlichen Umfeld am Beispiel von Krankenhäusern: BMBF Verbundprojekt iserveU Abschlussbericht*. Project Report. Oct. 2016.
DOI: 10.2314/GBV:87332112X.

[Gru09] Tom Gruber. "Ontology". In: *Encyclopedia of Database Systems*. Ed. by Ling Liu and M. Tamer Özsu. Springer-Verlag, 2009.

[GVH03] Brian Gerkey, Richard Vaughan, and Andrew Howard. "The Player/Stage Project: Tools for Multi-Robot and Distributed Sensor Systems". In: *Proceedings of the International Conference on Advanced Robotics (ICAR 2003)*. Coimbra, Portugal, June 2003, pp. 317–323.

[HBK11] Martin Hägele, Nikolaus Blümlein, and Oliver Kleine. "Wirtschaftlichkeitsanalysen neuartiger Servicerobotik-Anwendungen und ihre Bedeutung für die Robotik-Entwicklung (EFFIROB)". In: *Eine Analyse der Fraunhofer-Institute IPA und ISI im Auftrag des BMBF* (2011).

[Heg+12] Timo Hegele, Siegfried Hochdorfer, Matthias Lutz, Alex Lotz, **Dennis Stampfer**, Andreas Steck, Manuel Wopfner, Richard Cubek, Tobias Fromm, Markus Schneider, and Benjamin Stähle. *The Robot Butler Scenario*. Video. Feb. 2012.
URL: https://www.youtube.com/watch?v=nUM3BUCUnpY (visited: Dec. 27, 2016).

[Hei+08] H. Heinecke, M. Rudorfer, P. Hoser, C. Ainhauser, and O. Scheickl. "Enabling of AUTOSAR system design using Eclipse-based tooling". In: *International Conference: Embedded Real Time Software*. 2008.

[HHJ08] Andreas Helferich, Georg Herzwurm, and Stefan Jesse. "Software Product Lines and Service-Oriented Architecture: A Systematic Comparison of Two Concepts". In: *Proceedings of the First Workshop on Service-Oriented Architectures and Software Product Lines*. Ed. by Sholom Cohen and Robert Krut. CMU/SEI-2008-SR-006. Software Engineering Institute. Carnegie Mellon University, May 2008, A1–A7.
URL: http://www.sei.cmu.edu/reports/08sr006.pdf.

[HKF08] Olaf Hartig, Martin Kost, and Johann-Christoph Freytag. "Component Selection with Semantic Technologies". In: *Proceedings of the 4th International Workshop on Semantic Web Enabled Software Engineering (SWESE)*. Karlsruhe, Germany, Oct. 2008.

[Hoc+13] Nico Hochgeschwender, Luca Gherardi, Azamat Shakhirmardanov, Gerhard K. Kraetzschmar, Davide Brugali, and Herman Bruyninckx. "A Model-based Approach to Software Deployment in Robotics". In: *2013 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS'13)*. Tokyo, Japan, Nov. 2013, pp. 3907–3914.
DOI: 10.1109/IROS.2013.6696915.

[HV99] Michi Henning and Steve Vinoski. *Advanced CORBA Programming with C++*. ISBN: 0-201-37927-9. Addison-Wesley Professional, 1999.

[IL04] Marco Iansiti and Roy Levien. "Strategy as Ecology". In: *Harvard Business Review* 82.3 (Mar. 2004). Reprint R0403E, pp. 68–81.

[Int12] International Federation of Robotics (IFR). *Definition of Service Robots*. Online. 2012.
URL: http://www.ifr.org/service-robots/ (visited: Aug. 2, 2016).

[ISO10] *ISO/IEC/IEEE 24765:2010(E). Systems and software engineering – Vocabulary*. Standard. International Organization for Standardization, Dec. 2010.
DOI: 10.1109/IEEESTD.2010.5733835.
URL: http://ieeexplore.ieee.org/document/5733835/citations.

[ISO15] *ISO/IEC 2382:2015. Information technology – Vocabulary*. Standard. International Organization for Standardization, 2015.
URL: http://www.iso.org/iso/home/store/catalogue_ics/catalogue_detail_ics.htm?csnumber=63598.

[ite] itemis AG. *YAKINDU Statechart Tools Website*.
URL: http://www.statecharts.org (visited: Apr. 20, 2017).

[ITE09] *ITEA Roadmap for Software-Intensive Systems and Services*. 3rd ed. ITEA 2 Office Association. Feb. 2009.

[Its] Itseez. *OpenCV Website*.
URL: http://www.opencv.org/ (visited: Oct. 20, 2016).

[JA11] Sylvain Joyeux and Jan Albiez. "Robot development: from components to systems". In: *6th National Conference on Control Architectures of Robots*. Document No. inria-00599679. Grenoble, France, May 2011.
URL: https://hal.inria.fr/inria-00599679.

[Jan12] Slinger Jansen. *Interview by Vincent Wolff-Marting*. IT-Radar Interview. Universität Duisburg-Essen. http://www.it-radar.org/serendipity/archives/101-Software-OEkosysteme-Teil-1.html http://www.it-radar.org/serendipity/archives/102-Software-OEkosysteme-Teil-2.html http://www.it-radar.org/serendipity/archives/103-Software-OEkosysteme-Teil-3.html http://www.it-radar.org/serendipity/archives/104-Software-OEkosysteme-Teil-4.html. 2012.

[Jet16] JetBrains s.r.o. *JetBrains Meta Programming System (MPS) Website*. Online. 2016.
URL: https://www.jetbrains.com/mps/ (visited: Nov. 2, 2016).

[Jos09] Nicolai M. Josuttis. *SOA in Practice: The Art of Distributed System Design (Theory in Practice)*. O'Reilly Media, 2009. ISBN: 0-596-52955-4.
URL: http://www.soa-in-practice.com/soa-glossary.html.

[Joser] *Journal of Software Engineering for Robotics (JOSER)*. ISSN 2035-3928.
URL: http://www.joser.org.

[KB12] Markus Klotzbücher and Herman Bruyninckx. "Coordinating Robotic Tasks and Systems with rFSM Statecharts". In: *Journal of Software Engineering for Robotics (JOSER)* 3.1 (2012). ISSN: 2035–3928.
URL: https://joser.unibg.it/index.php?journal=joser&page=article&op=view&path[]=52.

[Kou15] Anis Koubaa. "ROS As a Service: Web Services for Robot Operating System". In: *Journal of Software Engineering for Robotics (JOSER)* 6.1 (Dec. 2015). ISSN 2035-3928, pp. 1–14.
URL: https://joser.unibg.it/index.php?journal=joser&page=article&op=view&path[]=97.

[KP08] K. Kritikos and D. Plexousakis. “Evaluation of QoS-Based Web Service Matchmaking Algorithms”. In: *2008 IEEE Congress on Services - Part I*. July 2008, pp. 567–574.
DOI: 10.1109/SERVICES-1.2008.53.

[Kum+15] Pranav Srinivas Kumar, William Emfinger, Amogh Kulkarni, Gabor Karsai, Dexter Watkins, Benjamin Gasser, Cameron Ridgewell, and Amrutur Anilkumar. “ROSMOD: A Toolsuite for Modeling, Generating, Deploying, and Managing Distributed Real-time Component-based Software using ROS”. In: *2015 International Symposium on Rapid System Prototyping (RSP)*. Amsterdam, Netherlands, Oct. 2015, pp. 39–45.
DOI: 10.1109/RSP.2015.7416545.

[Kum+16] Pranav Srinivas Kumar, William Emfinger, Gabor Karsai, Dexter Watkins, Benjamin Gasser, and Amrutur Anilkumar. “ROSMOD: A Toolsuite for Modeling, Generating, Deploying, and Managing Distributed Real-time Component-based Software using ROS”. In: *Electronics* 5.3 (Sept. 2016).
DOI: 10.3390/electronics5030053.

[Lee10] Edward A. Lee. “Disciplined Heterogeneous Modeling”. In: *MODELS 2010*. Invited Keynote Talk. Oslo, Norway, Oct. 2010.
URL: https://chess.eecs.berkeley.edu/pubs/706.html.

[Lik32] Rensis Likert. “A technique for the measurement of attitudes”. In: *Archives of Psychology* 22.140 (1932), pp. 1–55.

[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”. In: *International Journal of Information System Modeling and Design (IJISMD)* 5.3 (2014). IGI Global, ISSN 1947-8186, pp. 55–74.
DOI: 10.4018/ijismd.2014070103.

[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: *6th International Workshop on Domain-Specific Languages and models for ROBotic systems (DSLRob ’15)*. Hamburg, Oct. 2015.
URL: http://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: *IEEE International Conference on Simulation, Modeling, and Programming for Autonomous Robots (SIMPAR)*. San Francisco, CA, USA, Dec. 2016, pp. 170–176.
DOI: 10.1109/SIMPAR.2016.7862392.

[Lot+17a] Alex Lotz, **Dennis Stampfer**, Christian Schlegel, Enea Scioni, Nico Huebel, Herman Bruyninckx, Matteo Morelli, Chokri Mraidha, and Sara Tucci. *RobMoSys D2.1: Modeling Foundation Guidelines and Meta-Meta-Model Structures*. Project Deliverable. June 2017.

[Lot+17b] Alex Lotz, **Dennis Stampfer**, Christian Schlegel, Enea Scioni, Nico Huebel, Herman Bruyninckx, Matteo Morelli, Chokri Mraidha, Sara Tucci, Marie-Luise Neitz, and Daniel Meyer-Delius. *RobMoSys D2.2: Initial preparation of (meta-)models, prototypical DSLs, tools and implementation*. Project Deliverable. June 2017.

[LS12] Matthias Lutz and **Dennis Stampfer**. *Informed Active Perception with an Eye-in-hand Camera for Multi Modal Object Recognition*. Video. Oct. 2012.
URL: https://www.youtube.com/watch?v=rqnOqoto598 (visited: Dec. 27, 2016).

[LSS13] Matthias Lutz, **Dennis Stampfer**, and Christian Schlegel. "Probabilistic Object Recognition and Pose Estimation by Fusing Multiple Algorithms". In: *Proc. IEEE International Conference on Robotics and Automation 2013 (ICRA)*. Karlsruhe, Germany, May 2013, pp. 4244–4249.
DOI: 10.1109/ICRA.2013.6631177.

[Lud03] Simone A. Ludwig. "Flexible Semantic Matchmaking Engine". In: *Proceedings of 2nd IASTED International Conference on Information and Knowledge Sharing (IKS)*. AZ, USA, 2003.

[Lut+12] Matthias Lutz, **Dennis Stampfer**, Siegfried Hochdorfer, and Christian Schlegel. "Probabilistic Fusion of Multiple Algorithms for Object Recognition at Information Level". In: *Proc. IEEE International Conference on Technologies for Practical Robot Applications 2012 (TePRA)*. ISBN: 978-1-4673-0854-0. Woburn, MA, USA, Apr. 2012, pp. 139–144.
DOI: 10.1109/TePRA.2012.6215668.

[Lut+13] Matthias Lutz, Timo Hegele, **Dennis Stampfer**, and Alex Lotz. *Collaborative Robot Butler Scenario*. Video. Mar. 2013.
URL: https://www.youtube.com/watch?v=DjjNUPpj36E (visited: Dec. 27, 2016).

[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: *Workshop Roboter-Kontrollarchitekturen, co-located with Informatik 2014*. Ed. by E. Plödereder, L. Grunske, E. Schneider, and D. Ull. Vol. P-232. GI-Edition – Lecture Notes in Informatics (LNI). ISBN: 978-3-88579-626-8. Stuttgart: Bonner Köllen Verlag, Sept. 2014.
URL: https://www.gi.de/service/publikationen/lni/gi-edition-proceedings-2014/gi-edition-lecture-notes-in-informatics-lni-p-232.html.

[Lut+17] Matthias Lutz, **Dennis Stampfer**, Alex Lotz, Christian Verbeek, Sebastian Denz, and Puneeth Rajendra. *Industry 4.0 Robot Commissioning Fleet in Intra-Logistics,*

using Service Robotics for Order Picking. Video. Feb. 2017.
URL: https://www.youtube.com/watch?v=qRSDxBOUVxo (visited: June 3, 2017).

[LVS16] Matthias Lutz, Christian Verbeek, and Christian Schlegel. "Towards a Robot Fleet for Intra-Logistic Tasks: Combining Free Robot Navigation with Multi-Robot Coordination at Bottlenecks". In: *IEEE 21st International Conference on Emerging Technologies and Factory Automation (ETFA 2016)*. Berlin, Germany, Sept. 2016.
DOI: 10.1109/ETFA.2016.7733602.

[Man09] Anne Thomas Manes. *SOA is Dead; Long Live Services*. May 2009.
URL: http://apsblog.burtongroup.com/2009/01/soa-is-dead-long-live-services.html (visited: Apr. 24, 2017).

[Mar+04] David Martin, Mark Burstein, Jerry Hobbs, Ora Lassila, Drew McDermott, Sheila McIlraith, Srini Narayanan, Massimo Paolucci, Bijan Parsia, Terry Payne, Evren Sirin, Naveen Srinivasan, and Katia Sycara. *OWL-S: Semantic Markup for Web Services*. 2004.
URL: https://www.w3.org/Submission/OWL-S/ (visited: Mar. 18, 2017).

[MBF11] Rene van de Molengraft, Michael Beetz, and Toshio Fukuda. "A Special Issue Toward a WWW for Robots [From the Guest Editors]". In: *IEEE Robotics Automation Magazine* 18.2 (June 2011), pp. 20–20. ISSN: 1070-9932.
DOI: 10.1109/MRA.2011.941631.

[MFN06] Giorgio Metta, Paul Fitzpatrick, and Lorenzo Natale. "YARP: Yet Another Robot Platform". In: *International Journal of Advanced Robotic Systems* 3.1 (2006), pp. 43–48. ISSN: 1729-8806.
DOI: 10.5772/5761.

[MH13] Konstantinos Manikas and Klaus Marius Hansen. "Software ecosystems — A systematic literature review". In: *Journal of Systems and Software* 86.5 (May 2013), pp. 1294–1306. ISSN: 0164-1212.
DOI: http://doi.org/10.1016/j.jss.2012.12.026.

[MHB12] Filip Müllers, Dirk Holz, and Sven Behnke. "rxDeveloper: GUI-Aided Software Development in ROS". In: *Seventh full-day Workshop on Software Development and Integration in Robotics (SDIR VII) at IROS 2012*. St. Paul, Minnesota, USA, May 2012.

[Mik+13] Krzysztof Miksa, Pawel Sabina, Andreas Friesen, Tirdad Rahmani, Jens Lemcke, Christian Wende, Srdjan Zivkovic, Uwe Aßmann, and Andreas Bartho. "Case Studies for Marrying Ontology and Software Technologies". In: *Ontology-Driven Software Development*. Ed. by Jeff Z. Pan, Steffen Staab, Uwe Aßmann, Jürgen Ebert, and Yuting Zhao. Berlin, Heidelberg: Springer Berlin Heidelberg,

2013, pp. 69–94. ISBN: 978-3-642-31226-7.
DOI: 10.1007/978-3-642-31226-7_4.

[MLD09] Wolfgang Mahnke, Stefan-Helmut Leitner, and Matthias Damm. *OPC Unified Architecture*. 1st ed. Springer-Verlag Berlin Heidelberg, 2009. ISBN: 978-3-540-68898-3.
DOI: 10.1007/978-3-540-68899-0.

[Moo93] James F. Moore. "Predators and Prey: A New Ecology of Competition". In: *Harvard Business Review* 71.3 (May 1993). Reprint Number 93309, pp. 75–86.

[Nie+10] Tim Niemueller, Alexander Ferrein, Daniel Beck, and Gerhard Lakemeyer. "Design Principles of the Component-Based Robot Software Framework Fawkes". In: *Proc. of Second International Conference on Simulation, Modeling, and Programming for Autonomous Robots (SIMPAR'10)*. Lecture Notes in Computer Science. Darmstadt, Germany: Springer, 2010.

[NKH11] Thomas Nierhoff, Omiros Kourakos, and Sandra Hirche. "Playing Pool with a Dual-Armed Robot". In: *2011 IEEE International Conference on Robotics and Automation*. Shanghai, China, May 2011, pp. 3445–3446.
DOI: 10.1109/ICRA.2011.5980204.

[NOB11] Elisa Yumi Nakagawa, Pablo Oliveira Antonino, and Martin Becker. "Reference Architecture and Product Line Architecture: A Subtle But Critical Difference". In: *Software Architecture: 5th European Conference (ECSA 2011)*. Ed. by Ivica Crnkovic, Volker Gruhn, and Matthias Book. Vol. 6903. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 207–211. ISBN: 978-3-642-23798-0.
DOI: 10.1007/978-3-642-23798-0_22.

[Nor+16] Arne Nordmann, Nico Hochgeschwender, Dennis Wigand, and Sebastian Wrede. "A Survey on Domain-specific Modeling and Languages in Robotics". In: *Journal of Software Engineering for Robotics (JOSER)* 7.1 (July 2016), pp. 75–99. ISSN: 2035–3928.
URL: https://joser.unibg.it/index.php?journal=joser&page=article&op=view&path[]=100&path[]=0.

[Nor08] Linda M. Northrop. *Software Product Line Essentials*. Online. June 2008.
URL: http://resources.sei.cmu.edu/library/asset-view.cfm?assetID=21564 (visited: Mar. 10, 2017).

[Nor90] Fabrice R. Noreils. "Integrating Error Recovery in a Mobile Robot Control System". In: *Proc. of the IEEE International Conference on Robotics and Automation*. May 1990, pp. 396–401.
DOI: 10.1109/ROBOT.1990.126008.

[OMG12a] Object Management Group (OMG). *Robotic Technology Component (RTC)*. Document number: formal/2012-09-01. Sept. 2012.
URL: http://www.omg.org/spec/RTC.

[OMG12b] Object Management Group (OMG). *Service Oriented Architecture Modeling Language (SoaML)*. Document number: formal/2012-05-10. May 2012.
URL: http://www.omg.org/spec/SoaML/.

[OMG13] Object Management Group (OMG). *Unified Component Model for Distributed, Real-Time and Embedded Systems RFP (UCM)*. Document number: mars/2013-09-10. Sept. 2013.
URL: http://www.omg.org/cgi-bin/doc?mars/2013-09-10.

[OMG14] Object Management Group (OMG). *Object Constraint Language (OCL)*. Document number: formal/2014-02-03. Feb. 2014.
URL: http://www.omg.org/spec/OCL/.

[OMG15a] Object Management Group (OMG). *Systems Modeling Language*. Document number: formal/2015-06-03. June 2015.
URL: http://www.omg.org/spec/SysML/1.4/.

[OMG15b] Object Management Group (OMG). *Unified Modeling Language (OMG UML)*. Document number: formal/2015-03-01. Mar. 2015.
URL: http://www.omg.org/spec/UML/2.5/.

[Omm02] Rob van Ommering. "Building Product Populations with Software Components". In: *Proceedings of the 24th International Conference on Software Engineering (ICSE 2002)*. May 2002, pp. 255–265.
DOI: 10.1145/581372.581373.

[OroRTT] *Orocos Real-Time Toolkit Website*.
URL: http://www.orocos.org/rtt (visited: Apr. 23, 2017).

[Ost14] Christopher Oster. "Composable Architecture & Design: Applying Product Line and Systems of Systems Concepts to the Design of Unique, Complex Cyber-Physical Systems". In: *SERC Doctoral Fellows Forum 2014*. Washington, DC, USA, Dec. 2014.
URL: http://www.sercuarc.org/publications-papers/conference-presentation-serc-dff-2014-composable-architecture-design/ (visited: Mar. 10, 2017).

[OW13] Christopher Oster and Jon Wade. "Ecosystem Requirements for Composability and Reuse: An Investigation into Ecosystem Factors That Support Adoption of Composable Practices for Engineering Design". In: *Systems Engineering* 16.4 (Jan. 2013), pp. 439–452.
DOI: 10.1002/sys.21256.
URL: http://dx.doi.org/10.1002/sys.21256.

[Oxf] Oxford University Press. *Oxford Dictionaries Online*.
URL: https://en.oxforddictionaries.com/ (visited: Oct. 29, 2016).

[Papyrus] *Papyrus UML.*
URL: http://www.eclipse.org/papyrus/ (visited: Jan. 25, 2016).

[Pau+12] Liam Paull, Gaetan Severac, Guilherme V. Raffo, Julian Mauricio Angel, Harold Boley, Philip J. Durst, Wendell Gray, Maki Habib, Boa Nguyen, S. Veera Ragavan, Sajad Saeedi G., Ricardo Sanz, Mae Seto, Aleksandar Stefanovski, Michael Trentini, and Haoward Li. "Towards an Ontology for Autonomous Robots". In: *IEEE/RSJ International Conference on Intelligent Robots and Systems 2012 (IROS '12)*. Vilamoura, Algarve, Portugal, Oct. 2012, pp. 1359–1364.
DOI: 10.1109/IROS.2012.6386119.

[PV05] Mirva Peltoniemi and Elisa Vuori. "Business Ecosystem as the New Approach to Complex Adaptive Business Environments". In: *FeBR 2004 - Frontiers of e-Business Research 2004, proceedings of eBRF 2004*. Ed. by M. Seppä, M. Hannula, A-M. Järvelin, J. Kujala, M. Ruohonen, and T. Tiainen. Tampere University of Technology and University of Tampere, 2005, pp. 267–281.

[PW03] Mikel D. Petty and Eric W. Weisel. "A Composability Lexicon". In: *Proc. Spring 2003 Simulation Interoperability Workshop*. 03S-SIW-023. Orlando, USA, Mar. 2003.
URL: http://www.cs.virginia.edu/~rgb2u/03S-SIW-023.doc.

[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.

[RC11] Radu Bogdan Rusu and Steve Cousins. "3D is here: Point Cloud Library (PCL)". In: *2011 IEEE International Conference on Robotics and Automation*. Shanghai, China, May 2011.
DOI: 10.1109/ICRA.2011.5980567.

[ReApp] *ReApp Project Website.*
URL: http://www.reapp-projekt.de (visited: Dec. 27, 2016).

[REC16] REC GmbH. *Openrobotino Wiki*. 2016.
URL: http://wiki.openrobotino.org/index.php?title=Smartsoft (visited: Jan. 25, 2016).

[Rei14] Ulrich Reiser. "Eine webbasierte Integrations- und Testplattform zur Unterstützung des verteilten Entwicklungsprozesses von komplexen Serviceroboter-Applikationen". ISSN: 978-3-8396-0675-9. Dissertation. Universität Stuttgart, 2014.
DOI: 10.18419/opus-6848.

[Ride] *RIDE.*
URL: https://code.google.com/p/brown-ros-pkg/wiki/RIDE (visited: May 5, 2017).

[RMM08] Mikko Raatikainen, Varvana Myllärniemi, and Tomi Männistö. "Comparison of Service and Software Product Family Modeling". In: *Proceedings of the First*

Workshop on Service-Oriented Architectures and Software Product Lines. Ed. by Sholom Cohen and Robert Krut. CMU/SEI-2008-SR-006. Software Engineering Institute. Carnegie Mellon University, May 2008, pp. C1–C10.
URL: http://www.sei.cmu.edu/reports/08sr006.pdf.

[RMT14] Arunkumar Ramaswamy, Bruno Monsuez, and Adriana Tapus. "Model-Driven Software Development Approaches in Robotics Research". In: *Proceedings of the 6th International Workshop on Modeling in Software Engineering (MiSE 2014)*. Hyderabad, India: ACM, June 2014, pp. 43–48. ISBN: 978-1-4503-2849-4.
DOI: 10.1145/2593770.2593781.

[RN03] Stuart J. Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach*. 2nd ed. Prentice Hall Series in Artificial Intelligence. Pearson Education, 2003. ISBN: 0137903952.

[RobMoSys] *RobMoSys Project Website*.
URL: http://www.robmosys.eu (visited: Mar. 1, 2017).

[ROS] ROS Wiki: IDEs. Online.
URL: http://wiki.ros.org/IDEs (visited: Oct. 13, 2016).

[ROS11] ROS Community. *ROS Answers: Which IDE(s) do ROS developers use?* 2011.
URL: http://answers.ros.org/question/9068/which-ides-do-ros-developers-use/ (visited: Oct. 13, 2016).

[RosIde] *ROS Wiki: IDEs in ROS*.
URL: http://wiki.ros.org/IDEs (visited: Jan. 25, 2016).

[Rosin] *ROSIN Website*.
URL: http://www.rosin-project.eu (visited: May 5, 2017).

[RosInd] *ROS Industrial Website*.
URL: http://www.rosindustrial.org (visited: May 1, 2017).

[SA08] Hesham Shokry and Muhammad Ali Babar. "Dynamic Software Product Line Architectures Using Service-Based Computing for Automotive Systems". In: *Proceedings 12th International Software Product Line Conference (SPLC)*. 2008, pp. 53–58.
URL: http://hdl.handle.net/10344/1897.

[Sal+17] Damien Sallé, Guy Caverot, Anders Billeso Beck, and Ugo Cupcic. "Workshop on Systems Engineering - Agile for Robotics". In: *European Robotics Forum 2017 (ERF'17)*. Edinburgh, UK, Mar. 2017.

[SC] Ioan A. Sucan and Sachin Chitta. *MoveIt!*
URL: http://moveit.ros.org (visited: Mar. 1, 2017).

[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: *Work-*

shop Roboter-Kontrollarchitekturen, co-located with Informatik 2013. Vol. P-220. GI-Edition – Lecture Notes in Informatics (LNI). ISBN: 978-3-88579-614-5. Koblenz: Bonner Köllen Verlag, Sept. 2013.
URL: https://www.gi.de/service/publikationen/lni/gi-edition-proceedings-2013/gi-edition-lecture-notes-in-informatics-lni-p-220.html.

[Sch+14] Christian Schlegel, Alex Lotz, Matthias Lutz, and **Dennis Stampfer**. “Supporting Separation of Roles in the SmartMDSD-Toolchain: Three Examples of Integrated DSLs”. In: *5th International Workshop on Domain-specific Languages and Models for Robotic Systems (DSLRob) in conjunction with the 2014 International Conference on Simulation, Modeling, and Programming for Autonomous Robots (SIMPAR 2014)*. Bergamo, Italy, Oct. 2014.

[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* 57.2 (Mar. 2015). ISSN (Online) 2196-7032, ISSN (Print) 1611-2776, DE GRUYTER, pp. 85–98.
DOI: 10.1515/itit-2014-1069.

[Sch04a] Christian Schlegel. “Navigation and Execution for Mobile Robots in Dynamic Environments: An Integrated Approach”. Dissertation. Universität Ulm, 2004.

[Sch04b] Barry Schwartz. “The Tyranny of Choice”. In: *Scientific American* (Apr. 2004), pp. 71–75.

[Sch06] Christian Schlegel. “Communication Patterns as Key Towards Component-Based Robotics”. In: *International Journal of Advanced Robotic Systems* 3.1 (2006), pp. 49–54. ISSN: 1729-8806.

[Sch07] Klaus Schmid. “Keynote Speech”. In: *First International Workshop on Dynamic Software Product Lines*. 2007.

[Sch98] Christian Schlegel. “Fast Local Obstacle Avoidance under Kinematic and Dynamic Constraints for a Mobile Robot”. In: *Proceedings of the International Conference on Intelligent Robots and Systems (IROS)*. Victoria, Canada, 1998, pp. 594–599.

[Ser] Servicerobotik Ulm. *Servicerobotik Ulm Website*.
URL: http://www.servicerobotik-ulm.de (visited: Dec. 27, 2016).

[SF16] **Dennis Stampfer** and Sandra Frank. *Seamless Indoor and Outdoor Navigation based on OpenStreetMap*. Video. Feb. 2016.
URL: https://www.youtube.com/watch?v=_ConuKUOXH4 (visited: Dec. 27, 2016).

[SFC15] **Dennis Stampfer**, Sandra Frank, and Comland d.o.o Team. *Mobile Navigation Through an Indoor Environment: Demonstrating System Integration by Compo-*

sition. Video. July 2015.
URL: https://www.youtube.com/watch?v=cZ3tr3YNoGI (visited: Dec. 27, 2016).

[SFG] Cyrill Stachniss, Udo Frese, and Giorgio Grisetti. *OpenSLAM Community Website*.
URL: http://www.openslam.org/ (visited: Feb. 21, 2017).

[SG99] Anthony J. H. Simons and Ian Graham. “30 Things that go wrong in object modelling with UML 1.3”. In: *Behavioral Specifications of Businesses and Systems*. Kluwer Academic Publishers, 1999. Chap. 17, pp. 237–257.
URL: http://staffwww.dcs.shef.ac.uk/people/A.Simons/research/papers/uml30things.pdf.

[Sil+10] Fernando Silva Parreiras, Tobias Walter, Christian Wende, and Edward Thomas. “Bridging Software Languages and Ontology Technologies: Tutorial Summary”. In: *Proceedings of the ACM International Conference Companion on Object Oriented Programming Systems Languages and Applications Companion*. OOPSLA ’10. Reno/Tahoe, Nevada, USA: ACM, 2010, pp. 311–315. ISBN: 978-1-4503-0240-1.
DOI: 10.1145/1869542.1869626.

[SLS11] Christian Schlegel, Alex Lotz, and Andreas Steck. *SmartSoft: The State Management of a Component*. Tech. rep. University of Applied Sciences Ulm, Jan. 2011.

[SLS12a] **Dennis Stampfer**, Matthias Lutz, and Christian Schlegel. “Information Driven Sensor Placement for Robust Active Object Recognition based on Multiple Views”. In: *Proc. IEEE International Conference on Technologies for Practical Robot Applications 2012 (TePRA)*. ISBN: 978-1-4673-0854-0. Woburn, MA, USA, Apr. 2012, pp. 133–138.
DOI: 10.1109/TePRA.2012.6215667.

[SLS12b] **Dennis Stampfer**, Matthias Lutz, and Christian Schlegel. “Informed Active Perception with an Eye-in-hand Camera for Multi Modal Object Recognition”. In: *Proc. IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS), Workshop on Active Semantic Perception 2012 (ASP’12)*. ISBN: 978-972-8822-26-2. Vilamoura, Algarve, Portugal, Oct. 2012.

[SLS14] **Dennis Stampfer**, Alex Lotz, and Christian Schlegel. *FIONA Deliverable D2.2.1 State of the Art on Service-Oriented Software Component Models*. Project Deliverable. Mar. 2014.

[Sma] SmartSoft Sourceforge Project Page.
URL: https://sourceforge.net/p/smartsoft-ace/ (visited: Oct. 29, 2016).

[SmartBots] *SmartBots@Ulm*.
URL: http://www.hs-ulm.de/Fakultaet/Informatik/StudProjekte/RoboCup/ (visited: Dec. 27, 2016).

[Sof] Software Engineering Group, RWTH Aachen. *The MontiCore Language Workbench*.
URL: http://www.monticore.de/ (visited: May 5, 2017).

[SS10] Andreas Steck and Christian Schlegel. "SmartTCL: An Execution Language for Conditional Reactive Task Execution in a Three Layer Architecture for Service Robots". In: *Proceedings of SIMPAR 2010 Workshops, Intl. Conf. on Simulation, Modeling and Programming for Autonomous Robots*. ISBN: 978-3-00-0328. Darmstadt, Germany, 2010, pp. 274–277.

[SS11a] **Dennis Stampfer** and Christian Schlegel. "Modellierung und Reuse komplexer Verhalten als Bausteine mit Dynamic State Charts am Beispiel der Servicerobotik". In: *Proc. Embedded Software-Engineering Kongress 2011 (ESE)*. ISBN: 978-3-8343-2405-4. Sindelfingen: Vogel-Business-Media Würzburg, Dec. 2011, pp. 511–515.
URL: http://d-nb.info/1082052493.

[SS11b] Andreas Steck and Christian Schlegel. "Managing Execution Variants in Task Coordination by Exploiting Design-Time Models at Run-Time". In: *Proc. IEEE Int. Conf. on Intelligent Robots and Systems (IROS'11)*. San Francisco, CA, USA, Sept. 2011, pp. 2064–2069.
DOI: 10.1109/IROS.2011.6094732.

[SS13] **Dennis Stampfer** and Christian Schlegel. "Dynamic State Charts: Composition and Coordination of Complex Robot Behavior and Reuse of Action Plots". In: *Proc. IEEE International Conference on Proceedings of Technologies for Practical Robot Applications 2013 (TePRA)*. ISBN: 978-1-4673-6224-5. Woburn, Massachusetts, USA, Apr. 2013, pp. 1–6.
DOI: 10.1109/TePRA.2013.6556375.

[SS14a] Christian Schlegel and **Dennis Stampfer**. "The SmartMDSD Toolchain: Supporting dynamic reconfiguration by managing variability in robotics software development". In: *Tutorial on Managing Software Variability in conjunction with the Robot Control Systems at Robotics: Science and Systems Conference (RSS 2014)*. Berkeley, CA, USA, July 2014.

[SS14b] **Dennis Stampfer** and Christian Schlegel. "Dynamic State Charts: Composition and Coordination of Complex Robot Behavior and Reuse of Action Plots". In: *Journal of Intelligent Service Robotics* 7.2 (Mar. 2014). Springer Berlin Heidelberg, pp. 53–65. ISSN: 1861-2784.
DOI: 10.1007/s11370-014-0145-y.

[SSF16] **Dennis Stampfer**, Christian Schlegel, and Sandra Frank. *FIONA Deliverable D2.4.1 Handbook for Integrating Basic Services and Interface Components*. Project Deliverable. Mar. 2016.

[SSS09a] Andreas Steck, **Dennis Stampfer**, and Christian Schlegel. "Modellgetriebene Softwareentwicklung für Robotiksysteme". In: *Proc. 21. Fachgespräch Autonome Mobile Systeme 2009 (AMS)*. Ed. by Rüdiger Dillmann, Jürgen Beyerer, Christoph Stiller, J. Marius Zöllner, and Tobias Gindele. Informatik Aktuell. Karlsruhe: Springer Berlin Heidelberg, Dec. 2009, pp. 241–248. ISBN: 978-3-642-10284-4. DOI: 10.1007/978-3-642-10284-4_31.

[SSS09b] Andreas Steck, **Dennis Stampfer**, and Christian Schlegel. "Software-Engineering in der Servicerobotik – Der Weg zum modellgetriebenen Softwareentwurf". In: *Proc. Embedded Software-Engineering Kongress 2009 (ESE)*. ISBN: 978-3-8343-2402-3. Sindelfingen: Vogel-Business-Media Würzburg, 2009, pp. 513–516. URL: http://d-nb.info/999874837.

[SSS16] Christian Schlegel, Ulrik P. Schultz, and Serge Stinckwich. "From the Guest Editors of the Special Issue on Domain-Specific Languages and Models for Robotic Systems". In: *Journal of Software Engineering for Robotics (JOSER)* 7.1 (June 2016), pp. 1–2.

[Sta+10] Steffen Staab, Tobias Walter, Gerd Gröner, and Fernando Silva Parreiras. "Model Driven Engineering with Ontology Technologies". In: *Reasoning Web. Semantic Technologies for Software Engineering: 6th International Summer School 2010, Dresden, Germany, August 30 - September 3, 2010. Tutorial Lectures*. Ed. by Uwe Aßmann, Andreas Bartho, and Christian Wende. Vol. 6325. Lecture Notes in Computer Science. Springer-Verlag Berlin Heidelberg, 2010, pp. 62–98. ISBN: 978-3-642-15543-7.
DOI: 10.1007/978-3-642-15543-7_3.

[Sta+14a] **Dennis Stampfer**, Sandra Frank, Matthias Lutz, and Alex Lotz. *HSU iBeacon Example: Mobile navigator through an indoor environment*. Video. Oct. 2014.
URL: https://www.youtube.com/watch?v=G6fwnBAtyNc (visited: Dec. 27, 2016).

[Sta+14b] **Dennis Stampfer**, Sandra Frank, Matthias Lutz, and Alex Lotz. *HSU iBeacon Example: Mobile navigator through an indoor environment (using the MORSE simulator)*. Video. Dec. 2014.
URL: https://www.youtube.com/watch?v=qdetfVMP9is (visited: Dec. 27, 2016).

[Sta+15] **Dennis Stampfer**, Christian Schlegel, Mathias Bürger, Christopher Brown, Wei Mao, Mitja Pugelj, Neda Petreska, Ali Golestani, Stefan Rueping, and Çağlar Akman. *FIONA Deliverable D2.3.1: FIONA Platform Architecture*. Project Deliverable. June 2015.

[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: *Journal of Software Engineering for Robotics (JOSER): Special Issue on Domain-Specific Languages and*

Models in Robotics (DSLRob) 7.1 (July 2016). ISSN 2035-3928, pp. 3–19.
URL: https://joser.unibg.it/index.php?journal=joser&page=article&op=view&path[]=91.

[Sta+17] **Dennis Stampfer**, Alex Lotz, Vineet Nagrath, Christian Schlegel, Mathias Lüdtke, Björn Kahl, Sebastian Friedl, Christian von Arnim, Johannes Baumgartl, and Simon Krais. *SeRoNet Deliverable D4.1 "Leitfaden"*. Project Deliverable. Aug. 2017.

[Sta15] Michael Stal. *heise Developer: Der Pragmatische Architekt: Was heißt hier Softwarearchitektur?* Online. 2015.
URL: http://heise.de/-2808249 (visited: June 6, 2015).

[Ste+08] Dave Steinberg, Frank Budinsky, Marcelo Paternostro, and Ed Merks. *EMF Eclipse Modeling Framework*. Ed. by Erich Gamma, Lee Nackman, and John Wiegand. 2nd ed. The Eclipse Series. Addison Wesley, 2008. ISBN: 978-0-321-33188-5.

[SW02] Judith Stafford and Kurt Wallnau. "Component Composition and Integration". In: *Building Reliable Component-based Software Systems*. Ed. by Ivica Crnkovic and Magnus Larsson. Artech House, Inc, 2002, pp. 179–191. ISBN: 9781580533270.

[SW04] David Sprott and Lawrence Wilkes. *Understanding Service-Oriented Architecture*. CBDI Forum. Jan. 2004.
URL: https://msdn.microsoft.com/en-us/library/aa480021.aspx (visited: Apr. 24, 2017).

[SW99a] Christian Schlegel and Robert Wörz. "Interfacing Different Layers of a Multilayer Architecture for Sensorimotor Systems using the Object-Oriented Framework SmartSoft". In: *Third European Workshop on Advanced Mobile Robots. (Eurobot'99)*. Zurich, Switzerland: IEEE, Sept. 1999, pp. 195–202. ISBN: 0-7803-5672-1.
DOI: 10.1109/EURBOT.1999.827640.

[SW99b] Christian Schlegel and Robert Wörz. "The Software Framework SMARTSOFT for Implementing Sensorimotor Systems". In: *1999 IEEE/RSJ International Conference on Intelligent Robots and Systems*. Vol. 3. IEEE, 1999, pp. 1610–1616. ISBN: 0-7803-5184-3.
DOI: 10.1109/IROS.1999.811709.

[Szy02] Clemens Szyperski. *Component Software: Beyond Object-Oriented Programming*. ISBN: 0-201-74572-0. Addison Wesley, 2002.

[Szy03] Clemens Szyperski. "Component technology - what, where, and how?" In: *Proceedings of the 25th International Conference on Software Engineering 2003*. IEEE, May 2003, pp. 684–693.
DOI: 10.1109/ICSE.2003.1201255.

[TB09] Moritz Tenorth and Michael Beetz. "KNOWROB – Knowledge Processing for Autonomous Personal Robots". In: *IEEE/RSJ International Conference on Intelligent Robots and Systems 2009 (IROS '09)*. St. Louis, USA, Oct. 2009, pp. 4261–

4266.
DOI: 10.1109/IROS.2009.5354602.

[TcSoft] *RAS Technical Committee on Software Engineering for Robotics and Automation (TC-SOFT).*
URL: http://robotics.unibg.it/tcsoft/ (visited: Mar. 1, 2017).

[TH04] Anthony Towns and Joey Hess. *File Formats: interfaces(5). Debian GNU/Linux Manpage*. Apr. 2004.

[Vaa+] Sami Vaarala et al. *Duktape Website*.
URL: http://www.duktape.org (visited: Nov. 8, 2016).

[Vli98] John Vlissides. *Pattern Hatching: Design Patterns Applied*. The software patterns series. ISBN: 9780201432930. Addison-Wesley Professional, 1998.
URL: http://www.informit.com/store/pattern-hatching-design-patterns-applied-9780201432930.

[Voe+13] Markus Voelter, Daniel Ratiu, Bernd Kolb, and Bernhard Schaetz. “mbeddr: Instantiating a Language Workbench in the Embedded Software Domain”. In: *Journal of Automated Software Engineering* 20.3 (Sept. 2013), pp. 339–390.
DOI: 10.1007/s10515-013-0120-4.

[Voe13] Markus Voelter. *DSL Engineering*. CreateSpace Independent Publishing Platform, Jan. 2013.
URL: http://www.dslbook.org.

[Völ11] Markus Völter. “From Programming to Modeling - and Back Again”. In: *IEEE Software* 28.6 (Nov. 2011), pp. 20–25.
DOI: 10.1109/ms.2011.139.
URL: http://dx.doi.org/10.1109/MS.2011.139.

[W3C07a] World Wide Web Consortium (W3C). *Simple Object Access Protocol (SOAP) 1.2*. Apr. 2007.
URL: https://www.w3.org/TR/soap12/ (visited: May 2, 2017).

[W3C07b] World Wide Web Consortium (W3C). *Web Services Description Language (WSDL) Version 2.0*. June 2007.
URL: https://www.w3.org/TR/wsdl20/ (visited: Apr. 1, 2017).

[W3C12] World Wide Web Consortium (W3C). *OWL 2 Web Ontology Language Document Overview (Second Edition)*. Dec. 2012.
URL: https://www.w3.org/TR/owl2-overview/ (visited: Mar. 18, 2017).

[Wen+16] Monika Wenger, Waldemar Eisenmenger, Georg Neugschwandtner, Ben Schneider, and Alois Zoitl. “A Model Based Engineering Tool for ROS Component Compositioning, Configuration and Generation of Deployment Information”. In: *IEEE 21st International Conference on Emerging Technologies and Factory Au-*

tomation 2016 (ETFA). Berlin, Germany, Sept. 2016. ISBN: 978-1-5090-1314-2. DOI: 10.1109/ETFA.2016.7733559.

[WG04] Diana L. Webber and Hassan Gomaa. "Modeling variability in software product lines with the variation point model". In: *Science of Computer Programming* 53.3 (Dec. 2004), pp. 305–331. ISSN: 0167-6423.
DOI: 10.1016/j.scico.2003.04.004.

[Wit12] Christoph Witte. "Was hat denn SOA mit Innovation zu tun?" In: *Computerwoche*. Vol. 16. 16/2012. ISSN: 0170-5121. Apr. 2012, 36ff.
URL: https://www.wiso-net.de/document/CW__20120416031486400348548907O4.

[WSO06] Hiroshi Wada, Junichi Suzuki, and Katsuya Oba. "Modeling Non-Functional Aspects in Service Oriented Architecture". In: *2006 IEEE International Conference on Services Computing (SCC'06)*. Illinois, USA, Sept. 2006, pp. 222–229. DOI: 10.1109/SCC.2006.74.

[Wyl12] Diego Wyllie. *App-Entwicklung: Die besten Open-Source-Frameworks für iOS, Android und Co.* Apr. 2012.
URL: http://www.muensolutions.com/de/app-entwicklung-die-besten-open-source-frameworks-fur-ios-android-und-co.html (visited: Apr. 22, 2017).

[Xama] Xamarin Inc. *Xamarin Component Website.*
URL: http://components.xamarin.com (visited: Apr. 20, 2016).

[YL07] Stephen S. Yau and Junwei Liu. "Functionality-Based Service Matchmaking for Service-Oriented Architecture". In: *Eighth International Symposium on Autonomous Decentralized Systems (ISADS'07)*. IEEE, Mar. 2007, pp. 147–154.
DOI: 10.1109/ISADS.2007.39.

[ZAF16] Stefan Zander, Nadia Ahmed, and Matthias Frank. "A Survey about the Usage of Semantic Technologies for the Description of Robotic Components and Capabilities". In: *Proceedings of the 1st International Workshop on Science, Application and Methods in Industry 4.0 (SAMI 2016), co-located with i-KNOW 2016*. Graz, Austria, Oct. 2016.
URL: http://ceur-ws.org/Vol-1793/.

[ZAFH] *ZAFH Servicerobotik Website.*
URL: http://www.zafh-servicerobotik.de (visited: Jan. 25, 2016).

[Zan+15] Stefan Zander, Georg Heppner, Georg Neugschwandtner, Ramez Awad, Marc Essinger, and Nadia Ahmed. "A Model-Driven Engineering Approach for ROS using Ontological Semantics". In: *6th International Workshop on Domain-Specific Languages and models for ROBotic systems (DSLRob-15), co-located with IROS 2015*. arXiv:1601.03998. Sept. 2015.

[Zin05] Enrico Zini. "A cute introduction to Debtags". In: *Debian Developer Conference 2005 (Debconf5)*. Helsinki, Finland, July 2005.

URL: https://debconf5.debconf.org/comas/general/proposals/54.html (visited: Feb. 21, 2017).

[ZSK15] Begüm Ilke Zilci, Mathias Slawik, and Axel Küpper. “Cloud Service Matchmaking Using Constraint Programming”. In: *2015 IEEE 24th International Conference on Enabling Technologies: Infrastructure for Collaborative Enterprises*. IEEE, June 2015, pp. 63–68.
DOI: 10.1109/WETICE.2015.44.