Technische Universität München
Fakultät für Informatik

Composable Coordination for Service Robots: A Model-Driven Approach

Matthias Lutz

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.-Ing. Alin Albu-Schäffer

Prüfer der Dissertation:

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

Die Dissertation wurde am 18.10.2021 bei der Technischen Universität München eingereicht und durch die Fakultät für Informatik am 05.04.2022 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.

Service Robotics
autonomous mobile service robots

Zusammenfassung

Die Vision einer flexiblen und universellen Maschine, welche in der Lage ist, autonom verschiedene Aufgaben zu bewerkstelligen, ist eine der bemerkenswertesten und charakteristischsten Eigenschaften eines Roboters. Diese Eigenschaft spiegelt sich direkt in der Software wider, aus der ein Robotiksystem aufgebaut ist. Die einzelnen Systemteile müssen koordiniert werden, um die unterschiedlichen Aufgaben zu bearbeiten.

Softwareentwicklung für Serviceroboter ist vor allem ein Integrationsproblem, welches das Zusammensetzen verschiedener Bestandteile, das Wechselspiel unterschiedlicher Experten in verschiedenen Rollen, sowie das Anwenden verschiedenster Technologien beinhaltet. Die Integration der unterschiedlichen Teile und Expertisen erfordert einen enormen Aufwand und stellt einen der Kostentreiber für die Entwicklung von Robotikanwendungen dar. Die Systemintegratoren oder Produktentwickler müssen für die Entwicklung die gesamte Wertschöpfungskette an Kompetenz abdecken und ensprechend Expertise in vielen Technologiebereichen besitzen. Dies verhindert den Einsatz von Robotiklösungen in vielen Bereichen, insbesondere in solchen, die maßgeschneiderte Lösungen in kleinen Losgrößen erfordern.

Diese Arbeit leistet einen Beitrag zur Robotik-Softwareentwicklung und speziell zur Entwicklung von Handlungskoordination in der Servicerobotik. Die Handlungskoordination von Robotiksoftwaresystemen ist dabei ein außerordentlich wichtiges Element, sie verbindet das programmierte Verhalten (engl. Tasking) mit den funktionalen oder algorithmischen Softwareteilen. Ziel der Arbeit ist es, die Entwicklung von Robotersystemen inkl. Handlungskoordination durch das Zusammenstellen vorgefertigter Bausteine zu ermöglichen, welche räumlich und zeitlich getrennt im Kontext eines Robotik-Business-Ökosystems entwickelt werden. Diese Arbeit schlägt Strukturen und Muster vor, welche die Lücke zwischen den funktionalen Bausteinen und den Verhaltensbaustein schließen und das Zusammensetzen dieser getrennt entwickelten Bausteine ermöglichen. Die Einführung von Skills und Tasks als zwei getrennte Abstraktionsebenen für die Handlungskoordination ermöglicht die separierte Entwicklung von zusammensetzbaren Verhaltensbausteinen. Diese Arbeit stellte weiter ein Komponentenmodell mit einer dedizierten Komponenten-Koordinationsschnittstelle vor, welche den koordinierten Zugriff auf die Funktionalität in den Softwarekomponenten ermöglicht. Schließlich trägt diese Arbeit integrierte Werkzeuge (IDE) für die Entwicklung von Robotik Software Systemen inkl. Handlungskoordination bei, welche die Anwendungsentwicklung aus Ökosystem-Bausteinen vereinfacht.

Der Ansatz und die Beiträge werden anhand mehrerer Systeme und Anwendungen demonstriert und validiert. Diese reichen von Laborprototypen, welche mit Partnern in Forschungsprojekten entwickelten wurden, bis hin zu auf dem Markt verfügbaren Produkten.

Abstract

The vision of a flexible and somewhat universal machine capable of autonomously performing many different tasks is one of the most remarkable and distinguishing properties of a robot. This property is directly reflected in the software a robotics system is built of. A system that can perform multiple tasks requires the coordination of the system parts and their capabilities.

Robotics software development is foremost a challenge that involves the integration of different software parts, the interaction of different experts in different roles, and the application of many different technologies. The integration of all these different parts and expertise requires a huge effort and pushes the costs for robotics application development. Most robots used in real-world applications are still hand-crafted, highly optimized products, where system integrators or product developers need to cover the entire value chain. They need to become an expert in all the required technology fields. This prevents the widespread usage of robotic solutions in many domains, especially those that require tailored small batch size solutions.

This thesis contributes to service robotics software and behavior coordination development. The behavior coordination of robotic systems is crucial as it connects the developed behavior with the functional or algorithmic software parts. This thesis aims for the development of robotic systems, including robotics behavior, by composing "as is" building blocks that are developed separated in space and time within robotics business ecosystems. This thesis proposes structures and patterns that bridge the gap between the development of functionalities and robotics behavior coordination. The introduction of skills and tasks as two separated abstraction levels of robotics behaviors enables the separated development of composable behavior building blocks. The thesis further contributes a dedicated component coordination interface to enable coordinating access to the functionalities realized within software components. The organization of robotics application development is structured in a proposed workflow, incorporating the contributing roles in the ecosystem. Finally, this thesis proposes integrated tooling (IDE) for robotics behavior development, supporting the user in developing robotic applications composed of ecosystem-developed building blocks.

The approach and the contributions are demonstrated and validated using several systems and applications developed with partners in industry and academia, ranging from lab prototypes developed and demonstrated in many research projects to industry-sold products.

Acknowledgements

I would like to thank Prof. Dr. Alois Knoll at the Technical University of Munich for the opportunity to prepare this thesis. I would particularly like to thank Prof. Dr. Christian Schlegel at the Service Robotics Research Center at the Ulm University of Applied Sciences (Technische Hochschule Ulm). He provided the environment and the setting for many years full of challenging topics and interesting work. Christian, I am grateful for the opportunity to work in an excellent research environment and an extraordinary team. Thanks for the endless support and motivation while writing this thesis.

I would also like to thank the current and former members of the Service Robotics Research Center. In particular, I owe many thanks to my friends and colleges Dr. Dennis Stampfer and Dr. Alex Lotz. Without your support, I probably would not have finished this thesis. Thanks for the uncounted ups and downs and night shifts we withstood together. May what we have achieved together always motivate us to take on new challenges.

My gratitude also goes to my family and parents for their encouragement and support in all these years. Finally, I want to thank my wife Anne for her constant support and patience during the last years, especially in this thesis's final phase.

Contents

1. Introduction

Service robotics is one of the top emerging technologies with a market growth rate above 30% over the last few years (2018, 2019) and a similar expected market growth rate in the near future [Mül+20]. Despite this impressive growth, the only visible service robots in complex real-world applications are isolated and limited examples of service robots. While sensor and hardware development has made rather drastic improvements, such as the introduction of high-performance calculation to mobile applications, or the introduction of new 3D sensors, systematic software development for robotics remained underestimated and neglected. The challenges posed by software development for robotics have been identified as important - they are the major hurdles to be overcome to get the service robotic business really running (see [HBK11, p. 339, p. 351]).
Robotics software development is foremost a challenge that involves the integration of different software parts, different experts and roles, and many different technologies and algorithms. To make robotics and robotic applications a success, the challenges faced by software development and integration have to be tackled systematically. This is seen by many organizations like the European Commission (see "Robotics 2020 Multi-Annual Roadmap" [euR16] and the "Strategic Research Agenda" [euR13]) and studies such as EFFIROB [HBK11]. They state the necessity for robotics-specific software development engineering methods and tools that enable the step change from hand-crafted systems to systems that are systematically composed of reusable software parts, which are the key to a thriving business ecosystem. The European SPARC Robotics initiative and the German BMWi PAiCE program address these challenges. The recent (e.g. RobMoSys EU Horizon 2020 [RobMoSys]) and ongoing (e.g. German BMWi PAiCE SeRoNet [Bun17] until November 2021) initiative address the need for structures and software engineering in robotics. This thesis contributes in this context to composable robotics behavior.

Most effort concerning software development is, however, spent on developing algorithms, including the implementation of new capabilities. There is only a limited focus on the methods of how to build software for complex robotic systems. Therefore, most robots used in real-world applications are still hand-crafted, highly optimized products, or prototypes developed in labs for demonstrating new robotic algorithms and capabilities. The widespread usage of ROS within the robotics community led to a unification in robotics software development, at least to a certain extent. The ROS node as a component and the tools provided by ROS led to an ecosystem, which makes many different functionalities accessible for robotics system development. However, ROS does not provide the necessary structures to enable the composition of building blocks. ROS was intended for research by academic institutions [Ger14] and is deliberately developed

not to provide such structures, but to offer the maximum flexibility [Ger15]. ROS2 addresses this lack to some extent and provides some guiding structures; however, ROS2 still aims to maintain the flexibility of ROS [Ger15].

The vision of a flexible and somewhat universal machine being capable of autonomously performing many different tasks is one of the most remarkable and distinguishing properties of a robot. This property is directly reflected in the software a robotics system is built of. A system that can perform multiple tasks requires the coordination of the system parts and their capabilities to do so. While many coordination approaches are available today, their integration into robotics software systems is still hand-crafted. This results in tightly coupled or even interwoven coordination approaches that are bound to functional parts of the system. It is not possible to separate the roles involved, so each role must cover a large part, if not all, of the value chain. To get service robotics from developing expensive hand-crafted one-of-a-kind pieces or prototypes build from scratch to affordable mass-market products composed of building blocks, a more structured approach to developing software and dealing with coordination has to be introduced. Structures are required to manage the interfaces between the involved roles and building blocks on many different abstraction levels, see Figure 1.1. The approach of this thesis enables the development of system parts decoupled in time and space (separation of roles), as well as the integration of the separately developed parts in a composition-like manner, considering the robotics behavior coordination. This is the baseline to enable the development of service robotic products using value networks with different roles providing their expertise and collaborating in robotics business ecosystems.

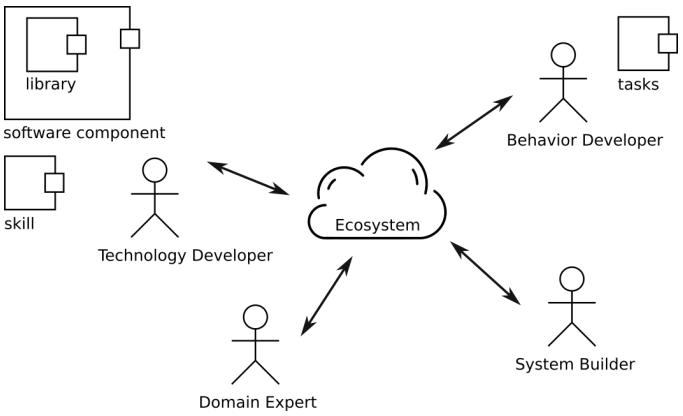

Figure 1.1.: Robotics software engineering is about managing the interfaces between roles and building blocks. The interfaces are the key to the composition and the separation of roles that enable a thriving robotics business ecosystem.

1.1. Motivation - Envisioned User Stories

The following paragraphs illustrate short user stories that envision the overall idea and the goal of this thesis. All the user stories deal with the development of robotics software systems and applications in general; they deal with the related coordination aspects and the taskability of robots in more detail. The underlying story is to enable the easy composition of building blocks developed by different roles and shared within an ecosystem. The stories highlight the tasking and coordination aspects involved in robotic system development.

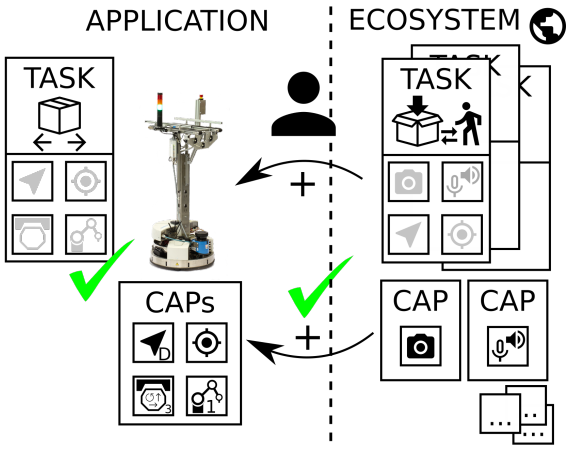

Figure 1.2.: A robot user selects and fetches new tasks and capabilities (CAPs) from an ecosystem marketplace and adds those to the ones already onboard the robot by composition.

New Tasks - Existing Robotic System: An existing robotic system is so far solely used for the transportation of goods in a logistics environment. It shall now be used to support workers to perform order-picking jobs, illustrated in Figure 1.2.

The system administrator or a skilled operator, hereinafter referred to as user of the robotic system, is able to extend the capabilities of the robot. The user utilizes a dedicated software tool to connect to both the existing robotic system and an ecosystem market containing available robot building blocks. This tool supports the user in filtering and selecting those building blocks the user is looking for, with matching types and properties. The user wants to be able to use the system for new job types. Therefore, the user selects and downloads a new robot task for human-robot collaborative order picking to realize the new jobs. While adding the new task to the existing system (keeping the previously available tasks and capabilities of the robot), the tooling highlights the capabilities required by the new task that so far is missing in the existing robotic system. In this example, the robot already features navigation, localization, and simple object manipulation capabilities. The new task additionally requires person-tracking and

human-robot interaction capabilities. The user has the option to choose from different available and compatible building blocks featuring different properties, such as price, license model, or realization details. The new block, in turn, could also require input from other building blocks – for instance, sensor information.
Once selected, the user is able to download matching building blocks from the ecosystem market. There is no need for the user to understand the internals of the building blocks. Those can also be inaccessible for closed source building blocks. The building blocks come with all information, representations, and software artifacts to compose them.
The user selects a speech-based human-robot interaction building block and a person-tracking building block that uses computer vision and requires sensory data (images) from a camera already present in the current robot (e.g., used for a different purpose). The user can compose the building blocks with the already used ones in the current system by selecting the matching interfaces only. There is no need to adapt any of the existing or new capability building blocks. With the missing capabilities now being available, the user can compose the new task building blocks that they want to use. The user is supported in composing the new tasks to the existing system, selecting when (in which situations) and how (connection to other tasks) the new tasks should be used by the robot. Again, there are no modifications to the building blocks. This time, however, the task building blocks are necessary. The blocks developed by some other ecosystem participants can be composed and decoupled in time and space. Finally, the user is able to deploy the updated software system to the physical robot, which is now capable of performing both tasks, namely the new and the old one.

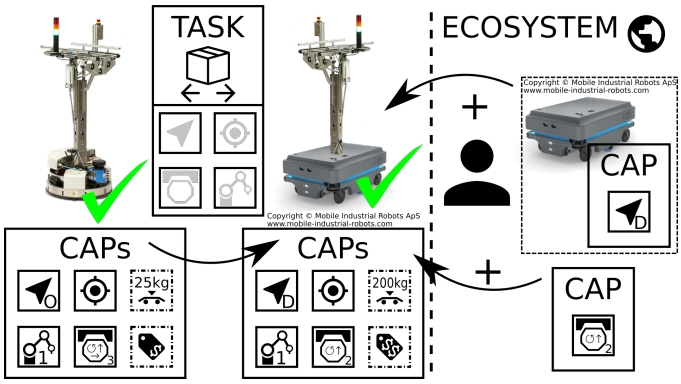

Figure 1.3.: A robot user modifies a robotic system, keeping the application and the tasks. No longer matching building blocks are replaced with new ones fetched from an ecosystem market place and composed to the existing ones.

Same Tasks - Modification of Existing Robotics System: An existing robotics system used for the transportation of goods in a logistics environment shall be extended to support a higher payload. The task building blocks the robot is executing, namely for

the transportation of goods, shall be reused. A stronger (higher payload) robot base needs to support the same capabilities the reused task requires, illustrated in Figure 1.3.

The system integrator or the skilled robotics operator, with some robotics experience but not a software expert, hereinafter referred to as user, uses dedicated tooling to connect to both the existing robotic system and an ecosystem market, same as in the previous user story. To be able to fulfill tasks of moving around a higher payload, the robotics base (drive, etc.) needs to be replaced. The physical robot base features different properties and the software building blocks, representing the base, explicate some of those properties through the provided capabilities of the switched blocks. The blocks feature digital data sheets that contain properties such as the interfaces of the block or non-functional ones such as the price or payload. In this example, the existing robot base features an omnidirectional drive, proving lateral movement capabilities. The application makes use of lateral movements to dock to stations for the handover of goods. With the need to transport heavier goods, the user selects a robot base featuring a higher payload, but provides a different motion model (differential drive). This new base provides the movement capability through a matching building block, which is, however, not able to perform lateral movements. The user replaces the software building block representing the old base with the one for the new base. Due to the not-matching interfaces (no lateral movements), the building blocks realizing the capability to dock the robot to a production station for the handover of goods cannot be composed with the new base building block. The tooling highlights the incompatible interfaces so that the user can locate the problem. In this example, the user replaces the docking building blocks using lateral movement with a planning-based approach capable of docking with differential drive kinematics, providing the same capability to the existing tasks. The user fetches those building blocks from the ecosystem market as well. With matching interfaces among the different building blocks, the user is able to compose a working system with no need to modify the software part realizing the tasking of the application. Finally, the user is able to deploy the updated software system to the changed physical robot, which is now capable of performing the same task with heavier goods.

Transferability of Tasks to Other Robotic Systems: An existing robotics system is able to perform the task of preparing and serving coffee. The system integrator or the skilled robotics operator, with some robotics experience but not a software expert, hereinafter referred to as user, shall be able to check whether the task could be executed by other robotic systems as well. In contrast to the previous user story, which was centered around the modification of an existing system, this user story focuses on the transferability of tasks to other robotic systems, illustrated in Figure 1.4.

The user is supported using dedicated tooling to connect to both the existing robotic system and an ecosystem market, the same as in the previous user story. The other robotic systems and their properties are available in the ecosystem market. The user selects the task in question and tries to compose it with other robotic systems he might

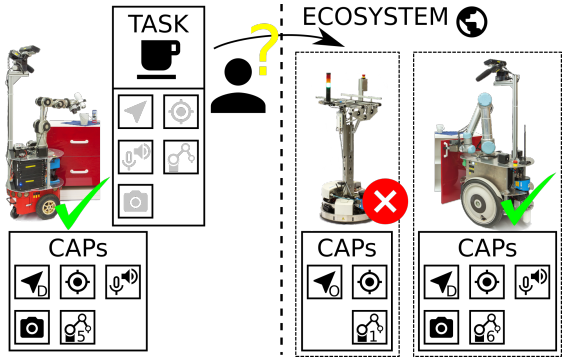

Figure 1.4.: A robot user validates if an existing task can be transferred to an other robotics system.

want to use. In this example, the user checks for compatibility of the task with two different robotic systems; a Robotino robot featuring a conveyor belt, as well as the services robot Larry equipped with a six DOF manipulator. As the task requires complex manipulation capabilities, the tooling checks whether matching capabilities are present in the potential systems. As a result, the tooling indicates the incompatible system and those interfaces which are not matching. Thereby, the user is able to check whether a task is transferable to another robotics system without the need to understand the internals of the realization of the task or the other building blocks. In this example, obviously, the Robotino is not capable of handling coffee cups, whereas Larry, equipped with a manipulator, is suitable to execute the task.

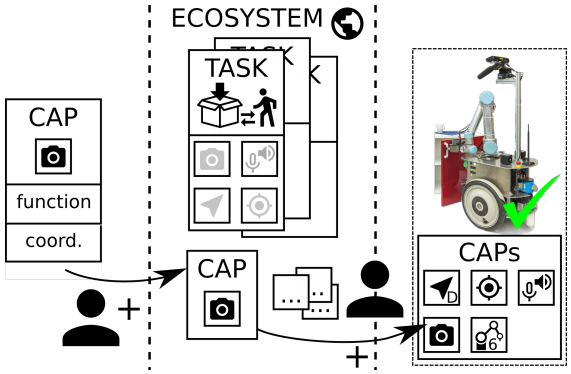

Figure 1.5.: A technology expert develops a functionality that provides a capability usable for the coordination and the tasking of a robot. He is able to push his building block into the ecosystem market. Others are able to pull his block from the marketplace and compose it with other building blocks, e.g., by modifying existing applications or creating new ones.

New Capability Building Blocks - Independent of Applications: An object recognition technology expert (hereinafter referred to as user) developing a recognition approach shall be able to sell his approach as a building block via an ecosystem marketplace. In contrast to the others, this user story is not directly dealing with a specific existing robotics system, but a specific existing functionality. The user story further takes the perspective of a technology expert and not the one of a system integrator or robotics system user. The envisioned user story is illustrated in Figure 1.5.

The user develops his recognition approach independent of any robotics system, resulting in a library. He then uses dedicated tooling to wrap his library into a building block that supports the structures required for composition and for exchange in a robotics ecosystem. Making use of the building block in complex applications requires coordinational access to the functional realization such as configuration or activation. Further, the capability needs to be put together with other capabilities. To make use of the object-recognition building block, for example, the capability to recognize objects needs to be used together with other capabilities to realize an application. To serve a cup of coffee, to stick with the example from the other user story, the recognition is required to grasp the coffee cup and hence the results of the recognition need to be used by other capabilities such as manipulation. The user, therefore, additionally defines how the building block can be used and coordinated in complex robotic systems. Therefore, the user realizes a capability usable by tasks to realize applications. As many algorithms intrinsically require input from others, and in many cases provide results to others, as in this case poses of objects, the building block needs to adhere to certain interfaces. The user is able to search the ecosystem for matching interface definitions, thereby providing composable building blocks ready for composition by other ecosystem participants.

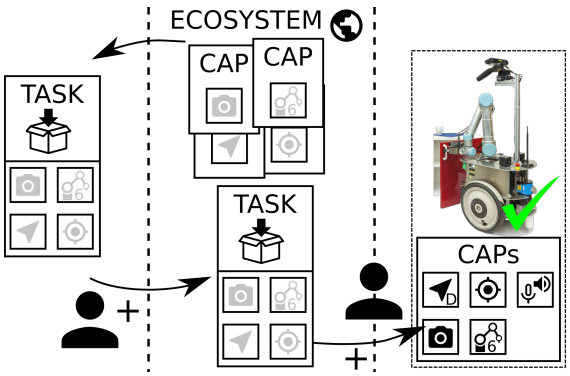

Figure 1.6.: A robotics behavior developer creates a task, specifying which capabilities it requires for the execution without binding the task to a specific realization of capabilities. The role is able to push its task as building block to an ecosystem marketplace where others are able to pull it and use it with their systems, given they offer all required capabilities.

New Task Building Blocks - Independent of Robot: A robotics behavior developer (in short: user) shall be able to create a behavior block encoding how a robot is able to perform a specific task or job. This user story is not dealing with a concrete robotics system, but with an application to develop. The user story takes the perspective of a robotics behavior developer with insights into how an application can be realized within a certain domain, illustrated in Figure 1.6.

In this example, the user develops an intralogistics order-picking application. To realize the behavior for the application, the user is able to use capabilities defined by domain experts, accessible via an ecosystem marketplace. In this example of an order-picking task using a mobile robotics platform with a manipulator and a camera mounted on the robot, the user requires the capabilities to move the robot and to detect and grasp objects. The user is able to fetch from the ecosystem marketplace the interfaces of the required and to develop the order-picking task without binding the task to a certain robot or any specific functional building block – e.g., object recognition component. Once done, the user can test the task using different robots and can publish his work as a robot-independent task building blocks. Other ecosystem users are then able to fetch these building blocks, compose them to their systems, and run them on their robots, as long as they provide the required capabilities.

User Stories - Summary

The presented user stories all illustrate certain aspects of robotics behavior development in an ecosystem driven by the idea of composable building blocks. The user stories envision the decoupled development of system parts by individual experts (separation of roles). Each user story is related to an ecosystem and one or many derived marketplaces. Many users pull content (building blocks) from the market, while some push their building blocks to it. While enabling the decoupled development of building blocks is a major prerequisite for the ecosystem, the even more important one is the ability to put building blocks together and to compose them to working robotic systems. None of the user stories would work if the individual participants were not able to work decoupled from others without the need to know and without the need to personally interact with each other. Any need to do so, e.g. to mange interfaces during integration, would hinder, if not break, the idea of a working business ecosystem, simply due to the number of possible participants required to get in contact with. For the composition of the parts, the ecosystem participants need to rely on structures and interfaces that are dealt with in a decoupled way. This thesis contributes to those structures and interfaces required to enable this vision. The following summarizes the envisioned user stories:

New Tasks - Existing Robotic System: The ability to provide a robot with new tasks by simply adding them makes the idea of a robot as a more universal tool more feasible. The user story is centered around the idea of enabling the easy integration of new tasks, which promises to bring down the costs for the modification of existing robotic systems

so that small changes in the application results in small efforts to be spent only. The user should not be forced to change or understand the internals of neither the task nor the capability blocks. The user story is driven by the need to manage the interfaces between the tasks and the capability delivering building blocks.

Same Tasks - Modification of Existing Robotics System: The user story is about the ability to modify an existing robotic system while keeping an existing task, which would enable the user to update and grow a robotic application iteratively. It envisions a lowered hurdle to invest in robotic systems, as the robot would be able to keep up with technological changes. Therefore, the user would not get trapped into investing in a product not being updatable or requiring expensive software integration steps to do so. The user story driven by the need to manage the capabilities required by the tasks.

Transferability of Tasks to Other Robotic Systems: This user story is about the transferability of tasks, which would enable the reuse of explicated domain-specific application knowledge and thereby, it would enable faster and more economic development of robotic applications. The user story is driven by the ability to check for the compatibility of a robotic system with a given task. Compatible tasks should be composable to a robotics system without the need to change them, or to understand the internal realizations of the robotic software system.

New Capability Building Blocks - Independent of Applications: This user story is centered around the separate development of capability providing building blocks. Enabling the composition of capability delivering building blocks to systems and applications realized by tasks would enable value networks where technology experts are able to sell technology-driven building blocks, which provide the necessary interfaces to be used to realize applications. This would enable a faster and more economic development of robotic applications by reusing building blocks and explicated knowledge. Applications can be composed of building blocks without the need to incorporate a robotic technology expert, as its expertise is encapsulated within the building blocks. The user story is driven by the need to provide coordination for a composable building block. Therefore, capability delivering building blocks needs to be usable for developing robotics behaviors and applications.

New Task Building Blocks - Independent of Robot: This user story is centered around the ability to realize task building blocks that are independent of specific realizations (functionalities, capabilities, robotic systems) and enable the explication of task level application knowledge. The composability of tasks with a different realization of capability building blocks would enable systematic reuse of the tasks and thereby of the explicated knowledge. The user story is driven by the ability to realize a task without the need to bind it to any specific robot or realization of a functionality.

1.2. Research Questions and Problem Statement

The overall problem addressed in this thesis is how to develop robotic software systems, considering the tasking and coordination of robotic systems consisting of many parts that are developed by different roles. Given the tasks and the environments robots are dealing with, robotic software systems inherently reach a complexity that requires to make use of the expertise provided by different experts. In order to benefit from this distributed expertise, it is necessary to be able to develop system parts separately. This requires interfaces (ports), the separated contributors (roles) can base their work on. Their parts must be composable to systems without having to change or understand the internals of them. Such composable system parts developed by individual experts can be distributed and sold to be reused in many systems. This is an important prerequisite to the cost-efficient development of complex robotic systems and leads to system development based on the idea of system integration via composition, using closed "as is" building blocks. At the same time, a robotic system should be capable of performing different tasks and working robustly within open-ended environments; therefore, coordination of the system parts during run-time is crucial. The combination of both *composable coordination* of robotic systems is, therefore, an important topic to be dealt with and motivates this thesis and its central research question:

> *How to develop robotic software systems, considering the tasking and coordination of robotic systems comprising many parts, developed by different roles in the context of a software ecosystem?*

This central question can be refined by focusing on the different aspects involved:

> *1. How to enable the development of **composable** robotic behaviors separately from concrete functional building blocks?*

The development of robotic behaviors needs to be possible independent of functionality and technical expertise to allow the contribution of procedural domain knowledge to the ecosystem. For other ecosystem participants to gain from this, by reusing behaviors from the ecosystem, the behaviors need to be composable to robotic systems and the functionality wrapping building blocks (components). Therefore, it is required to develop robotics behaviors independent of concrete building blocks.

> *2. How to realize the **interface** between software parts providing functionality and software parts realizing the behavior of the robotic system in such a way that it supports the vision of composable coordination in a software ecosystem context?*

The coordination of functionalities encapsulated in components requires access to them. To ensure the composability and the working coordination of a composed system, this access needs to be at the right level of abstraction and to follow guiding structures to

harmonize the access. Only with a harmonized interface the separated development of behavior and functional building blocks, required for a successful business ecosystem, is possible, as the developers can rely on the interface. The interface needs to raise the level of abstraction (to services) to avoid interwoven coordination and application-specific coordination logic within functional building blocks. The interface needs to build a stable foundation, which the separated roles can rely on for a connection between behavior coordination and the functionality within the functional building blocks.

> *3. How to organize the development of the building blocks, functional and behavior, in a **development workflow**, supporting the separation and the collaboration of the different roles in a software ecosystem?*

The development of a complex software system per se is not easy. The possibility to develop the parts of a system with roles separated in time and space adds to the complexity. The development of robotic applications, including the required behavior, features entities on different abstraction levels and involves different roles contributing. To enable the composition of software parts contributed by others in a robotics business ecosystem requires structures that organize the handover between the decoupled roles. Those structures need to be chosen in such a way that they provide enough freedom not to hinder the development and carefully thought-out boundaries to enable a successful composition of parts.

> *4. How to design integrated **tools** that supports the users in developing composable robotics behaviors?*

Tools and implementations are required to make the structures usable. Without proper tools, the effort required by the roles involved to deal with the structures would be too large. The composition of the building blocks would not be feasible. Different tools and implementations are required, some of them implemented within an Integrated Development Environment (IDE), while others are best implemented on the framework level. Without the tools, the pickup of the concepts by users that make up the ecosystem is likely to be poor due to the effort necessary. Therefore, tooling is a significant prerequisite for composition and the ecosystem. Tools and implementations are further required to bind left open semantics (semantic gap) that cannot be defined in structures due to the high complexity. The users cannot be burdened with textual descriptions of the semantics, and tool support is required.

1.3. Approach and Contributions

This section summarizes the approach taken to address the problem statement and the research questions, followed by the contributions made by this thesis.

> *The approach proposed within this thesis provides structures and patterns for composable robotics behavior coordination. It bridges the gap between the development of functionalities and robotics behavior coordination. The development of applications, so far done as one piece integrating functionality and coordination, is lifted to an approach based on composition. This enables the independent development of software components for functionalities, their coordination interfaces and of mechanisms for task level behavior coordination. Therefore, the approach captures, structures and consistently transfers into a model-based form the best practices and knowledge of how to design the coordination interface for software components and their interface to a task level coordination. This is the foundation for applying Model-Driven Software Development (MDSD) to make these structures usable and to support the participants in a robotics business ecosystem in building their assets and composing them to a huge variety of service robot applications.*

The thesis contributes to the software development of robotic systems, with a focus on robotics behavior coordination. The approach makes use of the service-oriented component-based software framework SmartSoft [Sch04] as a basis. It extends and builds on top of the framework. The following lists the main contributions of this thesis:

Conceptual Contributions

- The thesis provides structures and patterns for the systematic development of robotics behavior coordination. They enable the composition of behavior models with the functionality realized within components, as well as the horizontal composition of the behavior models themselves. Thereby, the approach proposed enables the separation of roles for technology and behavior development. It presents each roll with a consistent view of the parts they are concerned with while enabling the composition of their contributions in a larger setting. The design decisions for coordination taken by the individual technology experts (e.g., obj. recognition), has so far been needed to be considered during application development. Therefor the integration of functionalities to applications was expensive. Now the technology expert is presented with defined coordination structures to bind his realization to. Thereby the application development is now possible by composition.

- The thesis applies the concept of skills as a means to provide composable coordination building blocks. The skills coordinate components and provide access to the functionalities encapsulated within the components for robotics behavior coordination. They decouple realization-independent behavior models (tasks) from

concrete component realizations. Thereby, a behavior developer implements an application, precisely the tasking, and thereto refers to skills only. Thereto, there is no need to realize or even to understand how each individual software component needs to be coordinated (configured, activated, etc.). So far, the tasking of an applications has typically been developed from scratch, considering the concrete realization of all used components, which is expensive. Now it can be composed from ready made skill building blocks not being bound to specific components.

- The thesis provides a component coordination interface as an extension to the SmartMARS component model [SSS09; Ser]. The component model is extended to feature a dedicated interface for configuration, activation, asynchronous results, information query, and lifecycle access to be used for robotics behavior coordination. Thereby the interface enables the development of the skills without the need to develop a custom behavior coordination interface for each software component from scratch. The technology expert developing the skills is able to rely on the harmonized coordination interface with defined structure and semantics to realize skills.

- The thesis provides a workflow that organizes the development of a robotic system within a robotics business ecosystem. The workflow organizes the handover of the contributions of the separated participants.

Implementing and Realizing Contributions

- The thesis provides an extension to the robotics behavior coordination approach SmartTCL. The approach is extended to pick up the proposed structures. This enables the development of robotics behavior models using SmartTCL with separated roles in an ecosystem context. The thesis further provides tooling support for the extended language. Within the SmartMDSD Toolchain, the developers are supported by various tools such as model checks, syntax highlights, auto-completion, etc.

- The thesis provides a run-time interface to make use of skill-level robotics models by different systems and coordination approaches. This allows the usage of the skills by coordination approaches without the need to adopt the full MDSD development stack.

- The thesis provides several DSLs realized within the SmartMDSD Toolchain, all related to some aspect of robotics behavior development. This ranges from the definition of the skill and task types as domain-specific knowledge, to a DSL for the modeling of the configuration options of software components.

- The thesis provides tooling integrated into the IDE SmartMDSD Toolchain to support the different roles involved in robotics behavior coordination. The tooling

has been published with an open-source license and is used for the development of robotic systems in academia for demos as well as in industry for product development.

1.4. Outline

The remainder of the thesis is structured as follows:

Chapter 2 provides a short definition of the most important terms used throughout the thesis.

Chapter 3 relates this thesis to other selected work in robotics software engineering and behavior coordination.

Method and contribution:

Chapter 4 describes the envisioned ecosystem and composition approach, and it provides an overview on the overall approach and the patterns for robotics behavior coordination.

Chapter 5 provides the full concept of skills and coordination modules, including the related meta-models.

Chapter 6 provides the full concept of the component coordination interface, the related meta-models.

Chapter 7 closes the bracket opened by Chapter 4, providing the workflow structuring the development of robotics applications using functional and robotics behavior building blocks within an ecosystem context.

Results and closing:

Chapter 8 provides experiments and evaluation of the approach.

Chapter 9 concludes the thesis, summarizing the contributions and sketches open ends to continue with possible future work.

Appendix:

Provides further technical and realization relevant details to skills and the component coordination interface, as well as further technical experiments.

1.5. 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 (Aug. 2016), pp. 3–19. ISSN: 2035-3928.
  URL: https://www.researchgate.net/publication/305723618_The_SmartMDSD_Toolchain_An_Integrated_MDSD_Workflow_and_Integrated_Development_Environment_IDE_for_Robotics_Software.

- 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: A Robotics Perspective". In: *International Journal of Information System Modeling and Design (IJISMD)* 5.3 (2014), pp. 55–74.
  DOI: 10.4018/ijismd.2014070103.

Book Chapters

- Christian Schlegel, Alex Lotz, **Matthias Lutz**, and Dennis Stampfer. "Composition, Separation of Roles and Model-Driven Approaches as Enabler of a Robotics Software Ecosystem". In: *Software Engineering for Robotics*. Ed. by Ana Cavalcanti, Jon Timmis, Brijesh Dongol, Rob Hierons, and Jim Woodcock. Springer Nature Switzerland, 2021. Chap. 3

- Christian Schlegel, Dennis Stampfer, Alex Lotz, and **Matthias Lutz**. "Robot Programming". In: *Mechatronics and Robotics: New Trends and Challenges*. Ed. by Marina Indri and Roberto Oboe. CRC Press, 2020. Chap. 8. ISBN: 9780429347474.
  DOI: 10.1201/9780429347474

- **Matthias Lutz**, J. Inglés-Romero, Dennis Stampfer, Alex Lotz, Cristina Vicente-Chicote, and Christian Schlegel. "Managing Variability as a Means to Promote Composability: A Robotics Perspective". In: *New Perspectives on Information Systems Modeling and Design*. Ed. by António Miguel Rosado da Cruz and Maria Estrela Ferreira da Cruz. IGI Global, Nov. 2019, pp. 274–295. ISBN: 978-1-52-257271-8. DOI: 10.4018/978-1-5225-7271-8.ch012

Conference Contributions

- Matthias Rollenhagen, **Matthias Lutz**, Nayabrasul Shaik, Kevin Andrews, Sebastian Steinau, Manfred Reichert, and Christian Schlegel. "Towards Flexible Process Automation An Approach for Flexible Service Robot Adaptation and Allocation". In: *Proceedings of the 2019 3rd International Symposium on Computer Science and Intelligent Control*. Amsterdam, Netherlands, Sept. 2019.
DOI: 10.1145/3386164.3387292.
URL: https://doi.org/10.1145/3386164.3387292

- Cristina Vicente-Chicote, J. Inglés-Romero, Jesús Martínez, Dennis Stampfer, Alex Lotz, **Matthias Lutz**, and Christian Schlegel. "A Component-Based and Model-Driven Approach to Deal with Non-Functional Properties through Global QoS Metrics". In: *Proceedings of the ModComp'18 – 5th International Workshop on Interplay of Model-Driven and Component-Based Software Engineering (in conjunction with ACM/IEEE 21st Int. Conf. on Model Driven Engineering Languages and Systems (MODELS))*. Copenhagen, Denmark, Oct. 2018

- 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: *Proceedings of the 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

- **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: *Proceedings of the 2016 IEEE 21st International Conference on Emerging Technologies and Factory Automation (ETFA)*. Berlin, German, Sept. 2016, pp. 1–4.
DOI: 10.1109/ETFA.2016.7733602.
URL: http://ieeexplore.ieee.org/document/7733602/

- 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: *Proceedings of the 6th*

International Workshop on Domain-Specific Languages and Models for Robotic Systems (DSLRob 2015). Hamburg, Germany, 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: *Proceedings of the 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: *Proceedings of the 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: *Proceedings of the IEEE International Conference on Robotics and Automation 2013 (ICRA)*. Karlsruhe, Germany, May 2013, pp. 4244–4249.
DOI: 10.1109/ICRA.2013.6631177.

Other Publications

Other publications that disseminate the research conducted in this thesis:

- Zeljko Loncaric, Christian Schlegel, and **Matthias Lutz**. "Module für autonome kooperative und kollaborative Roboter." In: *Elektronik Praxis – Embedded System Development + IOT II* (Sept. 2018), pp. 42–44.
URL: https://www.elektronikpraxis.vogel.de/wie-universelle-module-die-entwicklung-von-robotern-beschleunigen-a-746387/.

- 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 published on http://www.robmosys.eu/wiki/.

2. Terminology

In this chapter, some of the central terms used throughout the thesis are defined. Many of the terms are used differently in different contexts. The following definitions are meant to be understood in the context of software development for service robots and the coordination of such systems.

Application An *application* in the context of this thesis is the usage of a robotic system in a concrete setting for a specific purpose - e.g., butler robot that serves beverages. An *application*, therefore, includes all parts of a robotic system, including the software parts realizing the robotics behavior coordination that could be specific to the concrete application.

Behavior In general, it describes how something is behaving or acting, typically being observed from the outside. *Behavior* in the context of a robotic system or any complex (e.g., cyber-physical) system can be seen on many different levels as the system consists of many parts. The *behavior* of a robotic system, without specifying which part of the system is meant, describes how the robotic system acts as a whole unit. In consequence, robotics behavior development deals with the development of software that organizes at a high level of abstraction how the whole robotic system acts, see *Robotics Behavior Coordination*.

Component A component is a unit that provides functionality to the system through formally defined services at a certain level of abstraction (cf. Szyperski [Szy02]). A robotics software component extends the general definition; it is the building block encapsulating functionalities required to realize a robotic system. The robotics software component raises the level of abstraction from the functional level to the service level, that hides implementation details, enforces loose coupling and restricts the sphere of influence of its internals. Components are mainly motivated by the need to deal with the overall system complexity, dividing the system into smaller and thus more manageable parts. In the context of this thesis, a component, in general, means a composable software component, following a well-defined model - e.g., a SmartSoft component.

Composition Composing, in general, is "to form by putting parts together" [CamDict]. *Composition* is the activity of putting together composable building blocks as they are. The term *composability* in a philosophical sense, following the definition of Gottfried Wilhelm Leibniz, gives more depth to the meaning with respect to the

relations of the parts. Leibniz defines *composability* as parts being "zusammen möglich", German for compatible with each other [KMH13]. Composability is the property of parts that are composable due to certain properties (e.g. self-contained, stateless, ...) The RobMoSys definition of *composability* is: "The ability to combine and recombine building blocks as-is into different systems for different purposes in a meaningful way." [Pro19d]. *Compositionality* extends composability, with compositionality the semantics of a compound is defined by the semantics of the parts and of the composition. *compositionality* thus is also the ability to compose parts in a methodological way in order to meet predictable functional and extra-functional requirements of the compound c.f. [Pro19d].

Coordination The term *coordination* has many different definitions. However, most of us have an intuitive sense of what *coordination* means. The Cambridge Academic Content Dictionary provides the definition of coordination being "the activity of organizing separate things so that they work together" [CamDict]. An important aspect of coordination is the management of dependencies. Therefore, the definition given by Malone and Crownstone in a survey about coordination theory is given in the following: "Coordination is managing dependencies between activities" [MC94]. This is also very elegant. Without dependencies, there is nothing to coordinate. While the word *coordination* can be used in robotics system development in many places, *coordination* in this thesis is the activity of organizing the software components during run-time in such a way that they work together as intended. The related adjective *coordinatable* describes the property of a component that it accepts coordination.

Domain-Specific Language (DSL) "DSLs are languages tailored to a specific application domain. They offer substantial gains in expressiveness and ease of use compared with general-purpose programming languages in their domain of application." [MHS05]. The presented approach uses DSLs as modeling tools.

Ecosystem The term *ecosystem* originates from ecology. Within an *ecosystem* entities live in relation to each other and benefit from this relation. This definition can be applied analogously to the software business ecosystem envisioned in this thesis. It is a collaboration model (c.f. [BB10]) in which the participants benefit from the interactions and contributions, sharing risks, costs, and opportunities around a domain or product.

Freedom from choice Principle which postulates deliberately restricting the options to choose from in order to reduce unnecessary degrees of freedom, see [Lee10]. This principle is applied throughout this thesis, introducing guiding structures that support the separation of roles and enable composition.

Job A *job* expresses something to be done, e.g., by a robotic system. While a *task* encodes the logic of how something is done, a job defines what to do. Connecting

job and tasks binds the robotics-specific realization of how something is done to an external (with respect to the robotics system) representation of what is to be done.

Orchestration *Orchestration*, with its root in the orchestration of an orchestra (music), is often confused with coordination and choreography. In the SOA community, some distinguish them by the concept of a central coordinator (c.f. [Lub08]) or a defined control hierarchy, which fits the usage in this work quite well. In SOA, service orchestration is used to compose a new high-lever coordinating service following a business logic [Erl07]. *Orchestration* controls the interaction of the services, whereas with choreography the services work independently and organize their collaboration. The sequencer component, for example, *orchestrates* the components of a system, itself being a part of the system (intra-organization). For the scope of this work, *orchestration* can be seen as a synonym to coordination.

Robotics behavior coordination The software of a robotic system that realizes the logic specific to the concrete task a robot as a whole is performing at a high abstraction level. *Robotics Behavior Coordination* makes use of the functionality realized in multiple software parts of the robotic software system and coordinates them to achieve the defined overall behavior.

Robotic system In general, a robot consists of the sum of all parts. In this thesis, a *robotic system* is used as a synonym to the robotic software system, which means that a robot consists of the sum of all software components.

Robot tasking The tasking of a robotic system, follows the idea that a robot is a versatile machine that is able to perform different things or tasks. The *tasking* of a robotic system, therefore, makes use of the functional parts of the robotic system to perform a specific task, see *Robotics Behavior Coordination*.

Skill A *skill* represents a capability that is provided by one or a set of building blocks of a robotic system. In contrast to the tasks, skills are more focused on individual parts composed to a robotic system and not on the overall system. While tasks describe a concept of how something is performed in an abstract top-down way, skills provide the realization of functionalities to the task bottom up. A skill provides a generic functionality to be used by robotics behavior tasks and realizes specific coordination actions towards components. Therefore, skills lift the level of abstraction from concrete and individual configurations of functionalities and services - e.g., robot movement or navigation - to a more abstract level where capabilities are named in a way independent of their implementation, thereby providing access to the functionality for robotics behavior coordination on the task abstraction level.

SmartSoft SmartSoft is an umbrella term for concepts, principles, tools, and content that are developed at the Service Robotics Research Center Ulm (Service Robotics

Ulm), located at the Technische Hochschule Ulm, Germany. The core concept has been proposed by Schlegel [Sch04]. The work presented in this thesis contributes to the SmartSoft World and to the RobMoSys structures.

Task A *task* represents an abstract encoding of something that a robotic system is able to perform. A coffee preparation task, for example, describes how to prepare a cup of coffee, which involves the following ordered steps: fetching a cup, loading the cup into the coffee machine, activating the coffee machine, and unloading the cup from the coffee machine again. Tasks can be composed hierarchically. For example, a task could be broken down into more fine granular tasks. Fetching a cup might be broken down into approaching the cup's location, recognizing and locating the cup, and grasping it. In contrast to a skill, a task does not link directly to functionalities and therefore is an abstract representation of how to perform something. Tasks make use of skills to bind to the functionalities required to perform a task. In the case of the coffee example, the skill to recognize a coffee cup is used. How the skill is implemented - for example, using a neuronal network or some model matching - is independent of the task using the skill. By encoding how to perform a certain task at a high level of abstraction, domain experts are able to use tasks to express their domain knowledge of how certain tasks can be accomplished. Logistics experts, for example, can encode a task that represents a robotic order-picking activity without the need to specify the concrete approach taken or the used functionalities and technologies.

3. Related Work

This chapter presents selected work related to software development in robotics in the context of robotics software business ecosystems in general and application development and behavior coordination of robotic systems in particular.

3.1. Software Business Ecosystem

The term "ecosystem" originates from ecology; within an ecosystem, entities live in relation to each other and benefit from this relation. "Business ecosystems" as a business model was introduced by Moore [Moo93]. Bosch describes a business ecosystem as a collaboration model (c.f. [BB10]). The participants benefit from the interactions and contributions, sharing risks, costs, and opportunities around a domain or product. Mapped to software engineering, Bosch [BB10] describes software ecosystems as the next extension step for software platforms. In more recent work, Bosch describes how software ecosystems relate to the digitization era [Bos19]. Bosch implies that digitization disrupts value chains in existing business ecosystems, shifting from products to services, where users rent services instead of buying products. This results in products being longer in the field with more continuous development to improve the services of the products, especially new software [OB20]. This again drives the need for more flexible and changeable software systems being developed and deployed quickly. With "Robotics-as-a Service" (RaaS), the shift from robotic products to services is a growing opportunity [Mar19]. Cloud services such as Amazon AWS RoboMaker, the Google Cloud Robotics Platform, or the Honda RaaS platform benefit from developing (Artificial Intelligence (A.I.)) services used with RaaS such as analytics, recognition, etc.

The smartphone domain is an excellent example for the class of ecosystems driven by keystone-players [IL04]. Apple and Google dominate this market and operate two large ecosystems. In contrast to the robotics ecosystem vision of this thesis, apps in the smartphone ecosystem are mostly isolated with little or no interaction among each other. The apps rely on the common central infrastructure only, driven by keystone players.

The Debian operating system community, for example, is a community-driven ecosystem [Jan12]. It is based on structures, infrastructures, and governance for content and processes, driven and developed by the community in an open and democratic process. In contrast to the smartphone domain, it is an excellent example of loosely coupled contributors and contributions based on the common defined structures. Within the Debian ecosystem many technical artifacts are related or dependent on each other.

ROS, as the most used robotics framework and platform, is arguably a robotics ecosystem. The step towards a successful business ecosystem is, however, obstructed by the lack of guiding structures. The lack of those structures leads to not composable building blocks and prevents their separated development. The composition of ROS nodes to systems in a correct-by-construction way is not possible due to the dependence on internals and conventions realized in individual ROS nodes [Ham+19b]. Thus, combining ROS nodes to systems is often done in a cumbersome trial-and-error way [Ham+19b]. The ongoing development of ROS2 tries to introduce some of the missing structures. The incompatibility to ROS1 leads to a split in the community and the ecosystem.

Many enterprises in robotics start to push an ecosystem centered around their products (UR, MIR, FRANKA, etc.). Iansiti and Levien [IL04] describe this as a keystone-player ecosystem. The UR+ program of Universal Robots (UR) is a perfect example of such an ecosystem. All participants in the ecosystem are related to the anchor product, in this case, a robotic arm. The ecosystem driver defines the structures and interfaces the participants have to adhere to, in this case, technically a java plugin structure to integrate into the UR software framework. In contrast to the ecosystem envisioned in this thesis, the software blocks are integrated with the core software structure, not with other software blocks. The plugins are tightly bound to the specific product and cannot be used with manipulators from other manufactures.

Stampfer argues in [Sta18] the necessity for structures to establish robotics business ecosystems. He proposed a meta-structure for robotics ecosystems, defining tiers of contributions. The approach contributed to the RobMoSys structures. The structures introduced by the approach presented in this thesis are compatible and extend those.

3.2. Robotics Behavior Coordination

Coordination, as defined by Cambridge Academic Content Dictionary, is “the activity of organizing separate things so that they work together” [CamDict]. Coordination can be used in robotics system development in many places; in this thesis, coordination is the activity of organizing the software components during run-time in such a way that they work together as intended.

In general, coordination in software engineering has a background with a long history. Malone and Crowston define coordination in a survey about coordination theory as follows: “Coordination is managing dependencies between activities.” [MC94]. Robotics behavior coordination narrows the scope to coordinating robotic software system parts to achieve a dedicated goal. Robotics behavior coordination realizes the tasking of a robotic system. This thesis is not contributing a new robotics coordination approach but is bridging the gap between robotics behavior coordination and software building blocks in the context of robotics business ecosystems and the composition of building blocks to applications. This thesis proposes a structured approach to achieve composition of building blocks, realizing functionality (components), and realizing

coordination. Thereby, the thesis defines structures that enable composable coordination and separation of roles and concerns as a foundation for a robotics business ecosystem.

Numerous different approaches to robotics behavior coordination exist. The following presents particularly noticeable approaches from the huge body of available work.

Statecharts proposed by Harel in [Har87] as a visual formalism for state machines are the foundation for a number of robotics behavior coordination approaches. A wide range of industry-grade tools for modeling and developing state machines with statecharts are available such as Rhapsody, Matlabs Stateflow, Labview, or Yakindu. With RTT-Lua [KSB10] Orocos offers a simple scripting language for the tasking using Restricted Finite State Machines (rFSM) statecharts. These are able to access the Orocos RTT bindings to communicate with the components. ArmarX, as an event-driven component-based robot software development environment, features its own statechart implementation [Wäc+16]. The approach is similar to rFSM, does, however, not focus on the coordination of components, but on the disclosure of the internal state of components.

Dynamic State Charts [SS13; SS14] address the reuse and composition of behavior blocks. The connection to the components with their functionalities is realized using the SmartSoft communication patterns. SMACH [BC11] is a python based hierarchical state machine with tight integration into ROS. The tooling includes a view for run-time introspection of the modeled state machines. ROSco [Ngu+13] and RAFCON [Bru+17] are two further ROS integrated approaches providing graphical user interfaces. They use skills developed by experts to develop state machines. The Skill Hybrid State Machine (SHSM) [NFL10] realized in Lua, features a hierarchical composition of hybrid state machines, their development was driven by RoboCup challenges using the robot Nao.

Another class of approaches is based on task trees. The RAP [Fir89] system is one of the first task tree robotics behavior coordination approaches. It implements situation-driven execution in 3-tier robotics architecture. RPL [Mcd93] as a predecessor of RAP supports concurrency and is implemented as a Lisp internal DSL. TDL [SA98] makes use of code generation to enable implementation of task tree nodes in c++. More recent implementations of tree based approaches are often based on the concept of behavior trees (BT). BT originate from the computer game domain, Michael Mateas and Andrew Stern [MS02], and Damian Isla [Isl05] provided the first key contributions to the concept. Current implementations of the approach are, for example, py_trees [Pytrees], ROS-Behavior-Tree [CÖ17] or BehaviorTree.CPP [Faca]. SKIROS and SKIROS2 [Rov+17] provide a platform to build robotics behaviors based on high-level skills on top of ROS and make use of planners and behavior trees for execution.

Iovino et al. (preprint, KTH Lab, Patric Jensfelt) [Iov+20] presents an extensive survey on BTs in robotics and AI, also providing an overview on current implementations.

SmartTCL [SS10] is implemented as a Lisp internal DSL and is motivated by the RAP system. SmartTCL supports the dynamic expansion of a task tree during run-time. SmartTCL is used as coordination approach to demonstrate parts of the approach presented in this thesis, as it is a suitable tool for the development of complex robotics applications (see Chapter 8). The language has been extended to use the proposed struc-

tures to coordinate the components via the proposed component coordination interfaces and the coordination modules. The approach proposed by this thesis is, however, not limited to SmartTCL and can be realized with other coordination approaches as well, as is demonstrated in the experiment Chapter A.4.1. Neither the proposed coordination interface, nor the concept of coordination modules or skills are limited to SmartTCL.

3.3. Software Engineering

3.3.1. Component-Based Software Engineering

In the context of this work Component-Based Software Engineering (CBSE) is applied. Components are entities that can be coordinated. They are the means to encapsulate the functionality required to realize robotic systems and applications. CBSE structures systems partitions them such that complexity can be dealt with and allows for the development of reusable coarse granular blocks [Szy02; Bro+98; CSS11; Frö02].

CBSE supports the development of building block-like entities where the mutual sphere of influence between them is decoupled. While other levels of granularity, such as software modules or software libraries, also enable reuse, none of them address it to an extend enabled by composition if combined with coarse granular interfaces (see Service-Oriented Architecture (SOA)). Composing loosely coupled software parts without knowing or changing the internals is a significant property of software components. The coarse granularity CBSE offers, reduces the required complexity within robotics behavior coordination, as fewer entities with fewer interfaces need to be dealt with.

CBSE can also be considered to be the state of the art development approach in robotics, see [BS09; BS10]. These paper also provide an overview on CBSE in robotics. Many robotic frameworks (e.g., ROS, Orocos, or SmartSoft) use software components to encapsulate implemented functionality.

For coordination of components, as is the main focus of this thesis, the components need to be organized as decoupled entities. The coordination of functional parts that are tightly coupled and interwoven with each other is much more complicated. The borders and interfaces of such interwoven entities are arbitrary. The sphere of influence a coordination action would have would be difficult to determine. This motivates the need for decoupled entities, i.e. software components.

3.3.2. Service-Oriented Architecture

Within SOAs [Erl07; SW04] services are the key elements, enabling reuse, loose coupling and abstraction. These properties are exactly those needed to enable composition of building blocks in an ecosystem. CBSE in combination with services enables the decoupling of the software components, with the interfaces at a service level granularity. For system-wide coordination of software components, as done when robotics behaviors realizing the tasking of the robot is developed, the components' communication services

are, however, secondary. For behavior development, the components are the entities of primary interest as they provide the functionality used for the tasking of the robot. However, CBSE without the right level of interfaces can lead to tightly coupled software components, hindering their composition, which would break the ecosystem idea.

One central point in choosing the functionality wrapping technology, is the right level of abstraction. As there is a significant change in abstraction between those levels that realize the functionalities and the behavior levels responsible for coordinating the others, the separation between those levels is reasonable. A high level of abstraction and coarse granularity of the components supports this separation. Therefore, CBSE combined with the service level interfaces are used in the here presented approach to encapsulate the functionalities. The combination of CBSE and services in robotics is uncommon, SmartSoft is an exception to this. In [Sta18] Stampfer proposed a meta-structure for system composition based on CBSE and SOA. The approach utilizes SmartSoft as technical foundation and applies MDSD on top of the SmartSoft framework to enable the composition of components in the context of robotics ecosystems.

Complex applications realized using a SOA often also make use of layered architectures, including an orchestration layer, similar to a robotics control architecture. Different modeling tools and languages can be used to express the business logic to be executed by the orchestration layer, e.g. IBMs WebSphere Business Modeler. The languages to express how to orchestrate the services and thereby the business logic evolved with the used service technology. Two of the major approaches used are the Web Services Flow Language WSFL [Ley01] and Web services Business Process Execution Language (WS-BPEL) [OAS07]. Both are used to model the orchestration of web services.

Microservices as a variant of SOAs are an architectural style without a fixed definition [FOW14]. The idea behind microservices is to provide independent (out-of-process) and easy to deploy services to build applications with, in contrast to monolithic applications. Lewis and Fowler describe microservices as follows: “In short, the microservice architectural style is an approach to developing a single application as a suite of small services, each running in its own process and communicating with lightweight mechanisms, often an HTTP resource API. These services are built around business capabilities and independently deployable by fully automated deployment machinery. There is a bare minimum of centralized management of these services, which may be written in different programming languages and use different data storage technologies.” [FOW14]. The communication between the services is typically realized using REST-based APIs [VKG17]. Applications realized using microservices are typically coordinated in a choreograph style using RESTish protocols [FOW14]. This is in contrast to the orchestration by a centralized entity, as typically done in robotic systems.

Representational State Transfer (REST) as architecture style is a realization for SOAs and is typically used to connect web services. REST introduced by Fielding in [Fie00] itself is not a standard. The realization of REST based APIs uses standards such as HTTP or JSON. However, it resulted in a de facto standard that is widely used. In contrast to the Simple Object Access Protocol (SOAP), REST is based on a state-less client-server

architecture, which aims for a simple and uniform interface. SOAP as the second major technology to realize SOAs, in contrast to REST, is standardised and defines an protocol.

While there are many different implementations of SOAs, the combination of CBSE and SOA is the basis this work rests on. This combination tackles both the management of the complexity necessary to realize complex robotic systems and the decoupled interfaces necessary for composition in an ecosystem context. Components additionally provide a coarse granular execution container which eases the coordination of the encapsulated functionalities. The application of an existing generic SOA implementation, e.g., REST or SOAP, without any tailoring to the robotics domain, would confront the robotics technology experts with having to deal with a communication mechanism not tailored to their needs. This would inevitably lead to incompatible implementations and break the ecosystem vision. Thereby, the SmartSoft with its communication patterns, is a good foundation for the realization of the approach presented in this thesis.

3.3.3. Orchestration and Choreography

Orchestration and choreography in the context of software engineering typically describe the automated configuration, coordination, and the management of running software entities [Erl05]. With SOAs, the orchestration of services is used to compose a new high-lever service following a business logic [Erl07]. Thereby, a centralized orchestrator uses the model's business logic to reuse existing services to create a new intended service. Orchestration controls the interaction of the services whereas with choreography the services work independently and organize their collaboration. The WS-Choreography specification [W3C04] defines interfaces and languages for service choreography. Choreography works without a centrally coordinating entity. The collaborating services need to know their part of the business logic to generate the intended new service. To enable the orchestration or choreography of SOA services realized with different types and formats, the Enterprise Service Buses (ESB) can be used for enterprise application integration.

In contrast to the orchestration or choreography with SOAs, the coordination of robotic system-parts aims at performing something (e.g., cooking coffee) with the system and not to create new services.

3.4. Model-Driven Software Development (MDSD)

Model-driven software development (MDSD) as technology is centered around the modeling of decisive aspects. Thereby, models increase the level of abstraction compared to source code, leading to a reduced complexity [BCW12] and lowered development efforts [Völ11]. The thesis' approach utilizes MDSD to explicate structures in meta-models. The meta-models are the foundation for role-specific tools. Those dedicated tools make the knowledge and structures explicated in the meta-models applicable to the

user without the need to understand them. The "Robotics 2020 Multi-Annual Roadmap" of the European Commission defines MDSD as a key to robotics software engineering: "in order to achieve a separation of roles in the robotics domain while also improving composability, system integration and addressing non-functional properties" [euR16].

A key element to realize MDSD are Domain-Specific Languages (DSLs). In contrast to general-purpose modeling tools, DSLs are focused on a specific problem or domain. They are a powerful method to provide dedicated tooling using terminology the users are familiar with, thereby simplifying the users' problem. The thesis realizes several DSLs, using the Eclipse ecosystem tools of the Eclipse Modeling Framework (EMF) [Foua], such as Ecore, Xtend, Xtext, Sirius, etc. Besides the Eclipse ecosystem, other major MDSD-suites are available. The JetBrains Meta Programming System (MPS) [Jet16] and MontiCore Language Workbench [Sof] are other notable approaches.

3.5. Model-Driven Software Development (MDSD) and Domain-Specific Languages (DSLs) in Robotics

Several projects addressed MDSD and DSLs in robotics, some particularly noteworthy approaches in robotics are RTC, RobotML, ROSMOD, BRICS, and ReApp. Their relation to robotics behavior coordination is discussed in more detail in Section 3.6.1 and 3.6.2. Nordmann et.al. present an extensive survey on DSLs in robotics [Nor+16].

The MontiArch language framework has been used to develop language families for robotics development [Ada+17]. In contrast to the here proposed work, those approaches focus on the upper system levels (MISSION and TASK). They assume the existence of a capable robotics platform, providing skill and task-level building blocks. With MontiArc they developed DSLs for different roles, for task and mission level modeling, for tasks, goals, world properties, robot capabilities (without the connections to the functionalities), etc., for different roles [Ada+17]. For execution, they transform those models into PDDL format and ask a symbolic level planner for a solution, which is then handed over to a linear execution using the skills provided by a capable robotics platform [Ada+16]. They propose a separation of roles for upper-level models to reuse them by different roles – taking into account an abstraction level from skill upwards only. Within [Ada+16] Adam et al. propose a service robotics reference architecture, which is primarily focused on the abstraction level of task and mission planning. The control architecture they propose suffers from the same problem as a classic SPA (sense plan act) control architecture. Namely, a linear execution of plans not considering uncertainties, unpredictability, and deviations from plan executions.

Multiple approaches target DSLs for manipulation tasks, [WDW20; Näg+18]. Nordmann et al. propose in [NLM21] the usage of system modes to abstract runtime state information and reconfiguration of components. The approach is integrated into the MROS modeling tool [Boz+21]. The approach has been developed in the context of RobMoSys and picks up structures such as the skill definition and realization of RobMoSys

which has been substantially shaped by the contributions outlined in this thesis.

3.6. Software Development in Robotics

3.6.1. Robotics Frameworks

Research in software development for robotics has spawned various robotics frameworks. The following relates selected robotics frameworks to this thesis. A comprehensive literature review on robotics frameworks can be found in [ES12].

The robotics framework Player [GVH03] has been an important early robotics software framework. It reached wide acceptance in academia, supporting many robotic devices [Big+13]. The combination with the simulator Stage and the still maintained simulator Gazebo added to the framework's popularity.

Orca [Bro+07] is one of the early component-based robotics frameworks, targeting the reuse of software. Orca, in contrast to Player, used existing communication middleware technology ICE. The inter-component communication is realized using remote method invocations. No dedicated interfaces or structures for coordination are provided.

The Orocos Real-Time Toolkit [OroRTT] realizes a framework for robotics control systems. It supports the development of real-time applications and has a focus on manipulation. Some tooling and basic code generation for Orocos is available. For coordination of components, Orocos uses operations to be called from other components, synchronously or asynchronously. The components further provide properties via a configuration interface. Properties contain data that can be changed by other components. With RTT-Lua [KSB10] Orocos offers a simple scripting language for the tasking using rFSM statecharts [HN96], which can access the RTT bindings to communicate with the components. The component model of Orocos features a common life cycle. The concept of services in Orocos can be used to extend components, which can be used to add RTT-Lua in the context of an Orocos component. Thereby, the coordination of components can be realized closely to the components themselves. This could be used to realize skills for robotics behavior coordination, similar to the skills proposed in this thesis. The authors of Orocos, however, describe the usage for the coordination of a single component only [Soe+]. Orocos does not provide domain models to capture the domain structures other than the message types being communicated.

YARP (Yet Another Robot Platform) [MFN06] is a lightweight middleware mostly used with humanoid robots. The authors describe YARP as "YARP is plumbing for robot software" [MFN06]. Consequently, YARP solely offers communication mechanisms for streaming and RPC-like communication. It does not provide the user with structures or tools for coordination. In the context of the RobMoSys Integrated Technical Project (ITP) CARVE [Pro19a], the interaction of YARP with SmartSoft and the SmartMDSD Toolchain has been demonstrated.

Fawkes is a component-based framework robotic software framework [Nie+10] using a plugin infrastructure to realize components in threads. The communication between

components is based on a hybrid blackboard/messaging approach. The messages communicated are defined within XML files and do not provide the level of abstraction of services. This results in tightly coupled components in systems. The blackboard-based communication allows for one to many (publish-subscribe) communication only. No further communication semantics is enforced. Network communication is directly mapped to TCP or UDP sockets, without the usage of any communication middleware. The components in Fawkes can be executed in a synchronized, ordered way, from an application's main loop, in contrast to most other CBSE robotics frameworks, that typically execute the components as own active entities (process or threads). The Fawkes framework is designed to work with a 3-tier system architecture. The sequencing layer is based on extended Hybrid State Machines (HSMs) [Hen96] that are implemented using Lua [IDC96]. This Lua-based Behavior Engine [Nie09] has been ported to ROS [Sri+12]. The execution on the deliberative layer is performed in sequence with the main loop execution step. Fawkes does not define a coordination interface for the components. The behavior is separated into skills and higher-level behavior blocks. In contrast to the skills proposed in this thesis, the skills in Fawkes are at a much lower abstraction level, e.g., controlling velocities of a mobile base for navigation. The missing coordination interface for the Fawkes components introduces the risk of pushing high frequent control loops upwards to the deliberation layer. The ROS realization of the behavior engine connects towards the functionalities within the components using the ROS actionlib. This at least avoids accessing low-level sensor data communicated via pub/sub and thus avoids pushing high-frequency control loops up into the deliberation layer.

The Rock framework [Rock; JA11] is component-based and utilizes the Orocos-RTT component model. Rock uses oroGen [FHC97] code generation to generate C++ code for the component implementation based on the specified interfaces. The functionalities realized in Rock are implemented within libraries to be used in components, which encourages the separation of functionality from glue code specific to the robotics framework. System architecture wise, Rock proposes a plan-based architecture with a central coordination component called plan manager. The behavior coordination itself is realized based on tasks, modeled in an embedded DSL [JKL10] running on Ruby. The coordinating connection to the components is based on the Orocos tasks, implemented within components. The tasks can be started and stoped and make use of the Orocos properties for configuration.

OpenRTM-aist/RT-Middleware [And+05b] is the CORBA based reference implementation of the OMG Robot Technology Component RTC [And+05a] standard. RT-Middleware featured a simple data-flow-like communication and an Eclipse-based IDE. The component model defines a dedicated interface for component run-time configuration. RT-Middleware as RTC implementation is one of the early robotics component models defined. RTC as an approach for a component model is outdated by more recent approaches such as BRICS, RobotML, or SmartSoft.

Microsoft Robotics Developer Studio (MRDS) was a short-lived but worth mentioning programming environment, given the size of the developing company. Based on the

.NET platform, it includes a REST-style service-oriented run-time. For robotics behavior coordination, the MRDS included the data flow-based Visual Programming Language.

ROS, as a collection of tools, libraries, and a communication framework, is currently by far the most well-known and widespread adopted robotics platform. The robotics community's uptake led to a unification in robotics software development, at least to a certain extent. The ROS node as a component and ROS tools lead to an ecosystem, making many functionalities accessible for robotic application development. ROS was intended for research by academic institutions [Ger14] and is deliberately developed not to provide guiding structures or patterns for software system engineering, but to offer the maximum flexibility [Ger15]. The framework's design philosophy is an example of "freedom of choice" contrary to "freedom from choice" [Cou+10]. The architectural design decisions taken by the framework experts and the lack of structures leads to the problem that each developer is on its own, to either reinvent design solutions or to check what preferred design is realized by similar or known open source components. This inevitably leads to conflicting and incompatible building blocks, which can not be composed to systems easily. Documentation instead of convention and structures is of little help, as there is still the need to understand the source code of other components or, even worse, to change them to get them compatible. ROS does not provide sufficient structures to support the separation of roles and concerns required to get a working robotics business ecosystem with composable building blocks. ROS2 addresses this lack to some extent and provides some guiding structures. However, ROS2 still aims to maintain the flexibility of ROS [Ger15]. Tooling-wise, there exist some simple IDE approaches. However, it is challenging to come up with an expressive component (meta-)model onto which MDSD-tools can be built if proper structures are missing.

Different robotics behavior coordination approaches have been applied to ROS. Hierarchical finite state machine such as SMACH [BC11] or ROSco [Ngu+13], or behavior tree-based approaches like py_trees [Pytrees] or ROS-Behavior-Tree [CÖ17]. All behavior coordination approaches need to connect to ROS nodes to orchestrate the system. ROS is not defining a dedicated component coordination interface for that purpose. Therefore, the realization of the coordination access to the components for system orchestration is up to the component developer. The communication interfaces offered by the ROS framework can be used in a broad and generic way. As there is no defined structure for the coordination, no semantics for the access to the wrapped functionality can be defined, such as activation, life-cycle connection, etc. For the configuration of the components, ROS does, however, provide a parameter server, which can be used for run-time parameterization as well. The parameters are defined within a global namespace in an ad-hoc manner and do not follow a domain-specific type definition. This is again in line with the design decision to offer maximum flexibility instead of providing guiding structures. Besides the parametrization, the components' coordination is typically realized using ROS services, a RPC-like communication mechanism [Ros]. RPC-like communication in component-based systems with many components with different communication semantics is very error-prone and leads to tight coupling between the components. This

makes, e.g., coordination of components via RPC difficult. This is in particular true for long-running activities because their results or messages needs to be considered asynchronously. The actionlib [MPA] as an extension to ROS adds some communication semantics to the RPC-like services by introducing a protocol and API for asynchronous communication. The actionlib is typically used for coordination access to the component as is done with SkiROS [Rov+17], for example. However, the actionlib is not explicitly designed for coordination and is also used for different purposes, e.g., to help realizing low-level PID controllers [San+17]. Therefore, the development of robotics behavior is dependent on the specific realization of the individual components and can not rely upon defined structures with dedicated semantics.

3.6.2. Tools and Software Workbenches

An increasing number of software development tools dedicated to the development of robotic software are available nowadays.

The general-purpose modeling tool Papyrus [Papyrus], based on Eclipse, has been extended in the context of RobMoSys to Papyrus4Robotics [Papy4Rob]. Papyrus4Robotics is a set of DSLs and tools to support the development of robotics software. The tooling makes use of UML/SysML and the UML profile mechanism to realize DSLs. It includes the RobotML DSL [Dho+12] which specifically addresses the modeling of mobile manipulation robotic systems. Papyrus4Robotics can be used to perform dysfunctional analysis on components and can generate code based on the RobotML-tools.

SmartMDSD Toolchain, to which this thesis contributes, is an Eclipsed-based robotics IDE. It supports the development of robotic systems and applications. The SmartMDSD Toolchain offers dedicated views and tools supporting all roles involved in the development of robotics applications. The toolchain is developed by the Service Robotics Research Center Ulm (Service Robotics Ulm). Since the first release in 2009, continuous development made the SmartMDSD Toolchain the most advanced MDSD-based robotics software engineering tool. It is available as open-source Eclipse project [Foub] under the umbrella of the Eclipse Foundation. The toolchain fully conforms to RobMoSys; its current realization primarily targets the development of SmartSoft bases systems. The toolchain does, however, also support other robotics frameworks, e.g., ROS, and reaches out to other worlds such as OPC UA. It is connected to the XITO platform and marketplace to participate in an expanding robotics business ecosystem.

While there are plenty of ROS tools, there is no ROS IDE, that seems to be more popular than using any general-purpose IDE, such as QtCreator or Eclipse [ROS11]. Simple tools or plugins exist that integrate into IDEs, such as Eclipse or CLion plugins for ROS. A comprehensive list of tools is given on the following ROS-wiki page [ROS20], including how to set up VIM or Emacs plugins for ROS. Most of those tools enable code browsing, allow for triggering the build system, and for accessing a debugger.

The RoboWare Studio based on VSCode is promoted as dedicated ROS IDE and offers essential tools such as workspace management and building support. ROS Development

Studio (RDS) as web-based IDE allows for the development and testing of ROS programs in the browser and targets the education of developers in ROS.

The Robot Operating System Model-driven development tool suite, short called ROSMOD [Kum+15], is a model-driven tool suite for ROS. It provides support for the design and deployment of ROS-based systems. ROSMOD defines its own ROS component model and provides tools for analysis and verification of threads that can occur among the components. ROSMOD does neither address separation of roles nor composability. For coordination of the components, ROSMOD features an integrated HFSM Design Studio [Emf] for the development of UML state machines based on the WebGME. Based on the HFSM, clue code can be generated to wrap the business logic [KE19]. The generated code and the realized business logic combined with the ROSMODE components result in interwoven components, neither enabling composition nor supporting separation of roles.

The BRIDE IDE of the European FP7 research project Best Practice in Robotics (BRICS) and the BRICS Open Code Repository aim at finding ROS packages in an online platform. While BRIDE supports the user in developing ROS nodes and simple coordinator nodes for robotics behavior coordination using state machines, the tools are no longer maintained. The project ReApp with the ReApp Workbench took up the BRICS component model, and some parts of the tool [Wen+16]. The ReApp Workbench [Wen+16] as tool for robotic application development is based on ROS nodes. The ROS component model is extended with an ontological classification system to simplify the discovery of components. Core parts of the workbench are based on the IDE from the Framework for Distributed Industrial Automation and Control project, an IEC 61499 implementation. The ReApp Workbench makes use of IEC 61499 and ROS to implement components. The coordination of the components can be modeled using Execution Control Charts as defined by the IEC 61499 standard. The workbench is neither addressing the separation of roles nor composition. The ReApp Workbench is not available as open-source and its status is described as prototypical only [Awa+16]. The Hyperflex Toolchain [GB14; GB11] is an extension to BRIDE IDE that allows to graphically design the software architecture of a robotics Software Product Line. The toolchain enables the development of architecture and variability models for ROS and other robotics frameworks.

The Robot Application Development and Operating Environment, as another ROS related tool, enables the graphical modeling and generation of ROS launch files [NLH15].

A very simple Ecore based meta-model for ROS to build MDE tooling upon is proposed by Garcia et al. in [Ham+19b; Ham+19a]. The idea behind the work is to model existing code artifacts to make them accessible to MDE to leverage the accessibility of ROS with its large ecosystem of existing content, while providing the advantages of MDE techniques. The tooling can analyze source code and the running instance of the ROS node to extract model relevant information to automate some parts of the modeling. The presented prototypical tooling is available as open source.

While there are many tools for robotics software development around, most of them focus on a specific aspect only. Integrated tooling, as is contributed by this thesis,

that organizes the handover between the different roles is not very common. The SmartMDSD Toolchain, to which this work contributed, is a rare exception.

3.6.3. Robotics Programming End-User Tools

An increasing number of robot programming tools aim at enabling users on the premise to develop robotic applications. Most tools make use of graphic visualizations and representations following the idea of low-code or no-code approaches. The user is able to select readily available skills (see also IEEE IES ETFA workshop series Skill-Based Systems Engineering [Kat20]) or templates to configure and combine them to runnable programs. Well-known examples for robot independent tooling are the ArtiMind Robot Programming Suite and the drag&bot tool. Many robot manufacturers also offer dedicated user tools for robot programming, such as UR, Franka Emika, or Nao with Choreograph. The approach proposed by this thesis can be applied to realize the skill building blocks the robot programming tools for end-users can make use of.

3.7. Initiatives and Projects

The majority of work in robotics research is still spent on developing algorithms, tackling technical challenges (c.f. published work on robotics conferences). However, with the "Robotics 2020 Multi-Annual Roadmap" [euR16] and the "Strategic Research Agenda" [euR13] of the European Union, and studies showing the need for software engineering in robotics such as EFFIROB [HBK11], challenges faced by software development and integration in robotics have reached more attention. The further growth of the robotics market [Mül+20], and the ever-new robotics products developed, presumably create a pull from industry to further research on software engineering for robotics.

The project EU H2020 RobMoSys created a community wide consolidation for model-driven software engineering in robotics. RobMoSys focuses on structures and meta-structures to enable a composition- and model-driven approach to robotics software system engineering. RobMoSys, together with EU H2020 ROSIN, aims for an EU digital industrial platform for robotics.

ROS-Industrial aims to bring ROS to industrial applications. The EU H2020 project ROSIN contributed to ROS-Industrial with improving the quality of existing components.

The German BWMi PAiCE SeRoNet project conforms to the RobMoSys structures and aims to ramp up a robotics building block marketplace. SeRoNet bridges to the Industry 4.0 domain, utilizing OPC UA and the concept of the asset administration shell.

The EU H2020 project ScalABLE aims to optimize and maintain production lines 'on the fly' and developed the robotics skill-based programming approach SkiROS2 as an extension to SkiROS developed in the past EU project STAMINA. The skill definitions in the manufacturing domain defined by the project can be used as domain-specific structures with the approach proposed by this thesis as well.

4. Composable Behavior Coordination in Robotic Systems

The overall scope of this thesis is to enable a systematic development of robotics behavior coordination for robotic systems, also known as the tasking of the robot.

From the perspective of developing robotics behavior coordination, a robotic software system can be roughly separated into two parts or aspects. The first part comprises the functionalities of a robotics system – e.g. navigation, device drivers, etc. Some of these functional blocks require access to hardware of the robot, while others are pure software blocks. The second part comprises the robotics behavior or tasking, using the functional blocks to build an application. The motivation for the need to coordinate a robotic system and its functional parts is manifold; it is described in more detail in the introduction of this thesis and can be briefly summarized as follows: A robot, perceived as one physical entity and controlled by many functional blocks, needs to use its functional blocks in a coordinated manner to perform different tasks. Figure 4.1 illustrates this relation showing three tasks and making coordinated use of different functionalities and hardware parts of a robot. A control hierarchy needs to be established at any point in time to ensure that all the functional blocks act in concert and contribute to the overall task the robot is performing at the time. For instance, fetching a book from the top layer of a shelf requires the manipulator, the perception system, and the rest of the robot to work together in coordination.

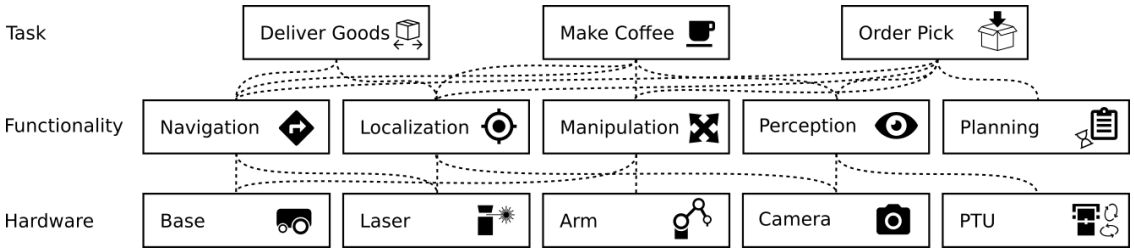

Figure 4.1.: Different tasks can be realized utilizing different functionalities of the robotic system. Coordination of those system parts is required to ensure that all parts act in concert and contribute to the realization of the tasks.

This thesis aims to bring these two separated parts or worlds (functional and behavioral) together in a way that would enable the development of composable system parts and a robotics business ecosystem. The effort currently spent on the integration

of functional building blocks and robotics behavior is a major hurdle in robotics system development. Currently, robotic systems are typically developed by applying an integration-centric approach, which holds true for both the integration of functional parts and for robotics behavior. The developed parts are integrated during the integration phase with different developers coming together, defining interfaces, and tailoring the parts to fit together to develop a working robotic system. With increasingly complex and larger systems being built involving many parts and developers, this integration-centric approach becomes too time- and cost-intensive, mainly due to the required changes of the system parts during integration and the effects caused by such changes (need to change other system parts as well). The behavior parts that realize the tasking of a robot are very often realized to be dependent on the functional parts. Thus, the reuse of domain knowledge captured within the behavior parts is limited to the same or very similar systems (defined by the functional parts).

To overcome these problems, there needs to be a shift in the used development paradigm, right from integration to composition. The different roles contributing to a robotics system need to be able to develop their parts in separation. The integration of the separately developed parts needs to be possible using composition, with no need to understand or change parts at the time of composition. This holds true for both function and behavior blocks. With blocks of both these worlds being composable within their world, a combination of the two worlds needs to be addressed. Both parts, namely functional and behavior, should come together in a composition-like manner, not changing parts over and over again while building new applications. Figure 4.2 illustrates the composition of a new localization approach to an existing system, keeping the other building blocks of the system unaffected.

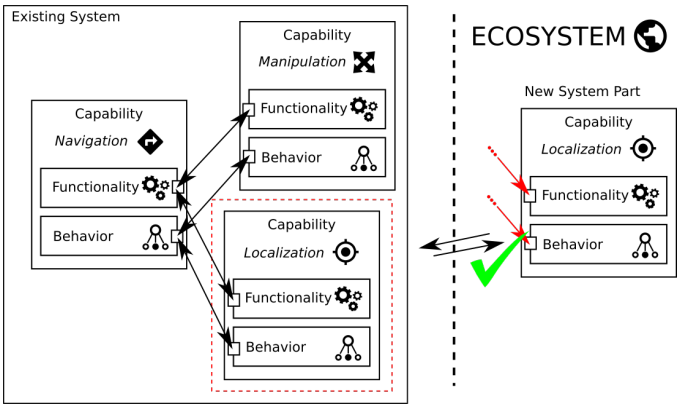

Figure 4.2.: Adding a new system part, in this example a new localization approach, to an existing system by composition. The other building blocks should be unaffected by the change due to the stable interfaces and structures decoupling the parts.

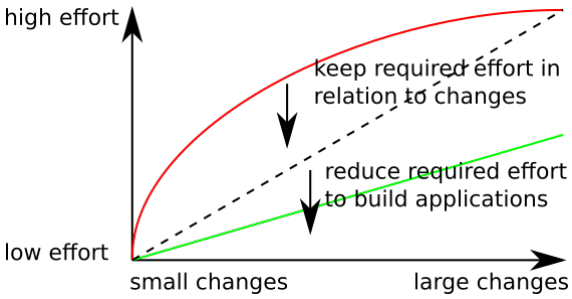

Figure 4.3.: Developing robotics applications and systems, the relation between changes in the application and the required effort to realize them. With current development approaches, even small changes in robotics applications lead to a disproportionate effort required to update the system. This hinders the introduction of new robotic applications. The ratio between effort and changes needs to be better balanced, as it is also targeted by the SeRoNet initiative [Bun17].

The ability to compose separately developed system parts is an important enabler for the vision of a working robotics business ecosystem. The goal is to have both parts (functional and behavior) available as reusable blocks within a robotics business ecosystem. The composition of new applications should be possible with ease. In current robotic systems, a slight change within the application or the system typically requires a rather large amount of effort to deal with those changes (compare Figure 4.3). By making both functional and behavioral parts composable and accessible within an ecosystem, the effort required for small changes in the applications or systems should be pushed to a reasonable balance.

The key to making those parts come together easily in a composition-like manner is the introduction of structures on both sides. Those structures should constitute stable foundations for the different roles to work on and that allow for work being done are separated in time and space. The structures build an interface between the two worlds of functional and robotics behavior. Care has to be taken to use the right level of abstraction for those interfaces and structures. Inappropriate granularities might lead to interwoven and none composable parts or might hinder the roles in the contribution of building blocks to the ecosystem.

An important prerequisite for a robotics business ecosystem is the clear separation of individual roles and their contributions to the development of robotic systems. This separation of roles within a robotics business ecosystem decouples the development of the system parts in a way that the roles can work and push their products (system parts) into the ecosystem independent of the work of other participants. With the development with a robotics ecosystem in mind, there is a shift in the development method, from integration-centric development to composition-oriented development of robotic systems, compare Figure 4.5.

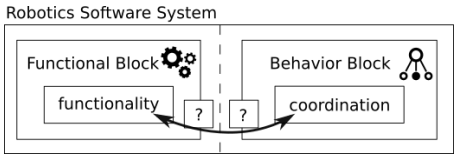

Figure 4.4.: The key to composition is the introduction of structures and patterns. The interface between functional and coordinating parts is crucial for the development of robotics applications.

Developing complex software systems, comprising many decoupled software parts, introduces the need to bring the individually developed parts together. With complex software systems, where parts are taken from a robotics ecosystem and therefore no detailed knowledge about the internals is given, the development of robotic systems has to be done in a composition-oriented manner.

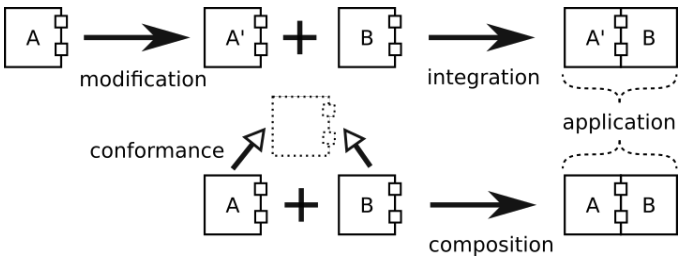

Figure 4.5.: Integration-centric development vs. composition-oriented development. Integration-centric development adapts the block and the interface according to the counterpart to integrate with, while in case of composition-oriented development the blocks are based on guiding structures that enable the composition, thereby leaving the block unchanged.

The structures and interfaces within or between the separated parts form the foundation for the collaboration of the roles within such a robotics business ecosystem.

The most difficult and important challenge with composition-oriented system integration is to keep or fulfill system-level properties spanning across the composed parts. With traditional integration-centric system development, the integration phase is used to realize system-level properties by changing the system parts to the needs of the concretely developed application. While this might be an advantage for a small system where the system parts are not heavily reused, the modification of the individual parts for each application would be fatal for the idea of a software ecosystem. The internals of the software parts from other ecosystem participants are not accessible or should not be of any interest, which is a wanted property.

While the development of robotic behaviors represents only one aspect of robotic system development, it is connected to the overall robotics system and its development. Those connections and implications, as well as the overall impacts on the development of the

robotics system, need to be considered in order to enable composable building blocks that could be coordinated in robotics business ecosystem.
In the following sections, the presentation of the envisioned robotics ecosystem focuses on composable robotics behavior development as this is the core topic of this thesis, followed by an explanation of the abstraction levels, the control layers a robotics system can be separated into, and their relation to the involved roles.

4.1. Baseline for Structures

The composition structures proposed by this thesis are methodically founded by *block-port-connector* models, following the definition given by RobMoSys [Pro19c]. *Block-port-connector* models are a specialization of a more abstract hierarchical (property) hypergraph [ES95; LP91] and entity-relationship models [Che76]. Entity-relationship models conform to hierarchical hypergraph models, and the block-port-connector models conform to entity-relationship models. This chain of conformance provides the scientific grounding for the proposed structures.

The *block-port-connector* models formalize the structures introduced by the architectural patterns. The models provide the abstract realization and technology-independent definition of the proposed structures. This foundation is used to realize the meta-models, on which the tool- and framework-implementations are based on, see Figure 4.6. The idea behind a *block-port-connector* model, with its reduced and yet powerful basic modeling set, is the ability to model any entities and relations between the entities, irrespective of the domain or abstraction level.

The *block-port-connector* model provides the entities block, port, dock, connector, and collection. The concept features the relations is-a, instance-of, conforms-to, constraints, as well as contains and has-a. Figure 4.7 shows the basic elements and relations. The concept of *block-port-connector* models is further described in the RobMoSys wiki, see [Pro19c].

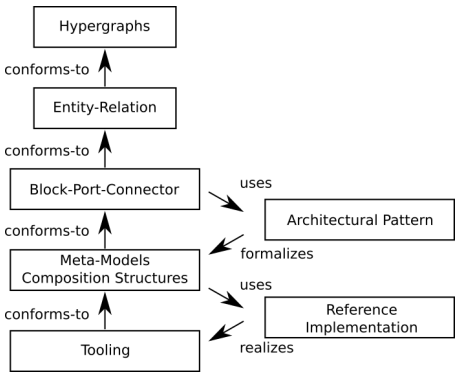

Figure 4.6.: Block-port-connector models as foundation for the composition structures the realizations are based on.

6

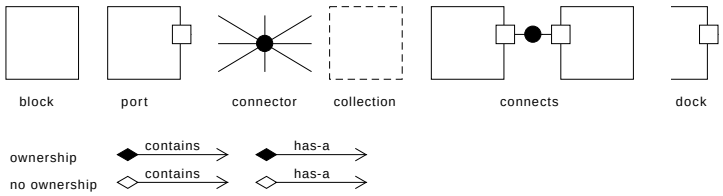

Figure 4.7.: The basic elements of the block-port-connector models, further details see RoboMoSys [Pro19c].

Block-port-connector models are used in this thesis to express all kinds of entities on several abstraction levels, as well as the relations between them, both horizontal and vertical. The thesis uses a *block-port-connector* approach to model structures, the behavior of structures, and the semantics of the structures by focusing on composable coordination. The proposed structures follow a defined motivation, mainly driven by the need to organize the handover of artifacts between the separated roles and the composition of the artifacts. Dependent on the level of abstraction, the structures within ports can, in turn, be defined using blocks, where a port becomes a block with inner and outer docks. Therefore, the same elements can either be a port or a block, depending on the view and abstraction level taken on the element.

The definition of blocks, ports, and connectors in isolation would result in arbitrary models and structures. Only with defined granularity, semantics, and behavior, the models provide the basis for a formalization of the required structures, their behavior, and semantics that enable the desired properties. Entities and relations without a proper definition lead to arbitrary concepts that seem compatible, but could be realized in many conceptually incompatible ways. Thus, the concepts would not contribute to the idea of competing realizations based on compatible concepts, but would result in fragmentation of the ecosystem. Even though it is not possible to formally define all relevant semantics in a formal way due to the high complexity, one can instead provide tooling and reference implementations to specify a procedural semantics.

The elements of main interest are the ports of the blocks as they define the interfaces between blocks. This applies to all kinds of blocks—e.g., roles connected horizontally or technical artifacts on different abstraction levels connected vertically. The ports decouple the entities represented by a block with an inner dock of the port and an outer one. Connecting the entities vertically, the upper entity should only be connected to the next lower one. Accessing a layer further below should not be possible or necessary, as this would introduce undesired dependencies. Connecting the entities horizontally, the entities should be decoupled using their ports. Irrespective of the internal realization, other entities are able to rely on the port as a stable interface, see Figure 4.8.

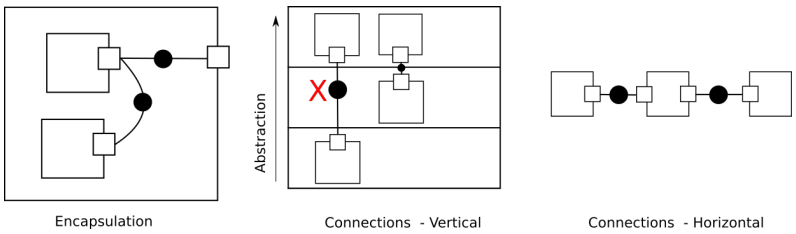

Figure 4.8.: Block-port-connector models used to model entities and their relations to define structures for composition.

4.2. Robotics Software Ecosystem - The Vision

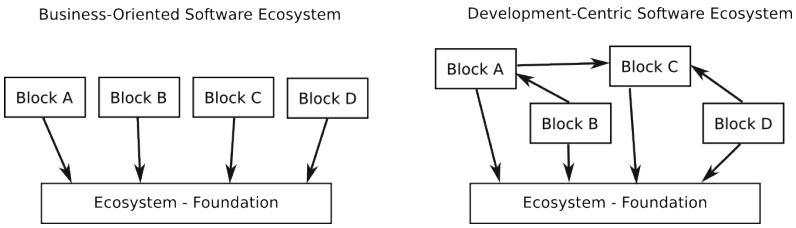

Figure 4.9.: Business-Oriented Software Ecosystem vs Development-Centric Software Ecosystem.

This section introduces the envisioned ecosystem with the main focus on robotics behavior development and how this will influence the required structures. Within a software ecosystem, the participants of the ecosystem typically share a common vision or common goals. For the envisioned robotics ecosystem here, the common vision is to build better robotic systems composed (can be put together by composition "composed") out of individually developed building blocks. The participants envisioned the development of more stable systems with a lot less effort by combining the individual expertise of the ecosystem participants. The individual goals of the participants within such an ecosystem depend on their role(s) within the ecosystem. The main focus of that ecosystem is on robotics behavior development and its connection to other system parts. Software ecosystems, in general, can be seen as a particular kind of business ecosystem featuring a shared technology platform as a pivotal point of interaction among the participants, as is described by Jan Bosch [BB10]. In a software ecosystem, the "products" interact (communicate) in a much more intense way as they typically do in classic business ecosystems. Software ecosystems exist in many different flavors, such as (illustrated in Figure 4.9): business-oriented (e.g. Android, iOS) and development-centric ecosystems (e.g. Debian), see Figure (4.10). The former focuses on the distribution of applications in shared markets [JFB09] where individual products can exist or work without others, while the latter focuses on the development of applications or systems with a deeper and maybe symbiotic relationship among the building blocks.

The vision of a robotics ecosystem, as illustrated here, clearly follows the idea of a development-centric type. The main vision is to develop better robotic systems, with the interaction among the parts within the ecosystem being central to this vision. However, this does not at all exclude the important business aspects of the envisioned ecosystem. The possibility to compete, share risks, and use the ecosystem as a platform for distribution is a major part and value of the ecosystem idea.

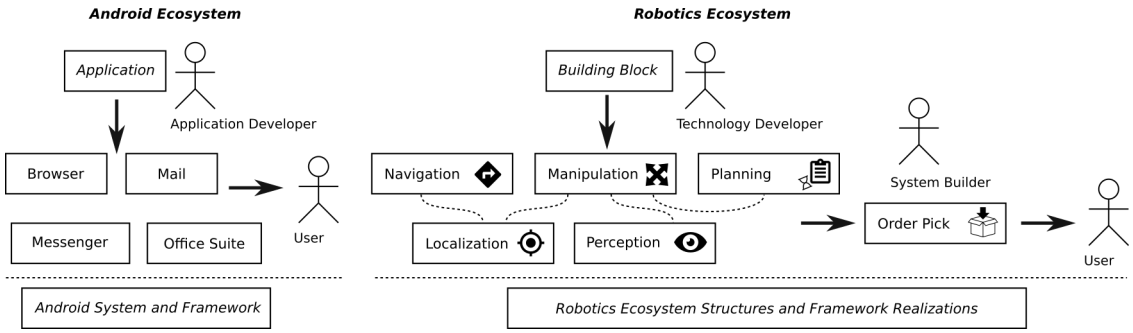

Figure 4.10.: The Android ecosystem is a typical example of a business-oriented, transaction ecosystem, where the developed applications/apps work in isolation to each other. In a robotics software ecosystem, the developed building blocks are composed to applications, with the building blocks interacting with each other to realize an application.

Robotics as a discipline within the field of software-intensive systems needs to deal with software complexity, which, to a big extent, originates from the inter-dependencies of functionalities and their access to consistently configured shared resources such as sensors and actuators. Many such blocks might not work without others providing necessary services to them. A robotics software ecosystem with the goal to enable the composition-like reuse of software building blocks needs to deal with this level of interaction among the building blocks to an even further extent as might be necessary for other software ecosystems (e.g., Android or iOS), where the blocks within the ecosystem typically do not interact at all. As a robotics software ecosystem will need to contain different types of software blocks, which are necessary to build robotic systems, the interaction among these blocks needs to be clearly defined. Those interfaces where the building blocks are handed over to other roles are most critical in enabling a working software ecosystem. The different roles developing these blocks need to be able to rely on these definitions and structures, as the users of the blocks should not be forced to know or event understand or change the internals of the blocks.

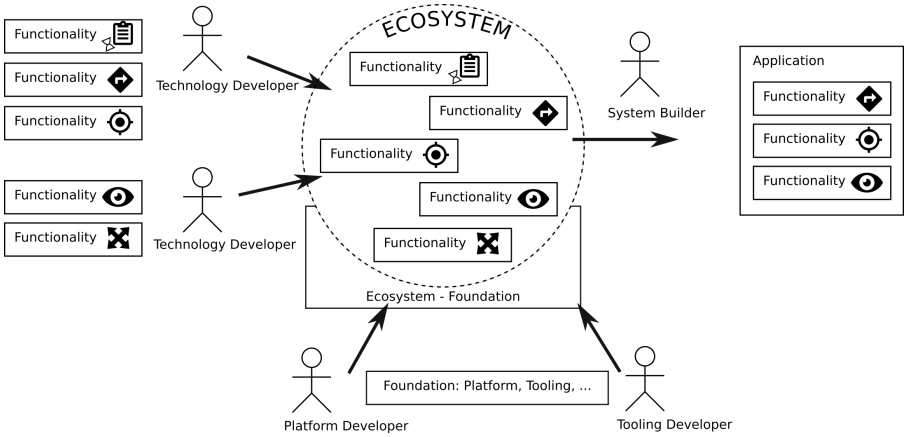

Figure 4.11.: The robotics ecosystem idea, where the development of building blocks of various kinds, is done by different ecosystem participants that are separated in time and space. For example, technology developers are able to focus on their role and push building blocks to the ecosystem, taking up the requirements and requested functionalities stated by system builders.

4.2.1. Behavior Coordination in a Robotics Software Ecosystem

This general robotics ecosystem vision abstracts in principle over all different types of building blocks or system parts, compare Figure 4.11. The main focus of this thesis is, however, on robotics behavior coordination development at the upper abstraction levels of robotic systems. Robotics behavior coordination is developed at a "higher" abstraction (symbolic) level with blocks like "grasp object" or "approach location" and is linked to functionality realized by lower abstraction levels. A more detailed description of the relevant abstraction levels and the related control architecture is given in the next two sections. For a simplified illustration of the relevant software parts to the ecosystem, the differentiation between the previously introduced functional parts providing functionality (e.g., navigation [moving a robot], object recognition, etc.) and behavior parts using the functionality and providing the coordination of multiple and concurrent running functionalities is helpful. The hardware (e.g., cameras, drives, etc.) is represented and made accessible to the behavior parts by the functional parts and is therefore included in this simplified model.

With those two kinds of software building blocks, the overall goal to build better robotic systems by composing building blocks by focusing on robotics behavior coordination can be addressed. The vision of the robotics ecosystem that considers functional and behavior parts can be briefly described as the following:

The development of robotic applications is possible by composing "as is" building blocks, shared via the robotics business ecosystem marketplace. To make use of the building blocks, the participants are able to rely on data sheets describing the outer ports, and no further knowledge or change of the internals is required to make use of them. The participants of the ecosystem are able to contribute without the need to directly interact with other participants. Both kinds of self-contained ecosystem building blocks – behavior and functional – can be developed, distributed, and composed to new systems and applications. The individual roles are able to focus on their expertise and to rely on the expertise of the other roles by using building blocks from them. The role of the application developer realizing the application and contributing application knowledge, which is of special interest for the robotics coordination perspective, is about developing the tasking of the robotics system. Changing parts of an existing robotics system is now achievable with much less effort as it is now about swapping parts of the system. This holds true especially for the change of the robot system tasking, where new tasks can be introduced reusing already available capabilities. In addition, the building blocks could also be realized as closed source hiding the intellectual property of the developers.

4.2.2. Composing Ecosystem Software Parts

The challenging part for the ecosystem, also illustrated by the user stories in the introduction, is to compose the functional parts as well as the robotics behavior parts to a working system. The right level of abstraction for the structures and interfaces has to be found in such a way that the sweet spot between Freedom of Choice and Freedom from Choice [Lee10] is found in order to enable the realization of a working robotics ecosystem. Too much freedom, or freedom at the wrong places, and the individually developed parts will not fit together. Too much restriction and the individual roles might become limited in solving their problems and thereby develop robotic systems.

The ability to compose individually developed parts is the most important requirement for a robotics software ecosystem. Without the ability to compose the parts of an ecosystem into a working system, the ecosystem will degenerate into a collection of individual software parts and lose its attractiveness.

Composing, in general, is "to form by putting parts together" [CamDict]. The term composability in a philosophical sense, following the definition of Gottfried Wilhelm Leibniz, gives more depth to the meaning with respect to the relations of the parts. Leibniz defines *composability* as parts being "zusammen möglich", German for compatible with each other [KMH13].

Petty and Weisel [PW03] separate the term composability into two parts— engineer-

ing composability or syntactic composability and model composability or semantic composability. Syntactic composability describes the ability that parts will fit together from an engineering perspective—e.g., the interface is compatible, regardless of what data is being transmitted and if both sides are able to understand the data. Semantic composability describes a higher degree of the relation among the parts as they will perform together reasonably. The RobMoSys definition of *composability* is: "The ability to combine and recombine building blocks as-is into different systems for different purposes in a meaningful way." [Pro19d]. *Compositionality* extends the ability of composability, *compositionality* is the ability to compose parts in a methodological way in order to meet predictable functional and extra-functional requirements c.f. [Pro19d].

Composition as a term in computer science has a rather long tradition and ranges from all kinds of abstraction levels. How to encapsulate, how to define the interfaces, and how to enable reuse are some of the fundamental design questions in computer science. Composition can be found on code, library, framework, model, and other levels; it is, for example, an important criterion and property to rate programming or developing paradigms and styles – e.g., comparing object-oriented programming and functional programming.
The composability of functions is given once their interfaces match regardless of classical or higher-order functional interfaces (no hidden states, side effects), whereas the composability of objects depends on how the objects are encapsulated.
This example demonstrates that composability is achieved by defining the interfaces with the right level of abstraction, thereby providing encapsulation. The right level of abstraction and the granularity of software parts to compose are important conditions for comparing different approaches to composition. Composing algorithms, for example, offers other constraints than composing more abstract parts such as applications.

To make parts composable, several aspects need to be considered. One of these aspects is the already mentioned aspect of defining the parts at a reasonable level of granularity. This level depends on the application to build and on how the ecosystem is structured. Too fine-grained parts lead to high coupling among the parts and limit their composition in different constellations.

The aspect of the interface among the parts in an ecosystem is probably the most important one. To make an ecosystem work, well-defined and enforced interfaces among the parts of the ecosystem is one of the most important property to achieve. Leaving alone the participants of an ecosystem with the choice of how to define interfaces would result in numerous different and incompatible ones. This would break the ecosystem as the parts would not be composable anymore. The interfaces among the parts should be limited as few as possible to enable the composition of the parts. The semantic meaning of the interface must be well defined. There should only be one possible interface option to achieve a particular purpose. The interface of the parts should, however, not

artificially limit the participants in their work. If the interfaces among the parts would limit the participants, the attractiveness of the ecosystem would obviously be reduced.

System-level properties or properties spanning across the boundaries of the individual parts need to be considered for a composable system. The challenge here is that those properties need to be kept stable regardless of the other parts are in the composition.

Most of the aspects are difficult if not impossible to fix, in general, for all kinds of domains. For specific domains, such as robotics or cyber-physical systems, the hurdle of fixing important aspects for the overall ecosystem is much lower, but still challenging.

4.3. Influences upon Composition Structures

Before going into the structures and interfaces enabling composition and a robotics ecosystem, including the development of robotics behaviors, this section introduces the most important influencing factors on the structures, namely the roles and the ecosystem entities. The overall goal the thesis is contributing to, is the development of robotic systems, including robotics behavior coordination, in the context of a robotics ecosystem. Thus, the involved roles need to be able to work and contribute their expertise in separation to others. The building blocks developed by the separated roles and distributed via the ecosystem need to be composable in order to make use of the expertise for the development of systems and applications. As a consequence, those structures that enable the later composition of the parts need to be explicated and fixed. The questions left open are:

> *Which roles contribute which parts to the ecosystem and how do they work together?*
>
> *Which entities need which structures to enable the composition of building blocks for system and application development?*

The Roles within the Ecosystem First of all, the most obvious influencing part of the ecosystem itself are the roles participating. Their roles and the motivation, or their expertise they bring into the ecosystem, influence those structures directly. A role solely motivated by the needs of the application should not be forced to deal with technological details such as the algorithmic details of another role. Therefore, the handover and interfaces between the roles need to be managed to enable their decoupled collaboration without the need to understand the internals of the blocks developed by other roles.

The Entities within the Ecosystem and their Granularity The entities within the ecosystem and their granularity clearly influence the interfaces and structures required. If there were only one type of entity within the ecosystem, the sole interface needed would be between just that type of entity. With different types of entities, the interfaces

among the interacting entities need to be identified and structured. How the granularity of the entities affects the interface within the ecosystem may not be so obvious at first glance. However, when two different types of building blocks need to interact, the interface between them changes drastically and depends on their granularity. For example, the interface between two blocks would change drastically if implemented at the function library level (programming language e.g., RPC [Whi76]) as opposed to a SOA-based interface [Erl07; SW04]. With the more fine-grained function level interface, the interaction and thus the coupling between the two parts gets tighter.

4.4. Roles in a Robotics Software Ecosystem

The participants and their roles in an ecosystem are pivotal points of the ecosystem idea. Their involvement and support need to be well-conceived and heavily influence the design of the structures and interfaces among the software parts. Different roles pursue different goals within the robotics software ecosystem envisioned here. Five fundamental roles can be distinguished. More roles could be defined, but those five roles are rather clear and fundamental. Without those roles, the ecosystem idea in the context of robotics behavior coordination would be very limited.

The role of the **technology developer** or component developer (Figure 4.12) contributes building blocks to be used by others in the ecosystem. These developed building blocks provide capabilities used to compose applications. The building blocks can, in turn, use other building blocks from the ecosystem to provide its capabilities. The technology-developer role requires the definition of domain-specific structures to develop composable building blocks. The goal or the common motivation for technology developers is to distribute/sell their technologies using the ecosystem.

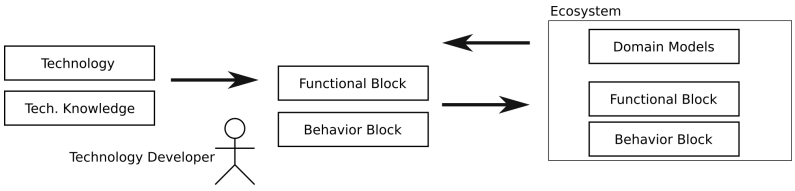

Figure 4.12.: Technology developer role, contributing functional and behavior blocks.

The second role is the **behavior developer** (Figure 4.13), which contributes robotics behavior coordination building blocks. These developed behavior building blocks express application knowledge and how something is achieved by making use of capabilities. The role requires domain-specific interface structures, as does the technology developer.

The third role is the **system builder** or application developer (Figure 4.14). The role uses the building blocks offered by the technology developer to compose systems and

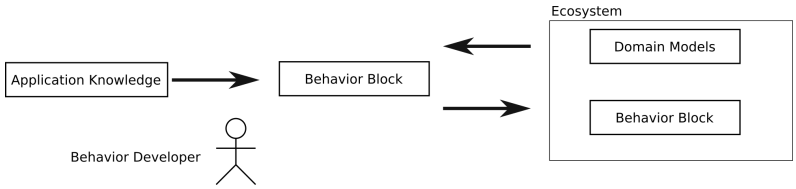

Figure 4.13.: Behavior developer role, contributing the behavior blocks required to build applications.

applications. Those applications can again be provided to others in the ecosystem. The common goal of the system builders is to realize systems by composing blocks offered by others. The role contributes the knowledge of how to build the applications, including the required domain/application knowledge. The system builders require building blocks to realize applications, without the need to develop the technology by themselves.

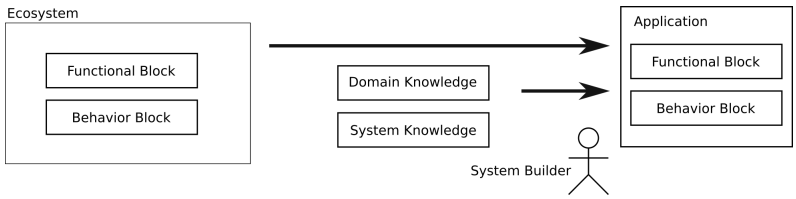

Figure 4.14.: System builder role, contributing to systems and applications, requiring building blocks.

The fourth role is the **domain expert** (Figure 4.15). The role contributes to the domain-specific structures that enable the composition of ecosystem building blocks. The structures contributed by the domain experts form the domain-specific part of the interface definition that is used to compose building blocks.

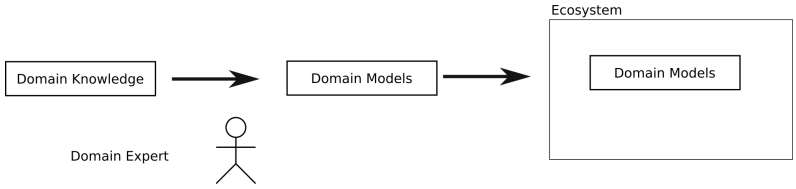

Figure 4.15.: Domain expert role, contributing to the domain-specific structures (interfaces) required to build composable building blocks.

The fifth role is actually a group of roles of the **ecosystem drivers** (Figure 4.16). The group develops the structures that build the foundation of the ecosystem. The goal of the role is to build the best ecosystem. Based on those common structures, the role of the platform or tool developer can realize tools to make the ecosystem accessible and to provide it to the participants of the ecosystem to make participation easy and attractive.

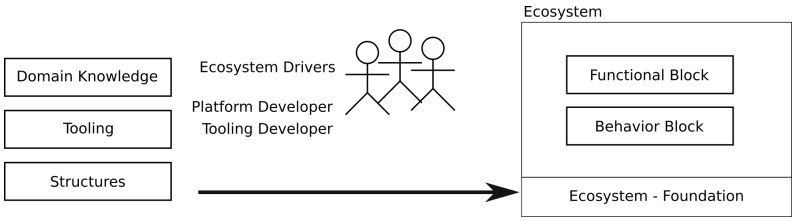

Figure 4.16.: The ecosystem drivers setup the core elements. The contributions of the role form the ecosystem foundation all other roles work on.

4.5. Composition of What - Entities and their Granularities

Composition in the context of this thesis includes different system parts on different abstractions levels. It spans across the control architecture of a robotics system. Before describing how and which parts can be composed and which are offered in a robotics ecosystem, the following will present which parts need to be considered.

This separation can be done along several views along a vertical axis. The following will first introduce the view along with abstraction levels a robotic system could be divided into. A second and reasonable view for robotics behavior development follows the layers of the control architecture. These two views can be seen side by side (Figure 4.17), representing the two different perspectives of the structure of a robotics system.

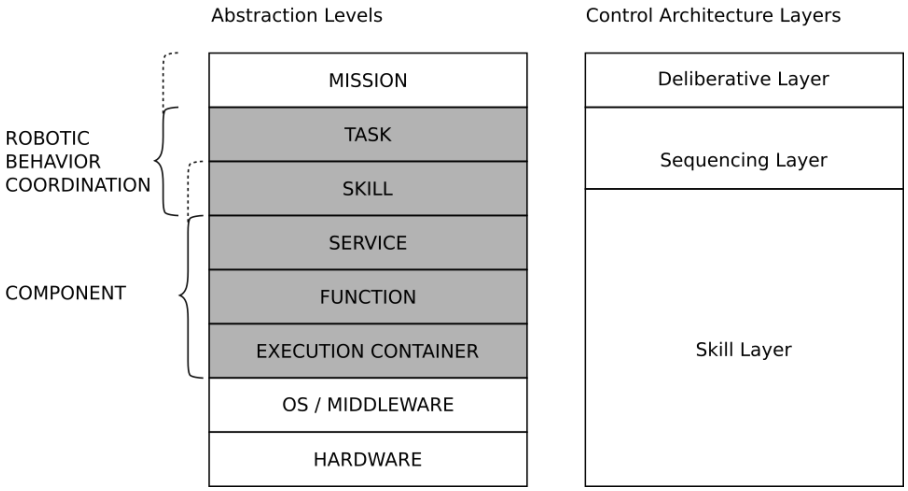

Figure 4.17.: Abstraction levels and the mapping of the control architecture layers in a robotic systems [Lut+19a] and [Lut+18a].

Besides describing the ecosystem-relevant entities, the section also presents the relation of the entities to the overall robotics system development. It shows those entities where interfaces between different control layers are given and how they are related to the roles involved in the robotics ecosystem.

4.5.1. Robotic Systems Abstraction Levels and Control Architecture Layers

This first section provides a view of the robotic system and the abstraction levels the system can be separated into. As robotic behaviors are most concerned with coordination and configuration (control), a side-by-side comparison of two stacks—abstraction level and control layers—is given, see also Figure 4.17.

Robotic System Abstraction Levels

A robotic system can be partitioned into different levels of abstraction, addressing different concerns. The separation of the levels enables the development and the use of level-specific and efficient solutions. The main benefit of doing so is to deal with the overall complexity of a robotic system. Each layer is a different reasonable level of abstraction, reducing the complexity faced by developers working on parts of a layer. The levels are stacked onto each other as they rely on the functionality of the levels below and provide an interface for the levels above it. Each level is, therefore, hiding certain aspects of its own and the underlying levels and their "functionality". Following the block port connector model, the layers need to provide interfaces to the level below and above. The number and the separation of the abstraction levels may be the subject of a controversial discussion. The functionality represented by the layers, shown in Figure 4.17 (left), is, however, present in most complex robotic systems. In some systems, some functionality, represented by the layers might not be clearly separated, is present there as well. ROS [Qui+09], for example, does not clearly separate the functionality of the middleware from the execution container, while others—e.g., SmartSoft [SW99; Sch98]—do separate this level. Explicating those levels or layers, as well as separating the individual concerns, has in many cases substantial effects and is a critical design decision. The number of layers should be kept as minimal as possible, with each layer providing a dedicated functionality decoupled from the layer below. If the decoupling provided by a layer is not required, the layer will be merged with another one.

There are some systems that do not feature all the stated functionalities, with typically being at the lower and upper ends. Some, for example, do not use task planning on the upper level, while others do not use an operating system on the lower end.
The individual level of abstraction does not limit the solutions developed for a level to be limited to a single robotic system. For some applications, the solutions might be spanning over multiple systems. For example, in the case of a robotics fleet delivering a service as a whole, the mission level could be realized spanning over a system of robots.

Robotic System Coordination Layers

Parallel to the levels of abstraction are the layers of the control architecture of robotic systems, Figure 4.17 (right). The control architecture applied within this thesis follows

the idea of the three-tier (3T) robot architecture [Bon+97; Fir89] from Bonasso, Firby, Gat, and others. It separates a robotic system from a control point of view into three layers, each with a focus on different concerns, which range from fast processing (e.g., sensors) via reactive task-sequencing to task-level planning.

Comparing the two stacks, the abstraction levels and the control layers show a similar granularity in the upper parts and a rather large difference at the lower end. This might look incompatible, even though they are not conflicting. As the control architecture's main concerns are coordination and configuration, the lower levels realizing the functionalities are simplified by a single layer only (*Skill Layer*).
The abstraction levels, on the one hand, distinguish between several levels at the *Skill Layer* of the control architecture, since other concerns and properties are considered there as well (computation, communication, separation of concerns and roles etc.). Care has to be taken to not distinguish the skill coordination layer from the skill abstraction level. The skill coordination layer subsumes all things necessary to provide abilities for robot tasking. The skill abstraction level, on the other hand, is the layer proving the translation from domain-specific generic terms towards the task abstraction level and the technology- or solution- specific interface toward the realization of the functionality via the services level.

Software Components in Abstraction Levels and Control Architecture Layers

The software component spans across multiple abstraction levels, see Figure 4.17. This is because the component is used to technically realize more than one of the abstraction levels. The software component is the wrapper around the realized functionality, thereby providing the execution container with all the associated concerns (computation, communication, coordination, configuration). The components are the main system building blocks. Each layer in the control architecture is realized using components. There is no activity within the system that is performed outside of a component, as it is the block that encapsulates all functionalities. This holds true independent of the component's functionality and the layer to which the component is "assigned." All components feature the same fundamental parts and properties (services, lifecycle, etc.). This includes the sequencing component, same as the components on the deliberation layer, such as symbolic planners or a skill layered motion controller component. All functionalities are encapsulated within such coarse granular blocks (software component), mainly to enforce decoupling between the functionalities, thereby enabling the reuse of the parts in different systems and settings, as well as helping to deal with the overall complexity of robotic systems and applications.

4.5.2. Layers, Entities and Robotics Behavior

This sub-section describes the coordination layers and their related abstraction level(s), flowing the coordination layers top down. It also introduces the involved entities.

Deliberative Layer - Mission Abstraction Level

Description The *Deliberative Layer*, as the topmost one, reasons about the high-level goals of the system, using a symbolic task planner, constraint solver, analysis tools, etc. This layer can easily be mapped to the MISSION level on the abstraction level stack. On this most upper level, typically strategical decisions are made. It is of less or no interest how the decisions taken are realized as long as the lower layers are capable of following the decisions. For example, a task planner does not need to know how to control a manipulator while delivering a plan that defines the order to stack cups into each other to efficiently clean up a table. As it is the nature of this level to act on a high level of abstraction using actions with high expressiveness, the frequency this layer works on is rather slow. The layer must not be able to deal with high frequent control loops. A system design that involves this layer in high frequent low-level activities, such as collision avoidance, would be in conflict with the system design presented here.

Entities Within the *Deliberative Layer*, at the mission abstraction level, the relevant entities are the jobs and the objectives. Both are described at a high abstraction level, above the robotics task and skill behavior models.

Interface As the uppermost layer of the robotics system, this layer acts on a symbolic level, typically at the granularity of goals and objectives to achieve. It uses the tasks and information provided by the task level or the *Sequencing Layer* below.
As this layer works on aggregated symbolic information, it typically uses information aggregated within a knowledge base.

Relation to Robotics Behavior Development For robotics behavior development, this layer is used as a layer of solver (such as symbolic planner etc.) to be asked for how to solve a specific problem. For instance, the layer below asks this expert layer in which order to perform some action to deal with a huge possible solution space that the lower level might not be able to address reasonably. In relation to this thesis, the layer is considered the upper end of the robotic system.

Sequencing Layer - Task and Skill Abstraction Level

Description The 3T control architecture most importantly decouples (fast) reactive processing on the lower *Skill Layer* from the (low frequent) symbolic level processing on

the *Deliberative Layer*. The bridge between both worlds is the intermediate *Sequencing Layer* . The *Sequencing Layer* is responsible for situation-dependent reactive task execution and coordinating and configuring all other software components in the system. The sequencer component is, therefore, typically the master component orchestrating all other components on all layers.

The *Sequencing Layer* can be matched best to the abstraction level of TASKs, since on the *Sequencing Layer* tasks are the elements that express what a robot should perform and how underlying skill components are coordinated and configured to do so. The second abstraction level that could at least partially be matched to the *Sequencing Layer* is the abstraction level of skills. Skills belong to both layers—the sequencing, as they are executed and used on this layer at runtime, and the *Skill Layer* since they directly interact with the components.

Entities Within the *Sequencing Layer* , at the abstraction level of task and skills, the relevant entities are the task and skill behavior blocks. The task behavior blocks realize the behavior of the robot to generate a service or to fulfill the objective and goals from the mission level above.

The skill behavior blocks realize the interface for the configuration and coordination of the components. Skill blocks are the linking parts between the skill-layered components and the tasks; they lift the level of abstraction from the functionalities to the level of tasks by interacting with the components. In doing so, skills realize the translation from domain-specific realization-independent terms (e.g., move-robot) used by the tasks to solution/technology-dependent interfaces toward the components and functionalities.

Interface The *Sequencing Layer* being at the core of robotics behavior development offers a symbolic interface to the *Deliberative Layer* or the mission abstraction level above. It explicates the functionality realized on this layer and the ones below as tasks. The offered interface is independent of the realization.

At the lower end, it requires a connection to those blocks or components that realizes the functionalities. In this thesis, this interface is achieved by the skills being the bridge between both the sequencing and the *Skill Layer*.

Relation to Robotics Behavior Development The *Sequencing Layer* is the central layer for behavior development as there the robotics behavior or tasking of the robot is achieved. The coordination of all the software blocks or components is achieved to realize a specific task or robot behavior.

Skill Layer - Skill and below Levels

Description In the 3T control architecture, the *Skill Layer* as the lowest layer realizes the functionalities required to fulfill the tasks and goals of the layers above. The components on the *Skill Layer* are those that need to be coordinated and configured to make the robot perform a task or to fulfill a goal. The *Skill Layer* represents plenty of abstraction levels as all the levels below can be wrapped by a single layer from a coordination view.

Entities Within the *Skill Layer*, at the skill abstraction level and all the levels below, the relevant entities are the skill behavior blocks and the components.

The skill behavior blocks realize the configuration and coordination of the components. Skill blocks are the linking parts between the components and the tasks; they lift the level of abstraction from the functionalities to the level of tasks by interacting with the components. The skill blocks can be seen on both layers—the sequencing and the *Skill Layer* – as they are the interface between both layers.

From a robotics behavior perspective, the second relevant perspective on the *Skill Layer* is the component as the functionality-containing block.

Interface The skill level essentially provides the translation between the sequencing and the skill component layer. It bridges the sub-symbolic and continuous execution—e.g., navigation path planning, obstacle avoidance, etc.—and the symbolic level task sequencing. On the upper side, it provides the domain-specific and generic interface to the tasks, making the functionality of the components accessible (e.g., moving the robot to a location). On the lower side, it interfaces the components via services to coordinate and configure. The skills decouple the tasks from the realization of the functionality at a lower level.

Relation to Robotics Behavior Development The skill level provides the basic building blocks for robotics behavior development. On this level, the coordination and configuration of the lower-level skill components are achieved. The separation of this level from the task level and from the realizing components below allows for a decoupling of the task level robotics behavior blocks from the realizing components. This is one of the prerequisites to allow for composition-like system integration or application building on the robotics behavior level.

All the abstraction levels below are represented within the *Skill Layer* in a robotics 3T architecture. This includes the following abstraction levels:

Service Level Component *services* are the interfaces for communication among the components. A service is realized within a component, lifting the communication level

between the components from functional- to service-level granularity. They decouple the sphere of influence between the components and prevent a fine-grained interaction among the components. The component services enable the reuse of the components in different settings. Services are realized on the functional level and accessible to other functionalities through the execution container. From a robotics behavior coordination view, services offer coordinating access to a component, thus resulting in vertical communication following the control architecture, in contrast to the horizontal communication between skill-level components.

Looking at the system from an abstraction level of services downwards, services are the dominant and decisive entities which define the systems component architecture. Expanding the view upward to the skill and task levels and thereto toward robotic behaviors and coordination, software components are getting more relevant. As software components are the system parts that are coordinated, these components encapsulate the realization of the functionalities. The services of the components are decisive entities while composing components to systems (horizontal).

Functional Level The abstraction level of *functionality* is where the realization of those functions represented by skills and used by tasks on the *Sequencing Layer* is located. In addition to the functionalities offered by the skills, the realization of the services is located within the functional level. The realization technology on this level is not limited to, but is typically done by using existing software libraries (e.g. Boost, OpenCV, MRPT). Semantically coherent functionalities having lots of dependencies are typically realized within one component (high cohesion low coupling).

Execution Container To realize functionalities on the level above, the abstraction level of the execution container provides infrastructure, such as threads, or the means to communicate with other functionalities in other components via services. The execution container also offers the means to realize a component-wide realization of configuration and activation, as well as a common and user-defined component lifecycle. The execution container abstracts away the direct access to the middleware, thereby providing a more easy-to-use, stable, and decoupled interface (middleware used) on which the functional layer can be built upon. The same holds to some extent for the abstraction of the operating system such as the threads (execution context). In an ideal world, this would also abstract away the operating system completely, which is, however, not possible since some OS-specific parts cannot be hidden easily (e.g., hardware access).

Operating System and Middleware Level On the operating system and middleware abstraction level, the resources for computation and communication are provided and organized. Located above the hardware level, it abstracts some of the generic hardware parts—e.g., computational resources and distribution of them across multiple nodes (computers)—as well as communication spanning across processes and nodes using

network transmission. The OS and middleware level lifts the level of abstraction from hardware to the pure functionality of computation and communication. From a robotics behavior development view, the OS and middleware level should be and can be hidden.

Hardware Level For the hardware abstraction level, there are two different cases that need to be highlighted. First, there is the simple, with respect to coordination, case of "generic hardware", which is not specific to robotic systems and applications. This kind of hardware is abstracted away by the above level of OS and middleware. This does not mean that the properties of the abstracted parts are not accessible anymore (e.g., QoS-like bandwidth). Examples of "generic" hardware are computational resources, memory, or network communication.

The second case is focused on robotic hardware parts, especially those directly interacting with the real-world. Some of the examples are manipulators, base drive units, cameras, etc. From a robotics behavior view, the abstraction level of hardware is limited to those parts that need to be organized on the level of tasks and skills. Even if those parts are organized there, it is done based on an abstract and symbolical representation. Those parts which do not need a symbolical representation on the skill or task level are managed within the components on the functional level. The skill behavior blocks are responsible for providing access to those hardware-relevant parts that they are abstracting (e.g., resources they are managing).

4.5.3. Robotic Systems and Ecosystem - Behavior Relevant Entities - a Verdict

Figure 4.18 illustrates the relations between the different views, abstraction levels and entities, while the areas marked by grey highlight the robotics behavior development-relevant interfaces between the entities.

Before going into how to compose the entities, this section summarizes the discussed entities of a robotics system and their ecosystem relation.

From a robotics behavior development perspective, there are three important entities that need to be considered within the ecosystem. Starting with the idea to compose capabilities, they are separated into two types of blocks—those blocks which realize capabilities (functional) and the blocks that coordinate system parts (behavior). Mapping those two parts to the abstraction levels proposed before indicates that all the levels below the service granularity can be wrapped by the component, seen from a robotics behavior development view. The coordinating part, on the other hand, must be separated into two parts: the skill behavior blocks directly interfacing the components and the task behavior blocks being independent of any realizing component.

The component block encapsulates the functionality and hides the complexity of the realization, the execution container, the operating system, and the middleware from the robotics behavior layers (deliberation and sequencing layer). The task behavior block realizes the robotics behavior independent of any realizing robotics component. The

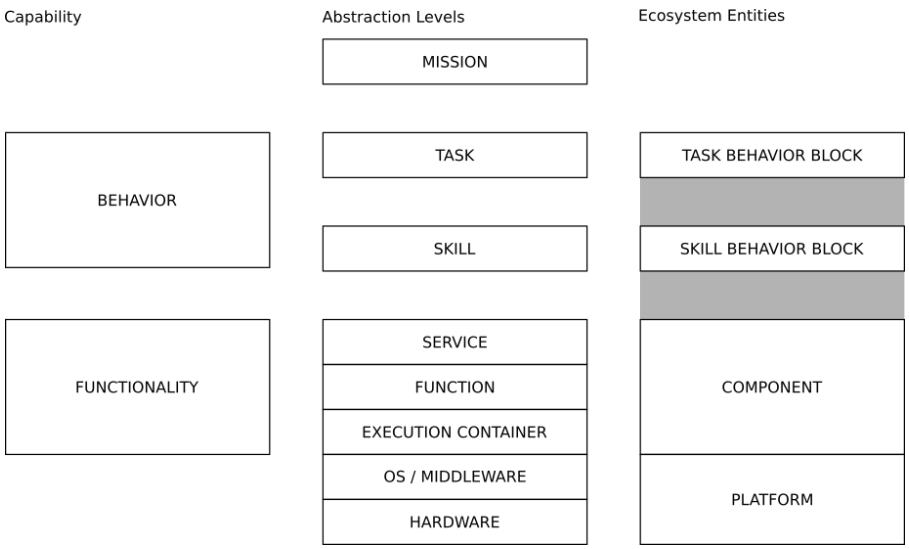

Figure 4.18.: Mapping between the idea of composable capabilities, robotics system abstraction levels, and the relevant ecosystem entities. The grey marked blocks are the connections organized within this thesis to realize a robotics ecosystem.

skill behavior block realizes the translation between the generic domain-specific terms used within the task behavior blocks and the realization-specific bindings to connect to the functionality realized by the components. The skills are the main building blocks used for robotics behavior development on the task level. They provide access to the functionality realized within the components. All lower levels—execution container, OS, middleware, or hardware – are not that relevant to the robotics behavior development as the component abstracts and hides them. The relevant properties or options need to be reflected (direct or refined) at the interface to the skill behavior blocks. Which is in line with the hierarchy of the abstraction levels.

From a robotics behavior development perspective (and its development workflow), composition is primarily necessary at two points: first, the composition of skills and tasks interfacing the roles of the technology developer and the behavior developer; second, the composition of tasks with tasks composing task behavior models created by different behavior developers. While those two compositions are the most important ones, further structures among different parts are relevant and also addressed in the following patterns.

4.6. Coordination Composition Patterns - The Approach

This section presents the proposed approach to address the above-shown topics as coordination composition patterns. The patterns are presented in a condensed and

focused schema, with the later chapters substantially detailing the patterns. The structure as how the patterns are described follows or is motivated by the series of books named *Pattern-Oriented Software Architecture*. The description given in the first volume *A System of Patterns* [Bus+96] is well established and captures the most important properties in a reasonable schema:

- Example, giving motivation for the problem the pattern solves.
- Context, describing the context where the pattern could be applied and where not.
- Problem, describing the issue to solve, including the influencing properties (forces) which need to be considered when solving the problem.
- Solution, showing how to solve stated problem by balancing the properties (forces). The description of the problem is typically twofold, describing the structures and the behavior of the solution.
- Consequences, stating the pros and cons of the application of the pattern.
- Implementation, where helpful remarks on how to implement the pattern will be given.

4.6.1. Component Coordination Interface - Composition of Behavior Coordination and Functionality

For the purpose of coordination, the *Component Coordination Interface* pattern explicates, structures, and semantically enriches the access to the functionalities within the components. The added semantics and structures are necessary to establish a robotics business ecosystem with software components capable of runtime coordination. The following describes the overall approach, introducing a structured coordination interface, with further details and the organization of the interface being described in Chapter 6.

Example

A robotics software component encapsulating the functionality to detect and recognize objects is used together with other software components in a robotic system. The system uses the recognition component in different contexts in cooperation with a changing set of other components. To make use of the components and to introduce application logic, the robotic system is developed together with robotics behavior coordination blocks. The coordination needs to be able to access the individual software components – e.g., the recognition component. The recognition components require data and need to be connected with the appropriate sensor components to receive the data. Depending on the task, it might be necessary to apply different recognition algorithms and the component needs to be configured to detect relevant objects. For the detection of pharmaceuticals with the required

reliability packages, a close-up image from an eye-in-hand sensor and a barcode recognition algorithm might be used to simply read the product's id – e.g., the pharmacy product number. For the detection and recognition of coffee cups, a scene-wide image or a point cloud-based algorithm might be sufficient in terms of reliability, requiring only little effort. Dependent on the scenario, the behavior coordination needs to activate different modes of the components. The recognition could be performed continuously or a single run only. The medicine example might be best dealt by capturing a single image once the arm is in the perfect place. For situation- dependent reaction and task planning, the coordination system finally depends on the results provided by the recognition component.

The same example, seen from a development and ecosystem perspective, introduces some additional aspects and adds an extended view to the example. The recognition component, provided via a software marketplace, is selected by two different robotic application developers. Both want to realize a robotics application by composing existing software components. They integrate the component and add the required application knowledge to the robotics behavior coordination abstraction level. Dependent on their application, each developer uses a different behavior coordination approach. In the case of a simpler and rather static application, such as the picking of pharmaceuticals packages, a state machine is used to realize the robotics behavior coordination. On the other side, for a flexible and reactive robot butler service, a hierarchic task tree approach might be better suited. Regardless of what approach is used, both approaches require an interface and access to the functionality encapsulated by the component. Both need to be able to connect to, to configure and activate recognition component and fetch data and results from it. Due to the structures and the semantics of the coordination interface, the developers are able to access the functionality of a component to make use of the as-is supplied component, as well as to connect the level of abstraction from functionalities and services to skill and task levels.

Context

The pattern is defined in the context of the coordination of closed software components that need to work together to achieve a goal.

Problem

To use different functionalities in a robotic system, capable of performing different tasks within an open environment, the functionalities need to be coordinated. The software component encapsulating the functionalities, therefore, needs to offer an interface to allow for the coordination of the component. If every component developer would push a component with a different coordination interface to the ecosystem marketplace, the composition of those components to a working robotic system would not be possible. The coordination approach that needs to make use of those interfaces would require the incorporation of a large number

of different interfaces to the components, not known in advance. Therefore, an approach that harmonizes and explicates the coordination interface offered by a software component is required to enable a working robotics business ecosystem.

Solution

To overcome the problem of numerous different interfaces for the coordination of components, this pattern introduces a harmonized and explicated interface. The interface considers the different use cases required to coordinate a robotic software component. Figure 4.19 illustrates the step change the pattern is enabling.

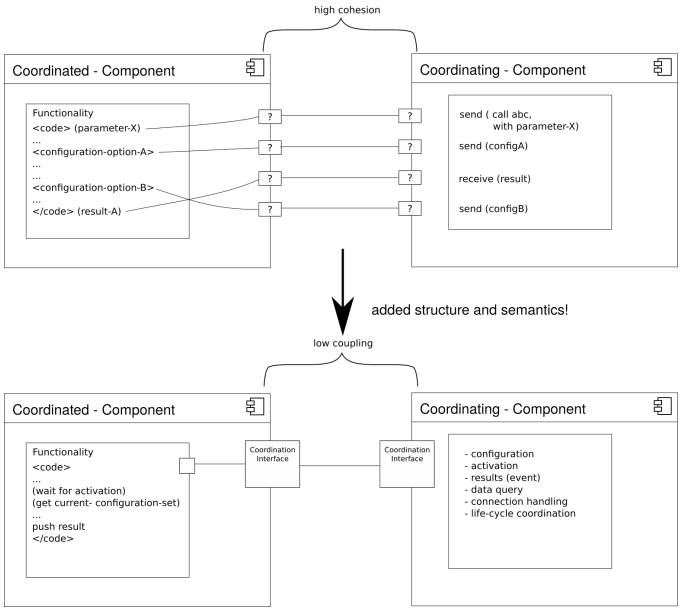

Figure 4.19.: Component Coordination Interface, explicating how to access the functionalities with components for coordination. The pattern adds structure and semantics to the coordination access, required to make use of dedicated robotics behavior coordination approaches, thereby enabling coordinatable software components in a robotics business ecosystem.

The interface consists of the following parts:

Configuration Runtime configuration of component.

Activation Activation of functionalities encapsulated with the components.

Results (Event) Receiving the results and messages of the activation of the functionalities within the components.

Connection Coordination of the inter-component connections, thereby configuring the data flow between the components.

Information Query Requesting and receiving information for coordination from components, as well as the use of expert components (e.g., symbolic planners) for coordination.

Component Lifecycle Providing access to the components' lifecycle—e.g., shutdown or error states of the components.

The coordination interface structures the coordination-wise access to components and their encapsulated functionality. The interface adds semantics to the coordination. Each part of the interface features a distinct and sharp functionality. The patter further defines the communication semantics of the interface parts and provides a user interface for the component developer (code API). The step from arbitrary access to the coordinated component, towards a semantically rich interface, enables the composition of behavior coordination and component encapsulated functionality. The usage of the interface for component coordination is further described by the *Orchestration Cycle* pattern, capturing the best practices how to make use of the coordination interface, see Chapter 6.

Consequences

As a main consequence, the pattern enables a working robotics software ecosystem that considers behavior coordination for complex robotics software systems. It further frees the roles of the technology developer from the burden to design its own interface and component-internal structures for component coordination. This allows for the development of software components more easily reducing the knowledge the developer has to gain.

4.6.2. Coordination Interface Types - Composition of Components

The *Coordination Interface Type* pattern introduces an external (with respect to the component) definition of the coordination interface. This implies the definition of coordination interface types (*Coordination Service*) that components will instantiate or realize. The parts the coordination interface consists of are defined by the previous pattern and are the same for all components. The content ("data types") communicated as well and their number depends on the functionalities that a component encapsulates.

The external definition of the coordination interface adds the possibility to compare interfaces of the components and allows for composition-like replacement and integration of software components. With defined coordination interface types, the skills reliant on the coordination interfaces can be developed decoupled from the actual software component realizing a skill. The separation can be used in two different ways. First and most obvious to enable a component-independent development of the skills, generating reusable skills compatible with different components. Why that is desirable in the context of an overall development workflow is discussed in Chapter 7. The second case is not so obvious, although it is rather important in practice. Often, useful skills are only created through the interaction of multiple components. A navigation approach,

for example, consists of a planner, mapper, and collision avoidance component which together provide skills to move a robot to a certain location. This means developing a single set of skills coordinating all three components and not being reliant on more fine granular skills that the individual component provide to realize the final interface skills usable on the task level (e.g., move-robot). With the externally defined interfaces, this can be done easily, as, irrespective of where the skills are defined, they are only dependent on the coordination interface service definition.

The following describes the overall approach of introducing coordination interface definitions, with the detailed description of the interface types being described in Chapter 6. The organization of the overall development workflow, including the definition of the interface types, is described in Chapter 7.

Example

An object-recognition component is available as a building block. A developer who is an expert in developing object-recognition components wants to develop a new recognition component and offer the component as a drop-in replacement for the existing recognition component, reusing existing skills (behavior coordination models). To develop an exchangeable component, the developer needs to realize the same interfaces, including the coordination interface the skills rely on. The object-recognition expert is able to make use of externally defined coordination interface definition to develop and offer a drop-in replacement recognition component. System or application developers are able to flexibly compose a system using one of those components realizing the same coordination interfaces.

Context

The pattern is defined in the context of the coordination of closed software components. While not explicitly required, the pattern is the most valuable when it is used with an explicated coordination interface, see the previous pattern.

Problem

While the *coordination interface* pattern defines the structures and semantics of the interface a component features toward the coordination, it does not define the domain- or component-specific types the interface uses. Further, the interfaces consist of different parts that could be present multiple times and used with different types such as information queries. To enable the composition of components as drop-in replacements by considering the coordination interface of a component, the specific coordination interface of a component needs to be explicated externally. Otherwise, the two components can only be identical (drop-in) by convention and not by definition.

Solution

The approach to tackle the proposed problem is the external definition of the coordination interface, namely *Coordination Service*. Thus, a specific coordination

service is defined as the domain-specific type. These types are defined by the role of the domain experts and capture the domain knowledge of how a components' coordination interface should be realized for a specific purpose, such as a navigation planner. The role of the component supplier developing the components is able to realize a component following and instantiating the defined coordination service. Figure 4.20 shows the basic principle and involved roles of the approach.

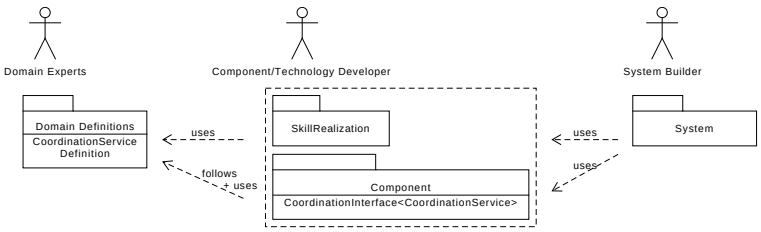

Figure 4.20.: Coordination Interface, overview illustrating the connections among the elements and the associated roles.

Consequences

The external definition of the coordination interface or the definition of types of coordination interfaces helps to realize exchangeable and composable software components. It provides the composition of different components using the same interface for coordination to systems. This is a major prerequisite for the development of a working robotics business ecosystem. Without this exchangeability, there will be no competition among technology providers.

On the downside, the pattern adds another step to the development workflow of the functionalities. This is especially considerable while developing a new functionality from scratch, with no existing interface definition available. To overcome this additional effort, appropriate tool support can be realized to guide the developers and simplify the development.

Apart from some practical considerations (coordination of multiple components), which could probably be solved on the tooling and implementation level, the use of the pattern and value mainly depends on the perspective of the components' interface for coordination. If this interface is primarily seen on the skill abstraction level and the reuse of existing skill realizations is negligible, the application of this pattern is not crucial. Otherwise, it introduces a separation, thereby contributing to the described possibilities and advantages.

4.6.3. Separation of Skill and Task Behavior Coordination Models - Composition of Behavior Coordination Models/ Composition of Functionalities

The pattern structures proposed here point to the development of robotics behavior coordination models. It separates behavior models directly depending on functionalities encapsulated in software components from behavior models independent of any software component. The pattern introduces and structures the skill abstraction level, separated from the task models. The separation enables the development of robotics behavior models on a task level without the need to bind them to realizing software components. The pattern adds to the composition of the behavior models and to different realizations of functionalities. The following describes the overall approach, separating skill and task abstraction level behavior models. The fully detailed approach is presented in Chapter 5.3. The organization of the development workflow, including the skills and tasks, is described in Chapter 7.

Example

This example focuses on the development of a robotic system performing a complex task by using the capabilities provided by the different involved and composed components. The task of fetching an object from a container located in another room utilizes the functionalities of navigation, perception, and manipulation. The logic of how to perform the task is specific to the domain and can be described in an abstract manner, without knowing which concrete components are providing the functionality. The task could, for example, be defined as *move robot to location*, *recognize and locate object*, *grasp object*, *move robot to delivery location*, and *hand over object*. This rather simple sequence captures the domain knowledge of the task. In real-world scenarios the logic encoded is complexer, including contingency handling and the many details that typically come with specific applications (e.g., pre-steps when approaching, interacting with humans, reacting specific to errors in different contexts). The value of this knowledge gets visible when trying to perform the same task by a different robot – e.g., featuring a different navigation approach or with fundamentally different H/W e.g. without a manipulator (asking for human help to realize the manipulation part). Being able to compose decoupled tasks with different functionality-providing skills and components pushed the idea of the ecosystem.

Context

The pattern is defined in the context of the coordination of closed software components. The components a robotic system consists of encapsulate the basic functionalities that are used to compose complex tasks.

Problem

The development of robotics behavior coordination tasks, making use of the functionalities provided by different components, poses the challenge of composable

behavior models. Software components realize functionalities that must be accessible for behavior task coordination. The behavior models representing those functionalities must therefore be composable to the tasks. The translation between the abstraction level from the functionalities within components, to services (coordination services), to the task behavior level has to be realized at some point to enable robotics behavior task development, decoupled from the realizing software components. Not addressing this translation leads to non-reusable and coordination interwoven software components, that are fixed to a specific application. With task models tightly coupled to the components. Thereby preventing the idea of a robotics business ecosystem.

Same as the software components developed and composed by different roles, the development of behavior models must be achievable in separation as well. Behavior models capturing the functionalities encapsulated within software components need to be developable and composable by different roles.

Solution

The pattern applies the concept of skills to robotics behavior coordination. Skills are behavior coordination models that realize the translation from functionality and services implemented within software components to the behavior coordination abstraction level. Therefore, skills offer an interface toward the tasks that uses generic (with respect to technology) domain-specific terms and make use of the coordination interface to access the functionality within the components. In contrast to the behavior tasks, skills directly access the coordinated software components via the coordination interface. The skill models contribute the functionalities realized by the software components and hide the concrete interaction with the software components.

This separation enables the definition of robotics behavior task models independent of any realizing software component. This is a further prerequisite to a successful robotics business ecosystem. Again, this enables the separation of roles that can now work independent of each other, knowing that their contributions will fit together later on. In this case, the role of the robotics behavior developer is able to capture the domain knowledge of the application in tasks, decoupled from the role of the technology provider or component developer contributing the functionality required to realize the tasks and the application.

The match between the building blocks are enabled via the introduced skills. As with the coordination services previously described, the skills also feature an interface (or type) definition to which both roles rely on. The behavior developer makes use of skills without the need to know which concrete realization of the skill or component behind the skill is used in the running system, while the technology provider realizes the skill according to the defined interface.

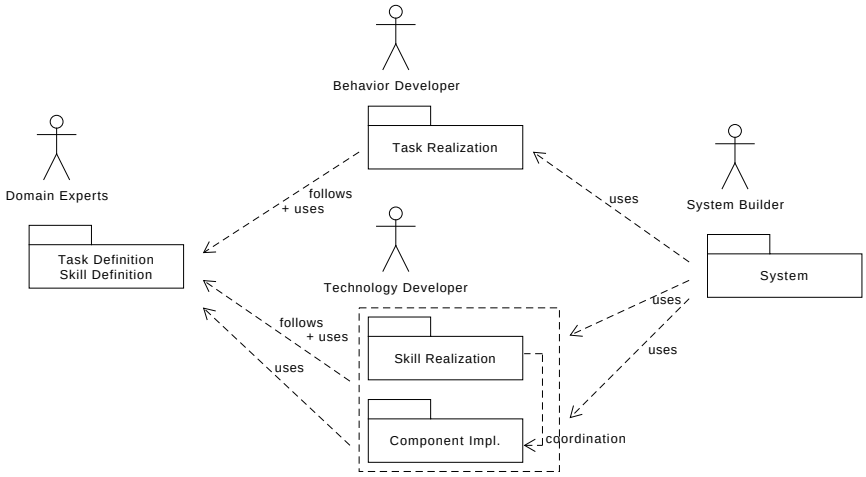

Figure 4.21.: Separated task and skill behavior models, overview illustrating the connections among the elements and the associated roles.

Consequences

As a major consequence, this pattern enables the composition of robotics behavior tasks using skill level behavior models that provide the functionalities encapsulated within components. Therefore, the behavior developer is freed from the burden to interface concrete software components and is able to focus on the development of applications. The developed task level behavior models are independent of any realizing software component, as well as the concrete realization of a skill. The pattern enables the separated work of the technology provider and the behavior developer and the composability of their building blocks. The separation of the tasks and skills further enables the use of different behavior coordination approaches without the need to redo the step change from services to skills for each new approach. The skills can be used as building blocks for the tasks. From a design perspective, the explicated interfaces of the skills enable a separated robotics task design phase. The explicated interface can also be extended to the realization of the tasks. The design and realization of tasks can be realized and tested without the need to have concrete software components and realizing skills. Mockup skill realization for testing the tasks can be used, or even generated automatically from the interface models.

4.6.4. Coordination Modules - Context for Composition

The pattern introduces *Coordination Modules*, abstracting the individual coordination interfaces to united modules containing skills and their used coordination interfaces toward the coordinated software components. The proposed pattern encapsulates the

skills and the components to form a functional entity to provide an execution context during runtime. This way, the coordination modules limit the sphere of influence of the contained skills to those coordination services defined to be used within the according module. The coordination modules as execution context coupled with the execution containers the components can be used to explicate resources. Coordination modules enable the coordination of multiple components that semantically belong together. The following describes the overall approach, more details and implementation remarks are described in Chapter 5.

Example

This example illustrating the pattern is situated in the context of object recognition for manipulation. Two competing and complementing recognition approaches are used in different contexts. The first approach can be used to recognize objects that are located fixed and structured in a known fixture. This approach is called “rack recognition”, it recognizes the fixture and not the individual objects to manipulate. The second approach can be used to recognize individual objects located inside a wrapping packaging and is called the “bin picking” approach. Both recognition approaches are used by the same robotic system, dependent on the application and the context the robot is operating within. The two approaches are realized using a different number of components; they, however, feature the same skills such as “recognize objects.” Both approaches can, therefore, be used interchangeably, at least from a perspective of robotics task coordination and the according interfaces. The robotics behavior coordination approach needs to be able to distinguish and use both approaches. In some cases, it is also necessary to make use of the same approach (and thereby eventually the components) in different setups, multiple times (e.g. the coordination of multiple cameras). As both recognition approaches differ in their realization, the coordination access to the components and thereby the realization of the skills differ for both approaches. In this example, the bin-picking approach is composed out of two components. The first one to detect the box the objects are located within, while the second one makes use of this information and recognizes the individual objects within the container.

The realization of the application by the role of the behavior developer is done without the need to choose any specific recognition approach. The tasking defines the use of two object-recognition approaches and uses them, both featuring the same skills in terms of the interface. Both approaches, “RECOG-BIN” and “RECOG-RACK,” are instantiated and the skills both components realize are used to set up the context, activate the recognition, and receive the results. This includes, for example, the configuration of the data sources (e.g., which camera to use) and which objects to recognize. The concrete coordination of components is realized within the realization of the skills provided by the recognition approaches. In case of the rack recognition, the skill “recognize objects" is refined to first perform a detection of the containing box and later recognize the objects within this box.

In this example, making use of both components realizing the functionality. The developer of the recognition approach is able to realize the skills accessing the coordination interfaces of both components belonging to the approach.

During system composition, the task models, with the named instances of the two recognition approaches ("RECOG-BIN" and "RECOG-RACK"), are composed to the components and skill realizations from the ecosystem marketplace. The best matching approach can be selected by choosing from competing approaches realizing the same interface, but with different non-functional properties, such as the two different types of recognition approaches. Both provide the same results but are tailored to specific scenarios and exploit the properties of those to optimize the result.

Context

The pattern is defined in the context of coordination of closed software components. The components a robotic system consists of, encapsulate the basic functionalities that are used to construct complex tasks. The pattern requires the separation of tasks and skills, and is reliant on the use of some kind of coordination interface toward the coordinated software components.

Problem

The development of applications based on behavior building blocks, namely skills provided by closed software components, is driven by the idea to make use of functionality abstracted from realization specifics. The realization of the offered functionality, however, always needs to deal with the limits of the physical world a robotic system operates within. This includes the physical limits and the resources of the robot itself as the very first thing. The resources are required and represented by the software components, providing the skills – e.g., a camera-driver component requiring the hardware resource camera and processing power to operate. The skills that offer or represent those functionalities need to consider these resources to some extent. The coordination of software components using skills, therefore, requires an execution context within which the skills can be executed. The execution of stateful skills – e.g., the setup skill of an object-recognition component –needs to be scoped to the coordinated object-recognition component providing and representing physical and logical resources. Otherwise, the coordination of complex robotic systems – e.g., consisting of multiple instances of the same component or precisely of multiple components using the same coordination service, would not be possible.

Solution

To overcome the problem of hidden context and resources, the explication of those is necessary. The *Coordination Modules* form the execution context for skills to be executed within. The context is needed to capture the resources necessary to provide the offered functionality. This context can be realized and captured in

different ways. Directly using the instance of the coordinated component as a scope for the execution context would be a reasonable choice at first glance. Two arguments call for the definition of a separated entity, namely the coordination module. First, the easy coordination of semantically correlated components that are best coordinated as one entity. In many cases, only a set of components provide a functionality and skills that is at a reasonable granularity for the tasking of robotics applications. A good example is a set of navigation components (e.g., planning, mapping, and collision avoidance) which together provide the functionality of moving a robot to a location. If those components would be coordinated separately, the individual components would need to provide fine granular skills used on the task level to coordinate each component individually, which would result in increased accidental complexity, that is not inherent to the problem.

In a service-orientated architecture, the services are decisive, not the components in which the services are realized. Therefore, the same functionality can be realized using a different number of components together again featuring the same services. As the second argument for the coordination modules, the separated coordination modules decouple the skills from the number of components to be used. Competing approaches can be used to provide the same skill, which is again a prerequisite for a working robotics business ecosystem. The number of components used for the realization of a skill is hidden from the behavior tasks.

The concept of *Coordination Modules* follows the idea of the type definition and realization as is done by the coordination services as well. Thus, the coordination modules are split into three parts, see Figure 4.22. First, there is the definition of the coordination module as domain knowledge. The coordination module definition wraps the interfaces of the skills the module provides. Additionally, the coordination module definition contains the provided and required services as resources for execution.

Second, the realization of the coordination modules, as is done by the role of the technology developer. The realization defines which coordination interfaces are used, as well as which instances of other coordination modules the module is using to realize its functionality. The coordination module realization further contains the realization of the skills. The additional instantiation of modules enables the use of skills from other modules. This enables the development of skills by making use of the functionality provided by other modules without breaking the encapsulation. The module instantiation by the coordination module realization, instead of the definition, pulls the dependency to a specific realization and not the generic definition, which adds to the composability of the modules and the skills.

Third, the abstract instances of the coordination modules, used for the development of tasks. The abstract instances are named elements to allow for the use of distinguishable instances of the coordination modules – e.g., the coordination of

two manipulators. When composing a robotics system, a mapping between the realization and the abstract instances needs to be defined. This mapping defines which abstract instance in the task is mapped to which module realization. This includes the mapping of multiple instances of the same module realization. Additionally, the instances of the coordination modules defined within the coordination realizations (other modules used to realize those modules) must be mapped during system composition.

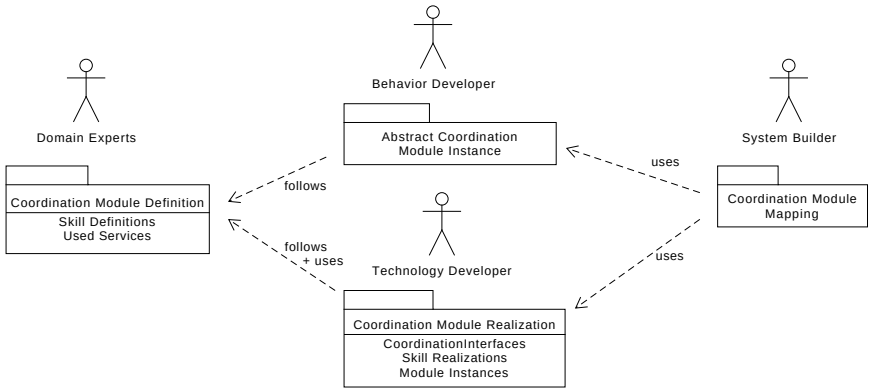

Figure 4.22.: Coordination modules, overview illustrating the connections among the elements and the associated roles.

Consequences

As a main consequence, the pattern enables the use of the same skills multiple times in the context of different instances of the components. The execution context provided by the pattern can further be used to coordinate semantically correlated components together, avoiding fine-grained skills and adding complexity.

5. Skills and Coordination Modules

Robotics behavior coordination and the tasking of robots need building blocks to compose systems and applications. The functionalities provided by and encapsulated within components are made accessible for coordination at the abstraction level of services via the coordination interface, see Chapter 6. For the composition of robotics behavior tasks in the sense of a working robotics business ecosystem, the behavior tasks need to be reusable in combination with different components. For example, a functionality "move robot to location," as is used at the abstraction level of robotics task, can be realized using different components, dependent on the navigation approach. To enable this separation, this chapter applies the concept of skills, decoupling the robotics behavior tasks from the components and their coordination interface. This chapter contributes the structures for the skill level building blocks and their the coordination modules. The structures are motivated by the idea of a robotics business ecosystem and the need to separate the work of the different roles involved in developing a robotics system by managing how their individual contributions can be composed. The concept structures are expressed as meta-models using block-port-connector as well as Ecore [Ste+08] and its graphical notation modeled with EcoreTools [Fouc]. However, they can be expressed and implemented in other modeling approaches as well.

Figure 5.1 shows an overview of the different involved roles. The main roles involved are the robotics behavior developer and the component developer. Both roles contribute to the behavior models, the component developer on skill abstraction level and the behavior developer on task abstraction level. The domain experts define the interfaces and types the other roles rely on. Finally, the system builder is involved in composing the decoupled parts to a concrete working system.

The chapter is organized in three sections, starting with the introduction of the concept of skills as basic robotics behavior coordination building blocks. It is followed by the coordination module concept forming the execution context of the introduced skills. The third section presents the task-level behavior models making use of the skills. Further, some integration and runtime relevant aspects when dealing with skills are presented in the appendix A.1.

5.1. Skills

The term "skill" in computer science is used for different things. Even in the robotics domain the details of what a skill represents are not used consistently. The common

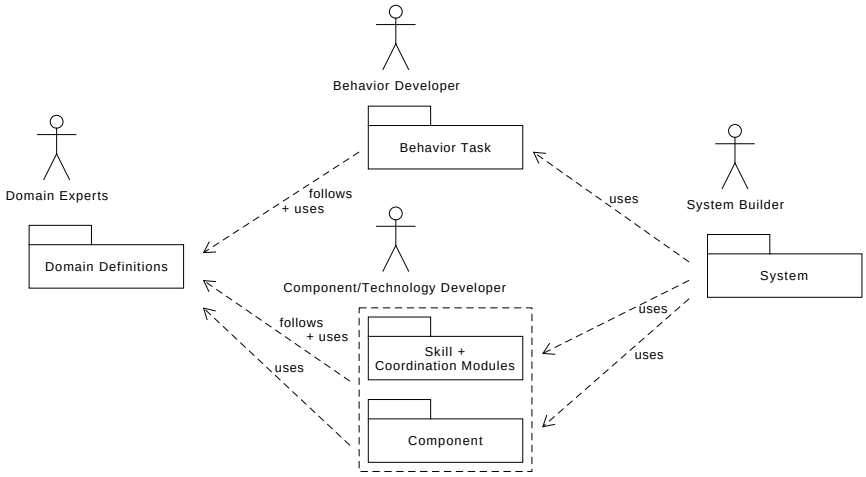

Figure 5.1.: Overview on skills and coordination modules illustrating the links among the elements and the associated roles.

ground on which most skill definitions are based is close to the English definition of a skill or ability, Oxford English Dictionary: "A particular ability" or "possession of the means or skill to do something."
A skill used within this work also rests on this definition as a skill represents a particular ability. The definition of a skill used for this work is more technical and connected with the context:

A skill represents a capability that is provided by one or a set of software components. A skill provides a generic functionality to be used by robotics behavior tasks and realizes specific coordination actions towards components. Therefore, skills lift the level of abstraction from concrete and individual configurations of functionalities and services - e.g. robot movement or navigation - to a more abstract level where capabilities are named in a way independent of their implementation, thereby providing access to the functionality for robotics behavior coordination on the task abstraction level.

Following the overall idea of composable coordination in an ecosystem context, as described in the previous chapter, skills are defined, realized, and used by different roles. The following sections will introduce how the individual roles are linked to skills and how the interfaces between the different roles are dealt with, see also Figure 5.2.

5.1.1. Skill Definition

A skill definition is a formal structure that defines the interface of a skill. It is required to make use of a skill without the need to know how and by whom a skill is realized. This separation of the interface from the realization enables the replacement and composition

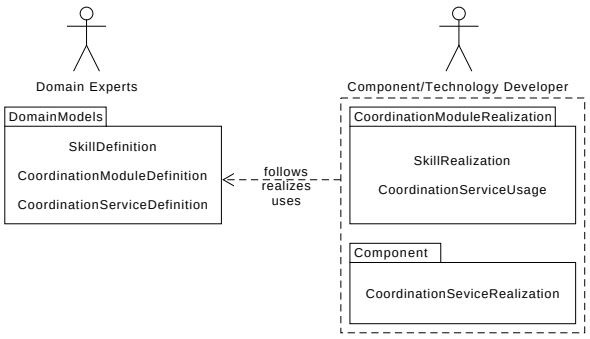

Figure 5.2.: Overview on skills, separated into the definition of a skill, providing the upwards interface, and the realization of a skill, providing the coordination and connecting towards the component.

of components (realizing the same skill) to systems and decouples the role of the component developer from the behavior developer. In addition, this separation enables the possibility of utilizing different robotics behavior coordination approaches while reusing the skills as interface and abstraction to functionalities realized within components.

Skill definitions are part of the domain models and capture the consolidated domain knowledge expressed by the role of the domain expert. A skill definition represents the consolidated interface of how a concrete functionality provided by a software component can be accessed for robotics behavior coordination. The skill definition meta-model (see Figure fig:skill-definition-metamodel block-port-connector notation and Figure 5.4 in Ecore) defines the interface of a skill. It is kept simple to allow for a wide technology-independent adoption, which, however, captures the pivotal parts to make use of a skill. Further extensions like capturing non-functional aspects are possible and can extend the presented structure.

Skill definitions are modeled using a Domain-Specific Language (DSL). The syntax of the DSL is of relevance to allow for a user-friendly handling in the tooling. However, from a structural and conceptual point of view, its detailed syntax is not decisive. The language follows the structure defined in the presented meta-model. The DSL and its syntax are presented with the experiments in Chapter 8.

The meta-model class of a *SkillDefinition* is a named element as it is referenced by other models and roles. Multiple skill definitions are grouped within a coordination module as a logical container, which will be presented in the later section of this chapter. The *CoordinationModuleDefinition* contains instances of two classes to model the required and provided services used by the coordination module. Both contain a reference to a *CommunicationSeviceDefinition*, defining the communication semantics and the data type of the service. The two **ServiceRef* classes are used to deal with connections between components spanning across coordination modules, their usage is detailed in Section 5.2. Semantically correlated skills (in the sense of a domain such as wheeled indoor

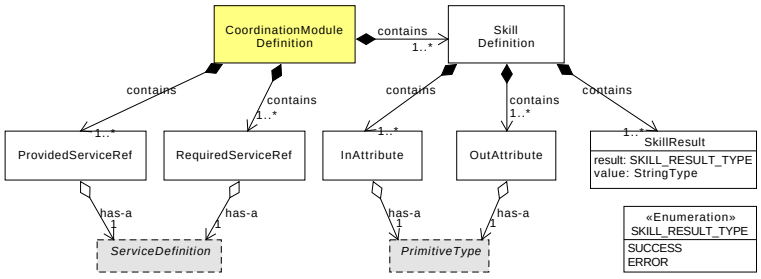

Figure 5.3.: Skill definition meta-model (block-port-connector).

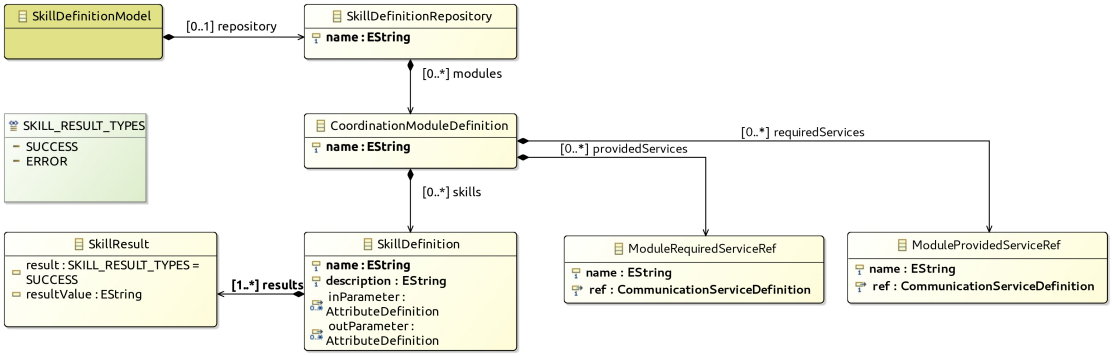

Figure 5.4.: Skill definition meta-model (Ecore), modeling the interface of a skill.

navigation) are grouped in a skill definition repository. Each skill features a description in form of a human readable text, used to help the users to understand the intention of the skill definition. The main elements a skill contains, are its name, in and out parameters and a result value. The parameters are used to parametrize a skill and to receive values from a skill after its execution completed. This is required to use skills by tasks or other skills, realizing an application by composing different skills. To keep the implementation compatible and technology-agnostic, only primitive data types are usable as parameters. A skill parameter is a simple name value pair using a simple data type model. An example of how to model such data types can be found in the coordination interface Chapter 6.4.4. The skill result value is of Boolean type, which means that as a common denominator the execution of a skill can either be successful or failing. To allow for further refinement of the different cases, each skill result contains an additional result value that can be returned.

5.1.2. Skill Realization

A skill realization models how a component or a set of components is used and coordinated to realize a particular skill. In this way, a skill is realized following the skill definition. Skill realizations are contained within coordination module realizations, as

they are defined by the skill definitions and their containment within a coordination module definition. In contrast to the skill definition, a skill realization as an entity is used to express business logic. The skill realizations can be implemented using different approaches, ranging from dedicated tools e.g. state machines to simple programming. The approaches can be implemented using different technologies – from libraries in general-purpose programming languages to dedicated DSLs or GUI workbenches. Skill realization follows a skill definition on the upper end and the component coordination interface to access the components on the lower end.

Skill realizations are building blocks used to build robotic systems, just like software components. A skill and its realization supplements a component or a set of components as a building block providing functionality to a system. It enables the use of the building blocks by composition for robotics behavior coordination and thereby the tasking of the robotic system and its application. Consequently, skill realizations are modeled by the role of the technology experts. A new role specific to the development of skill realizations can be defined as the used interfaces of a skill realization decouples the potential role from others. However, there exists a large overlap in the knowledge required to develop a skill realization with the role of the component developer. A skill realization uses components to realize a particular skill by coordinating one or multiple components. The coordination of the individual components is done via the coordination interface of the components. The concrete coordination of the components, e.g. the usage of particular parameters, is knowledge that is best known by the developer of the particular software component. Besides the fact that the role has the inside knowledge of how to use his own component, the role also has to test the component and its coordination interface during development. This is best done by developing skills that use the coordination interface and the software component. The users of both the component and the skill-building blocks will later rely on the same interface.

The following meta-model depicted in Figures 5.5 and 5.6 (bblock-port-connector and Ecore) defines a generic meta-model for skill realizations. It does not include the parts dependent on the realization approach of the skills. A full meta-model, for example, pertaining to skill realization adopting the task net approach SmartTCL [SS10] is shown in Appendix A.3.

The generic skill realization meta-model can be logically partitioned into the following parts: The skill realization part itself, the coordination interface action part, and the coordination module realization part.

SkillRealization - Upper Dock/Interface and Business Logic The *first block*, with classes representing the skill realization itself depend on how the skill realizations are implemented. The outer interface following the skill definition is fixed. Therefore, a *SkillRealization* is a named element and again there is a need to reference it by others. It contains a reference to the skill definition it is realizing. It has access to the in and out parameter as well as the modeled skill result value. The link to the skill definition is further required to allow for direct tracing of the required and realized skill realizations in a system.

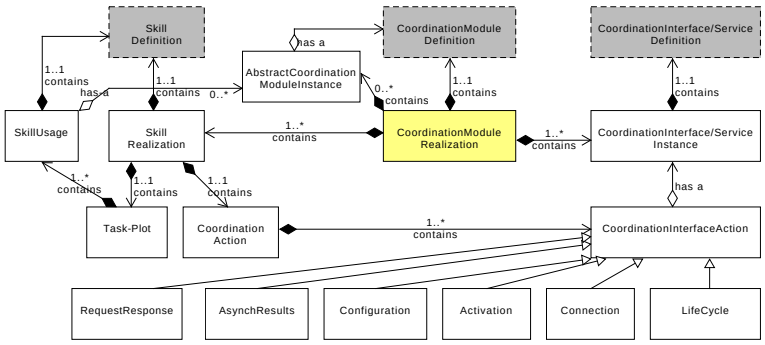

Figure 5.5.: Skill realization meta-model (block-port-connector).

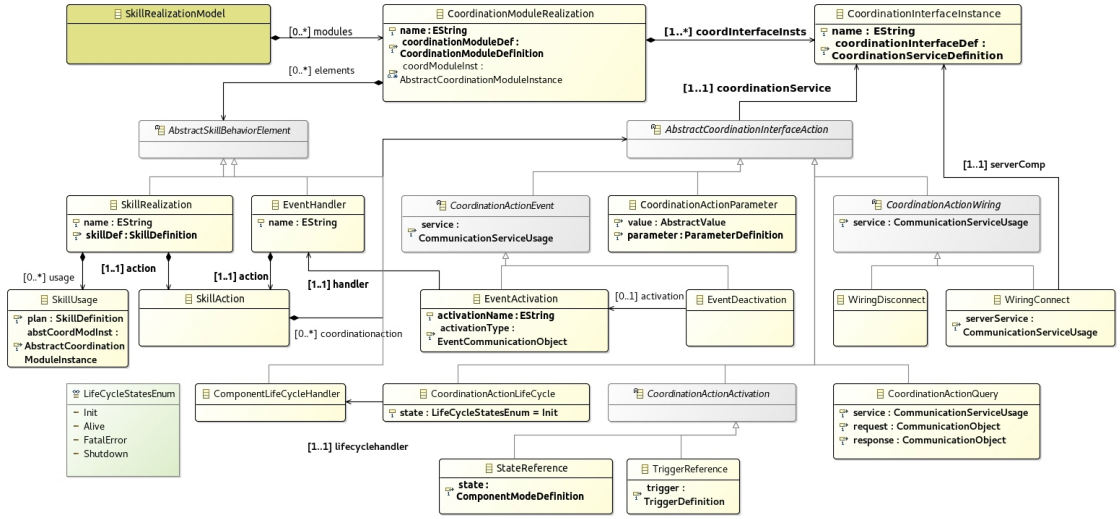

Figure 5.6.: Generic skill realization meta-model (Ecore), modeling the mandatory parts required to realize a skill.

Irrespective of the realization, each skill realization contains a *SkillAction* class, where the business logic of the coordination is expressed and the coordination actions using the components' coordination interface are performed. Besides skill realization, another implementation-independent entity is the *EventHandler*. The event handler is driven by the need to deal with the asynchronous nature of the coordination approach using the already defined component coordination interface. Activities are activated within the component, and the coordinator using the skills is waiting for the results captured by the event handlers. In addition to those, skill realization contains a *SkillUsage* class to enable the use of other skills with skill realization. The skill usage itself refers to a skill definition as well as an AbstractCoordinationModuleInstance to make use of a skill located within another coordination module.

SkillRealization - Low Dock/Interface The *second block* is the part modeling the coordination interface actions, accessing and coordinating the components to realize a skill. This part is independent of the realization technology of the skills, as any realization has to access the component coordination interface. The elements within that part are all derived from the abstract supertype *AbstractCoordinationInterfaceAction*, containing a link to the used *CoordinationInterfaceInstance*. The link is contained by the *CoordinationmoduleRealization* which also contains skill realization. Thus, a skill cannot access the coordination interface of components outside of its own coordination module realization. Following the concept of the coordination interface, with its fixed set of patterns to access the component, every coordination action is one of the following types. None of the derived classes are named as they are used as a reference to the modeled elements within the domain models (coordination service) only.

The *CoordinationActionAcivation* class is used to activate activities within components. The activities can either be cyclic or one-shot, thereby referencing the states or triggers modeled within the coordination service definition within the domain models, see Chapter 6 for further details. The two derived classes— *StateReference* and *TriggerReference*—are used to do so.

The *CoordinationActionLifeCycle* class is used to coordinate the lifecycle states of the components. Dependent on the used component model and the associated lifecycle automaton, more or fewer states are accessible. A minimum set is assumed here with the states *Init*, *Alive*, *FatalError*, and *Shutdown*. The class contains a link to the *ComponentLifeCycleHandler* used to communicate asynchronous notifications of the coordinated component-induced lifecycle state changes to the coordinating component. Further details can be found in the component lifecycle section of the coordination interface Chapter 6.9.

The *CoordinationActionWiring* class is used to connect to the runtime wiring port of the component coordination interface. The wiring is limited to the components and services within the current coordination module in line with the overall sphere of influence of a skill. This is required to enable the principle of separation of roles, thereby allowing for the composition of components and skills. The role of the technology developer is not able to and not obligated to know services within other modules, as the role does not deal with systems and hence the composition of multiple modules and components. If a connection between the two components of different coordination modules is required, the modules need to explicate this in the coordination module definition. The using part on the task abstraction level is able to use this explicated required and provided services to wire the connection during runtime. The *CoordinationActionWiring* class contains a reference to the *CommunicationServiceUsage* to link to the specific service of a component to be connected or reconnected. Only the connector side (service requestor) of the service can be reconnected; therefore, when realizing the tools and DSLs, the elements need to be filtered to be of the connecting side only. The class features two derived classes for connecting and disconnecting. The connecting side is defined by the modeled coordination interface instance via the superclass *AbstractCoordinationInterfaceAction*

and the *CommunicationServiceUsage* reference. The side to connect to is also defined by a coordination interface instance in the *AbstractCoordinationInterfaceAction* class and a modeled *CommunicationServiceUsage* reference.

The *CoordinationActionParameter* class is used to access the component configuration via the coordination interface. It contains a reference to an instance of a *ParameterReference* class referencing a domain model-defined component parameter set. Additionally, the class contains values to be set for the referenced parameter. Following the component coordination interface, only primitive data types are used to allow for easy implantation and adaption by different realization approaches. The value class is not detailed here, the definition of the used simple data type system can be found in the coordination interface Chapter 6.4.4.

The *CoordinationActionQuery* class is used to connect to the information query part of the component coordination interface. The class contains a reference to the *CommunicationServiceUsage* used, as is defined in the coordination interface. This is required because a single coordination interface can contain multiple query services, while the used one needs to be defined here. Additionally, the class contains a request and response attribute.

The *CoordinationActionEvent* class is used to connect to the asynchronous communication mechanism of the component coordination interface, used to wait for the results of an activation within a component. The abstract class contains a reference to the event service defined within the coordination service definition. The two derived classes are responsible for activating and deactivating the events. The class *EventActivation* is a named class as it models a specific event activation, which needs to be referenced from the runtime interface of the component coordination interface as well as from the *EventDeactivation* class. Multiple events on the same source can be activated, see Chapter 6.7 for further details. The class further contains a reference to an instance of the named class EventHandler. This link is required to map the activation of an event to the EventHandler that would later be executed during the runtime execution of the model, once a component notifies the skill of the activation results.

SkillRealization - CoordinationModuleRealization The *third block* is centered around the *CoordinationModuleRealization* class containing all the other elements of the skill realization model. The class is named as it is referenced during system composition. It logically groups the realization of all skill elements and connects the abstract *CoordinationModuleDefinition* with the instances of the coordination interfaces used within this specific realization. Therefore, it contains multiple instances of the *CoordinationInterfaceInstance* class. The *CoordinationInterfaceInstance* class represents the use of the coordination service definition defined in the domain models. The class is a named element as it is referenced by all coordination interface actions via the *AbstractCoordinationInterfaceAction*. A dedicated class is required as the same coordination interface can be used multiple times within a coordination module; therefore, the instance needs a name. An example of this case could be a navigation module containing two laser rangers that need to be coordinated explicitly and activated when driving forwards or backwards. The class

further contains multiple instances of the *AbstractCoordinationModuleInstance* class to model other coordination modules this module is using. This enables the skills contained in the module to make use of skills from other coordination modules. To do so, those skills need to be contained in a coordination module to provide an execution context.

5.2. CoordinationModules

Coordination modules form the execution context the skills are operating within. They abstract the individual components from the perspective of a task abstraction level, hiding their coordination interfaces. Coordination modules limit the sphere of influence of contained skills to those coordination services defined within the related module. Stateful skills require a context to be executed within e.g. object recognition components offer skills to set up, detect, and query information from, those skills are stateful and context dependent, the execution of a skill changes the state of the components. The coordination modules as execution context, which are coupled to the execution containers – the components, are also used for the explication of resources.

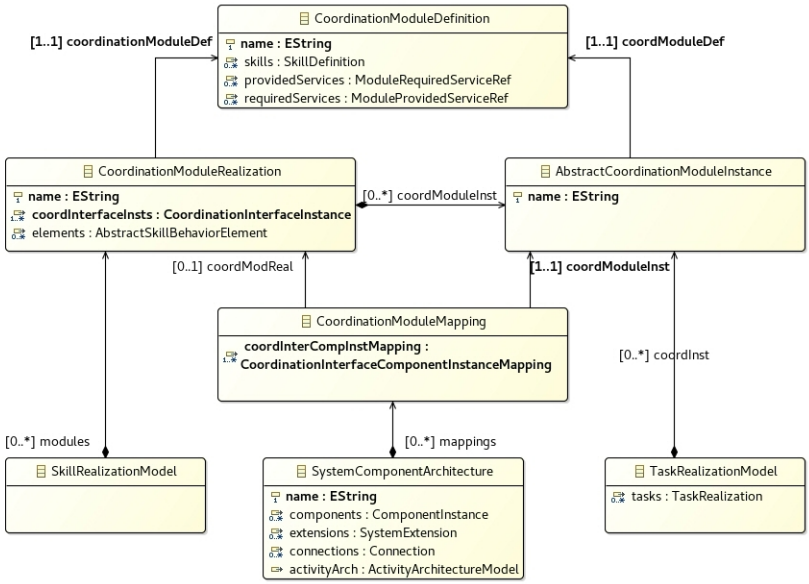

Figure 5.7.: Coordination module overview meta-model, showing the relations between the different parts involved.

The coordination module as a container enables the direct coordination of multiple components without the need to use fine granular skills explicating otherwise hidden access. This could also be achieved through composite components, in which case the composite component would be the container. The advantage of the presented approach is the separation of the component and the coordination module. Thus,

the skill realization is not bound to a specific component implementation but to a component-independent coordination module and its coordination interfaces. This adds to the composability of building blocks considering robot behavior coordination and the tasking of robots.

The concept of coordination modules does not reduce the compositional “possibilities" as components within such modules can still be replaced by others. It, however, eases the coordination of correlated components and makes the development of robotic applications more straightforward. A very intuitive example for the combination of multiple components to a single unit is a typical service robotic navigation approach. They typically consist of an online mapper, a path planner, and a reactive local obstacle-avoidance component. On the robotics behavior task level, the skill that is typically used is something like “move-robot to a location.” To realize this, all the three components need to be coordinated to perform the robot movement. Without the grouping, each component would need to offer fine granular skills to allow coordination, which would increase the effort. The grouping of skills within coordination modules is further driven by the perspective of the robotics behavior coordination developer. Combining skills to a higher level of abstraction, when being able to coordinate multiple components directly, makes the development of those skills easier. Thus, the robotics behavior developer benefits from a higher level of abstraction and more capable skills available for task composition (the tasking of the robot).

One could ask why the component is not used as a container, that deals with the resources the coordination module represents. While this is possible, the granularity of a component and the functionality it is providing are not at the right level in many cases. This can be shown with the example of a robotics navigation approach, where mapping, planning, and collision-avoidance components are required to create a skill “move robot to location.” None of the components alone would be able to realize the relevant functionality. The skill making this functionality accessible must use all the three components and must deal with them all them as one usable entity. Another approach to realize the united structure would be the idea of a composite component wrapping other components. The composite would then need to deal with the same requirements the coordination module is dealing with. The introduction of the coordination module as a dedicated entity can also be connected with the concept of composite components, following the principle of separation of concerns which is very reasonable.

The coordination modules connect to the idea of an explicated interface, with a defining type. Others are able the use of the interface without knowing the realization by making use of the definition. The following introduces the core meta-model elements modeling this concept. An overview of the core elements of the concept of the coordination module is expressed in Figure 5.7, which is an excerpt of the full meta-model. The meta-model is primarily defined by the following four coordination module-related classes.

The *CoordinationModuleDefinition* is part of the domain models defining the interface

of a coordination module. The *CoordinationModuleRealization* is part of the *SkillRealizationModel* developed by technology developers that also develop the components. The *AbstractCoordinationModuleInstance* is part of the *TaskRealizationModel* expressing instances of the coordination modules used by the task-level behavior models. The *CoordinationModuleMapping* is part of the *SystemModel* mapping the provided skill realizations to the abstract ones used by the task-level models, which are developed during the system composition by the system integrator. The following sections detail the sketched parts and how the roles are involved. The proposed structures are realized and made accessible to the user via a group of DSLs. The DSL are realized using Xtext [Foud], following the presented meta-models. The meta-models are presented using the graphical notation of Ecore [Ste+08], which, however, could be expressed and implemented in other modeling approaches as well.

5.2.1. CoordinationModule Definition

A coordination module definition defines the interface of a coordination module used by all involved roles, with Figure 5.8 showing the coordination module definition meta-model. It captures the consolidated domain knowledge expressed by the role of the domain expert. The separation of the interface from the realization enables the separation of roles for robotics behavior development. The role of the robotics behavior developer, working on task-level robotics behavior models, is able to use the coordination module definitions to make use of the functionality encapsulated within the module. This is done without binding a robotic task to a specific realization of functionality expressed by a skill. For example, a "move robot to location" model can represent a transportation task, without having any implementation of such a skill yet and without a need to know by whom it is realized. A set of components can, for example, realize this by moving a robot with wheels or flying a robot through the air. The coordination module definition defines the context and namespace for the skills.

The coordination module definitions and hence the skill definitions need to be independent of any coordination interface type (coordination service definition). Otherwise, the skills would be bound to specific realization approaches. Not necessarily to a specific component, because the coordination interface is per se not bound to components. However, a specific coordination interface would bind the skill definition to a certain type of component to realize a skill, as the coordination interface reflects some specific properties of the class of coordinated components. A wheeled robot will reach a location differently than a flying one. The skill definition for reaching a location might be reused, while the coordination interface for the coordinated component will most certainly look quite different for the two cases.

The *CoordinationModuleDefinition* is a named class as it needs to be referenced by other classes, used by other roles for realization and usage of the module. The class contains multiple instances of the *SkillDefinition* class, thereby modeling which functionality the coordination module contains and the robotics behavior developer can make use of. The

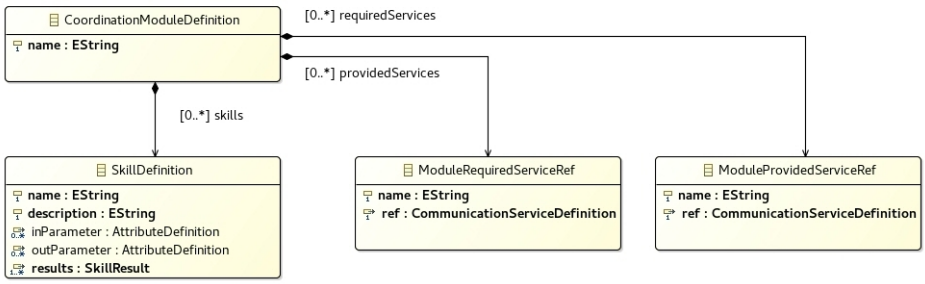

Figure 5.8.: Coordination module definition meta-model, showing the interface of the coordination module.

CoordinationModuleDefinition further contains instances of the classes *ModuleRequiredServiceRef* and *ModuleProvidedServiceRef*, modeling the communication services used and provided by the coordination module and a component within the coordination module, both of which model the inter-module communication dependencies. The explication of those resources is essential for the runtime connection of the services. Task-level behavior models need the ability to connect or reconnect those, although the service endpoints are hidden to them inside the module.

5.2.2. CoordinationModule Realization

Coordination module realization models the container used to realize the module and, more importantly, the skills contained within the coordination module. Figure 5.9 shows the meta-model of the coordination module realization, which binds the independent definition of the module to specific coordination interface instances. This is the foundation to enable the realization of the skills by making use of the functionalities of the components, using the components' coordination interface. The coordination module realizations are modeled by the role of the technology developer who also develops components and models skill realizations.

The named class *CoordinationModuleRealization* is the central element in this meta-model. It needs to be named as it defines a specific realization of the realization-independent coordination module and skill definitions. This realization needs to be referenced during the system composition to abstract instances and mapped to the coordination module definitions used by the task models. Mapping by type only is not sufficient since multiple instances of the same module could sometimes be used even with different realizations. For example, two different types of object recognition approaches follow the same definition. The coordinating access to the components is realized via the coordination interface of the components and therefore the module contains instances of the *CoordiantionInterfaceInstance* class. The *CoordiantionInterfaceInstance* class follows a *CoordinationServiceDefinition* belonging to the domain models and expresses the instantiation of the coordination interface of the component. The

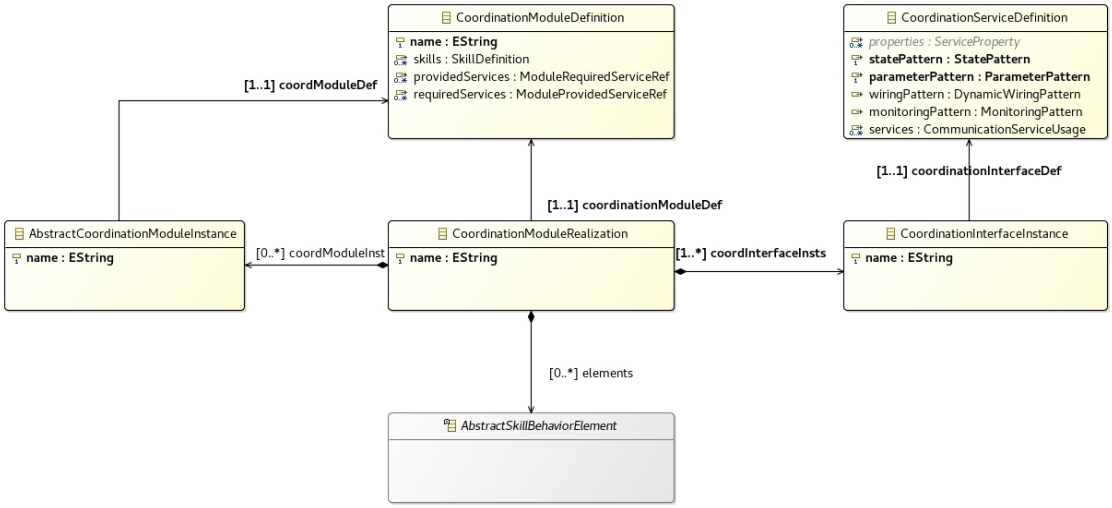

Figure 5.9.: Coordination module realization meta-model, binding the realization-independent module definitions to specific coordination interface instances.

class *CoordinationModuleRealization* further contains all the classes used to realize skills grouped by the abstract upper class *AbstractSkillBehaviorElement*. Further details can be found in the skill Section 5.1. In addition, the class *CoordinationModuleRealization* further contains multiple optional instances of the class *AbstractCoordiantionModuleInstace* to model the use of other coordination modules and their skills. Therefore, the module needs to be instantiated to provide the context for the skills in the same way as it is done for the task models. The module instantiation by the coordination module realization, instead of the definition, moves the dependency to a specific realization and from a generic definition, thereby adding to the composability of the modules and skills.

5.2.3. CoordinationModule Instantiation

The coordination module instances model the abstract and implementation-independent, instantiation of the coordination modules, with Figure 5.10 showing the related meta-model. The models are defined by the role of the robotics behavior developer to make use of the skills that are explicitly defined in the coordination module definitions. In this way, the role realizing the building blocks providing the functionality and the role realizing the tasking or behavior coordination of the robot are decoupled from each other. Both roles follow the same definitions from the domain-level models. The instances of the class *AbstractCoordinationModuleInstance* representing the coordination module instances are, therefore, part of the class *TaskRealizationModel*; they express instances of the coordination modules used by the task-level behavior models. This enables the reuse of robotics behavior task models for accessing and using the functionality provided by the components via skills while not being bound to specific realizations of the skills and

the components providing them. For example, an intralogistics order-picking task can be modeled using the navigation components' skills to move the robot, without being bound to a specific kind or realization of navigation.

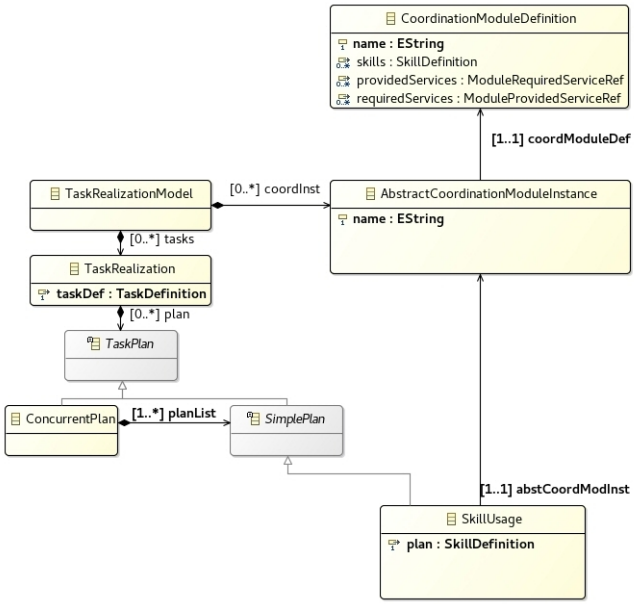

Figure 5.10.: Coordination module instance meta-model, enabling the realization-independent usage of skills in robotics behavior tasks.

The class *SkillUsage* is part of the task realization, and the instances are contained within a *TaskPlan* representing a refinement of a task. The task realization meta-model depends on the approach chosen to realize robot behaviors. The presented meta-model focuses on how the task can make use of skills contained by the coordination modules and does only contain the necessary elements of a sketched task realization meta-model. To make use of a skill, the class *TaskPlan* references a *SkillDefinition* instance contained within a coordination module and is therefore linked to the execution context defined by the class *AbstractCoordinationModuleInstance* of which the *SkillUsage* features a reference to. A task-realization block named "transport" would, for example, use a skill named "move robot" contained in the coordination module instance "navigation1". Another example could be a robot with two arms grasping two objects called "arm1.grasp-object 1" and "arm2.grasp-object 2". This example makes use of the skill "grasp-object" with the objectID as parameter and the two coordination module instances "arm1" and "arm2", explicating a duplicated instantiation of the manipulator components. The *AbstractCoordinationModuleInstance* follows the definition of the *CoordinationModuleDefinition* and contains a reference to the one it is instantiating. Using the abstract instances following the *CoordinationModuleDefinition*, the task level behavior models are able to make use of the provided and required services spanning across the modules' boundaries. There-

fore, the behavior developer is able to configure the dataflow between the components of different modules. As the provided and required services are explicated with the CoordinationModuleDefinition, they are part of the interface that a module offers. This again prevents the binding of task-level behavior models to specific realizations. The inner module connections among the components are dealt with on the skill behavior level and are not exposed to the task level.

5.2.4. CoordinationModule Mapping

The coordination module mapping bridges the gap between the abstract instances used for robotics task coordination by the role of the robotics behavior developer and the realizations developed by the role of the technology developer. Figure 5.11 shows the related meta-model, with the coordination module mapping being part of the system modeling. During system composition, the role of the system builder selects the building blocks required to compose the system. The selection consists of the components along with the skills inside the coordination module realizations and the behavior tasks. To enable the execution of the robotics behavior task models using the skills, the mapping of the abstract coordination modules to the realizations provided by the components needs to be done. Let's illustrate that by the example of navigation, the abstract coordination module instance "NAV1" using the skill definition "move-robot" needs to be mapped to those components providing the coordination module realization, following the same module definition. If multiple instances of the same coordination module mapping are used within the robotics behavior task models, the mapping also expresses logical information that cannot be derived by the coordination module types automatically. In a manipulation scenario with two arms the mapping of the coordination modules binds the physical arms to their logical representation on task level. Further techniques such as reasoning or learning mechanisms could be used to automate this selection. They could make use of the explicated resources modeled with the coordination modules. However, in many cases the explicated definition is sufficient and adequate.

The central unnamed class *CoordinationModuleMapping* models the mapping between the *AbstractCoordinationModuleInstance* class and the *CoordinationModuleRealization*; it, therefore, contains a reference to both. The mappings per se are unnamed as the identification of the mapping is not required to make use of the mapping during the execution of the task behavior models. As the *CoordinationModuleRealization* might contain multiple abstract coordination modules itself, to model skills and modules used for the realization of the module, those need to be mapped to coordination module realizations as well. Since a coordination module can contain multiple instances of the same coordination interfaces type—e.g., the same component can be used within the coordination module twice—multiple instances of the class *CoordinationInterfaceComponentInstanceMapping* are contained by the *CoordinationModuleMapping*. To do so, the class *CoordinationInterfaceComponentInstanceMapping* itself contains a reference to both the *CoordinationInterfaceInstance* modeled by the role of the technology developer within the *CoordintionModuleRealization*

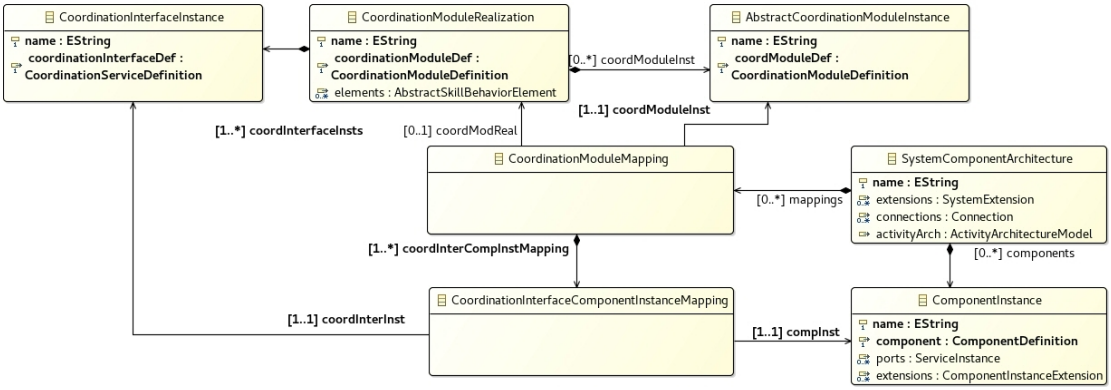

Figure 5.11.: Coordination module mapping meta-model, mapping the abstract module instances to the realizations provided by the technology developer.

and the *ComponentInstance* modeled by the role of the system developer contained in the central class for system composition *SystemComponentArchitecture*.

5.3. Task Level Behavior Models

The introduction of the skills separates the behavior coordination models into two abstraction levels, skills and tasks. Skills make the functionality realized within components accessible for the tasking of the robot. Skills as behavior building blocks are not bound to specific applications. Tasks, on the other hand, realize applications and encode how to utilize the robotic system to provide the desired service and behavior. Tasks encode domain-specific knowledge about the way something is done in a realization-independent manner. The task models, are also split into a definition and realization part for the same reason as for the skills, namely to enable a decoupled usage. The task definition meta-model is shown in Figure 5.12, the task definitions are similar to the interface of the skills. It provides a simple and abstract common interface for wide technology-independent adoption.

The meta-model of task realization is shown in Figure 5.13. The meta-model contains the main class *TaskRealization*. The main class features the attribute *TaskDefinition* to link to the interface, a *TaskPlan* to refine the task, and a *TaskAction* to enable the usage of logic to realize the task. The *TaskAction* is not further detailed in this abstract concept as the expression of logic depends on the realization of the coordination approach. The tasks are, however, bound to be evaluated to the results given in the task definition, with the overall result being either *success* or *failure*. The *TaskPlan* is further detailed to either support concurrent or sequential execution of sub-blocks, which could be skills or tasks. Whether to realize concurrent execution again depends on the realization of the coordination approach. A plan entry can be another *TaskUsage* or the usage of a skill (*SkillUsage*). Therefore, the model has to define which coordination modules to

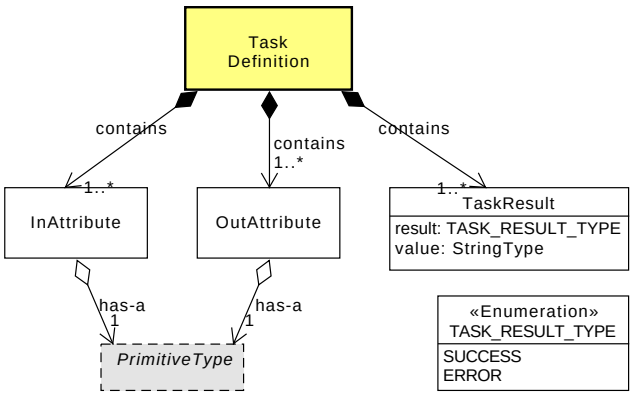

Figure 5.12.: Task Definition Meta-Model, interface structure of a task.

instantiate to use the contained skills. The classes *AbstractCoordinationModuleInstance* and *SkillUsage* enable this. Both reference the definitions to use the skill decoupled from their realization, see Chapter 7 for the organization of the development.

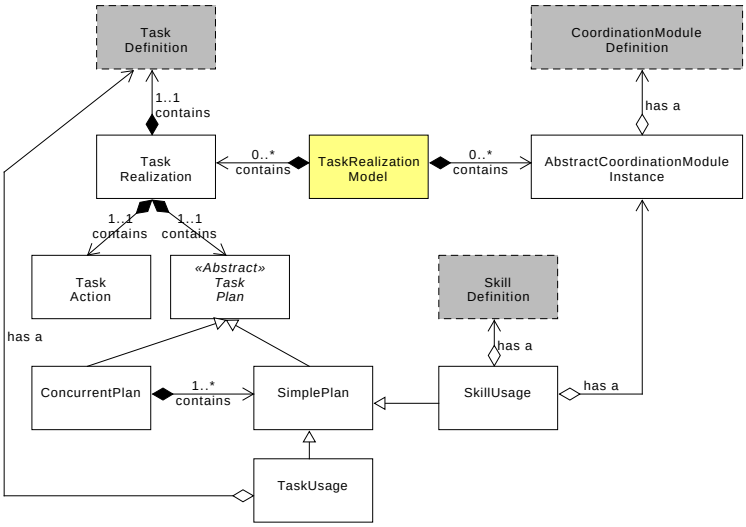

Figure 5.13.: Task Realization Meta-Model, using skills to realize the tasking of the robot.

6. Component Coordination Interface

The coordination of functionalities encapsulated in components requires their accessibility. To ensure the composability and the working coordination of a system composed of building blocks developed by separated roles, this access needs to be at the right level of abstraction, and needs to follow guiding structures. Within this chapter, a uniform behavior coordination interface is proposed, structuring the coordination access to a component. The behavior coordination interface for robotic software components is two-folded, the coordinating component part and the coordinated component part. The coordinating component part is typically used by a sequencer component if a 3-tier architecture is realized. The purpose of the interface is the connection of the component wrapped functionality with the robotics behavior coordination on skill abstraction level. The interface raises the level of abstraction (to services) and harmonizes the coordination-wise access to the components and the functionality encapsulated by them. This allows to separate the behavior coordination and behavior models from the functionalities and their realization.

Further, the interface helps to decouple the used coordination approach (e.g., state charts, behavior trees, etc.) from the coordinated components. The coordination interface allows for the composition of the components to systems and applications, without the need to develop a custom behavior coordination interface every time from scratch, thereby harmonizing the coordinating access by introducing a structure with defined semantics. The structures help to follow the principle of separation of concerns, avoiding interwoven coordination and business logic within software components. These unified structures are in turn a crucial prerequisite for a robotics business ecosystem, dependent on the composability of the building blocks it contains.

The coordinating access to a component can be grouped into six basic categories, each with a different use case, semantics, and communication mechanism. These categories and how the proposed coordination interface is handling them are described in detail in this chapter. Figure 6.1 illustrates the proposed interface and how it is connected to the entities on their respective sides – coordinating and coordinated components. The chapter further describes the combined usage of all six categories for system orchestration, for robot tasking. The following six parts structure the interface:

Configuration
: Run-Time configuration of components, for coordination.

Activation
: Activation of functionalities within the components.

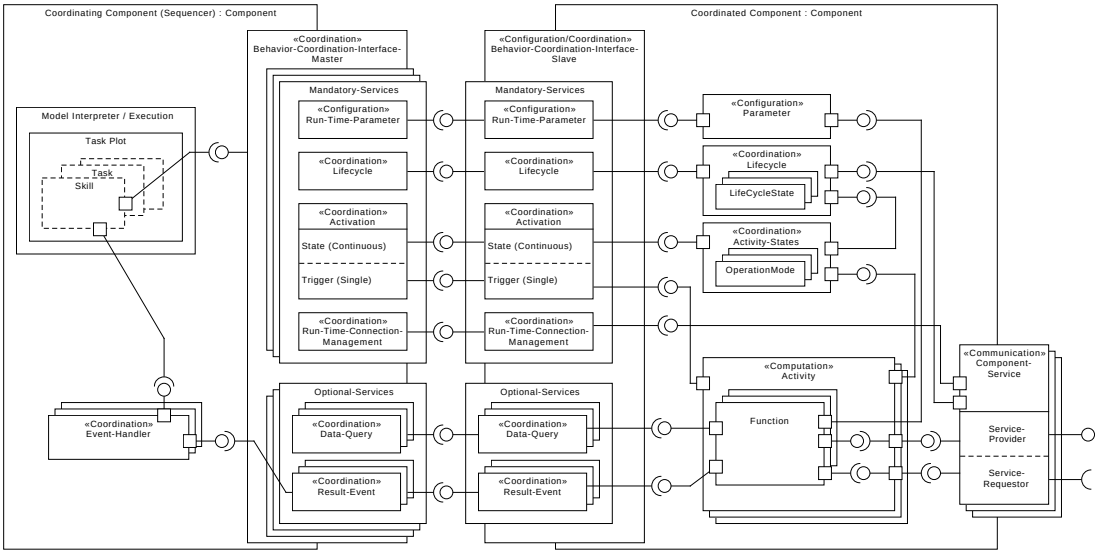

Figure 6.1.: Robotics Component Coordination Interface structures, coordinating and coordinated side at a glance.

Results (Events)
Receiving the results of the activation of the functionalities within the components.

Connection
Coordination of the inter-component connections and thereby configuring the data flow between coordinated components.

Information Query
Requesting and receiving information for coordination from components as well as the usage of expert components (e.g., symbolic planners) for coordination.

Component Life-Cycle
Providing access to components' lifecycle, e.g., shutdown or error states of the components.

The chapter is divided into three parts, the first part starts with a short overview of the approach of the coordination interface, the typing, and the connection to the interface. The second part describes the context the coordination interface is embedded in, its usage, and its connection to software components. The third and last part describes in detail the individual parts and patterns the coordination interface consists of.

6.1. Coordination Service

The primary purpose of the coordination interfaces is to enable coordinating access to the functionalities encapsulated within the software components. The coordination interface thereby harmonizes and lifts the level of access to a more coarse granular service level, preventing fine granular and business logic interwoven component coordination. While the parts the interface consists of (from configuration to component lifecycle) are generic and are used the same for all components, the "data types" communicated and the number of used parts depends on the functionalities and components encapsulating them. In the previous chapters, the composition of different components on the level of coordination modules and their skills is highlighted. The same, however, holds true for the coordination interface as well. Just as skill definitions can be reused to be realized by different components, so can the "data types" of a coordination interface.

With the coordination interface being the same for different components, the skills using the interface can be reused as well. To enable the composition of the coordination interface with the skill realizations, the typing of the coordination interface needs to be explicitly defined and separated from the using and realizing side. The coordination service as explicated entity types a specific instance of a coordination interface. Figure 6.2 shows an overview of the separation of the entities and the involved roles.

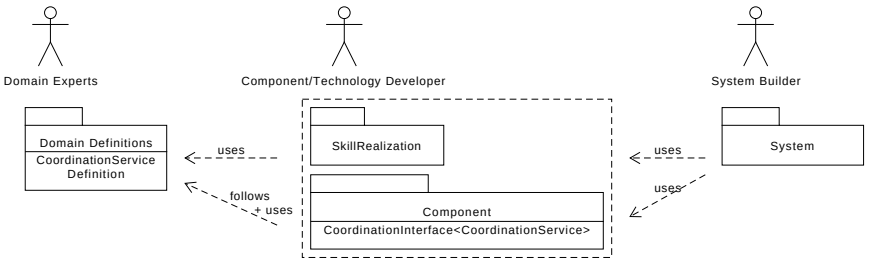

Figure 6.2.: Coordination Interface, overview illustrating the connections among the elements and the associated roles.

6.1.1. Coordination Service Definition

The coordination service definition defines a concrete type of a coordination interface. It is defined within the *ServiceDefinitionModel* by the role of the domain expert. Figure 6.3 shows the meta-model containing the coordination service definition. The RobMoSys "Service-Definition Metamodel" [Sch+18c] concept introduced by Stampfer in [Sta18], is extended by adding the concept of a coordination service definition. An abstract superclass *AbstractServiceDefinition* is introduced to deal with both types of definitions. Both the *CommunicationServiceDefinition* as well as the newly introduced *CoordinationServiceDefinition* class are specializations of the superclass. All definitions are contained by

the class *ServiceDefinitionRepository* grouping logically correlated definitions, e.g., those belonging to navigation. The class *CoordinationServiceDefinition* contains the classes the coordination interface consists of. The *ParameterPattern* for component configuration and *TriggerPattern* for acyclic activation of functionalities within components. All three classes contain references to the related definitions also modeled within the domain models. The *StatePatter* deals with the activation of cyclically activated functionalities within a component. The *EventPattern* class models the events/results from component activations. The class contains references to the data types for the event pattern. The class *DynamicWiringPattern* models the interface for run-time connection management of components, and contains the two functions connect() and disconnect(). The class *LifeCycleCoordinationPattern* links the interface to the components' lifecycle. Finally, the class *CommunicationServiceUsage* adds those communication services which should be used for coordination as well and cover the data-query part of the coordination interface.

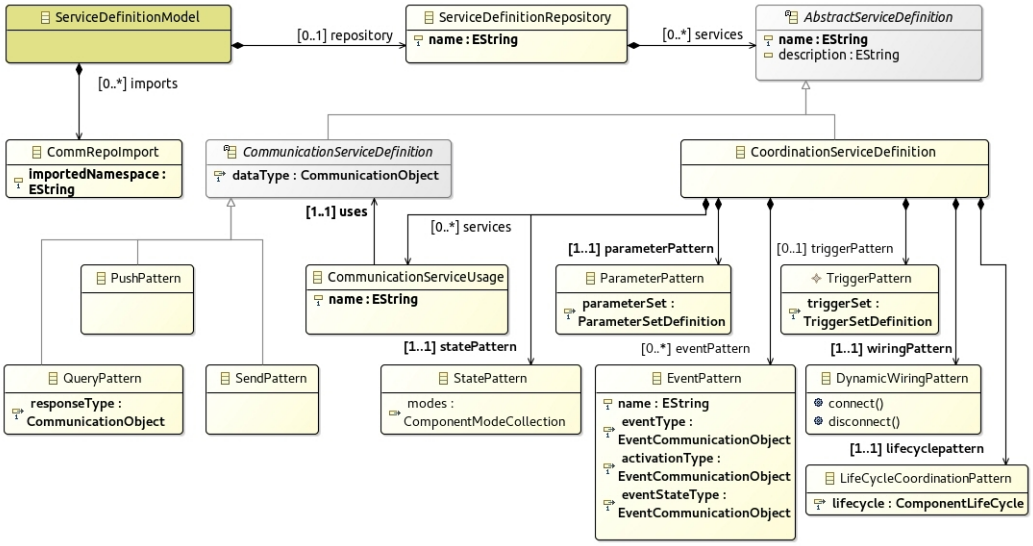

Figure 6.3.: Coordination Service Definition meta-model.

6.1.2. Coordination Service - Component Usage

Following the definitions of the coordination services defined by the domain experts, the technology developer or component developer models the usage of a coordination service as coordination interface within the component definition model. Figure 6.4 shows parts of the component meta-model extended by the coordination ports. The class *CoordinationSlavePort* defining the usage of the coordination interface of a component is contained by the *ComponentDefinition* class and does itself contain a reference to the *CoordinationServiceDefinition* it is following. The class itself contains all further elements abstracted by a superclass *AbstractCoordinationElement*. On the meta-model level, any

parts used to manage the separated roles, and their collaboration are defined. Most of this is achieved by separating the definition, the realization, and the usage of the interface elements. The class *CommunicationServiceUsageRealization* establishes the link between the reused communication service as defined within the coordination service definition, with the realizing service defined in the component definition. Fully automatic mapping is not possible as the component could feature multiple services of the same type. The class *CoordinationMasterPort* defines the usage of the coordination interface master side. This class is an optional element and is only instantiated for the coordinating component. As the master side of the interface needs to deal with any type of coordination service, it is not typed. The class *SkillRealizationRef* defines the link to the coordination module realization. This links the component to the coordination module and the contained skill realizations, thereby forming a package with both the component and the skills it is contributing. The link is required during system composition when composing the components and tasks to a system and application. Without the link, the composition of the tasks with the skills would not be possible as it would be unknown which skills are provided by the components. Implementation wise this link can be realized outside the meta-model as well, e.g., via the containment in a project package. It would, however, also be possible to define the link during system composition if the component developer would not implement the skill realization. Linking the component to the coordination module during the component development (by the role of the component developer technology provider), is very reasonable as the component developer is typically the role also modeling the skill realizations.

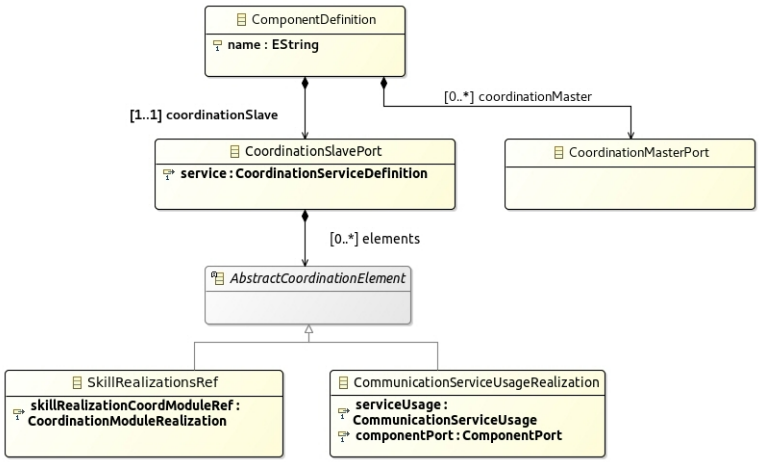

Figure 6.4.: Coordination Service usage meta-model, connecting the coordination interface with its services to the component definition.

Figure 6.5 shows the meta-model excerpt listing the dedicated coordination patterns the coordination interface makes use of. The following sections detail the patterns and their contribution to the coordination interface.

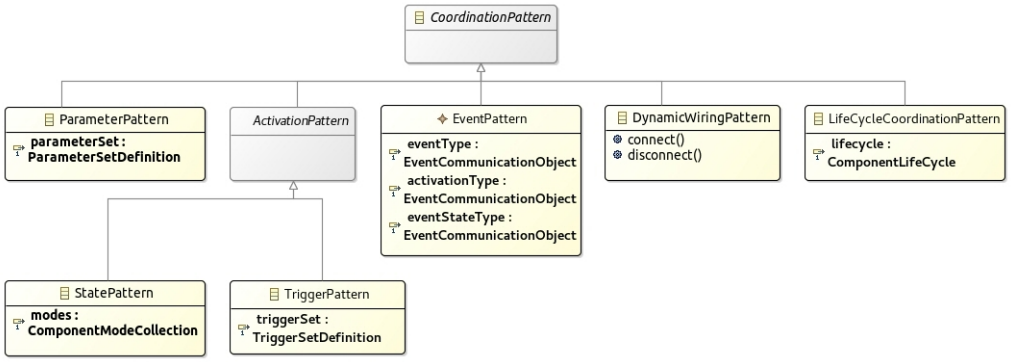

Figure 6.5.: Coordination Pattern meta-model, dedicated pattern the coordination interface makes use of.

6.2. Orchestration Cycle - Usage of the Coordination Interface

The coordination interface is used to coordinate the individual components of a system, thereby orchestrating the overall system. The interaction between the coordinator and a single coordinated component can be described as an *orchestration cycle*. It describes the coordination of a single component and the usage of functionality wrapped within a component behavior-wise. The coordination interface describes in detail how the coordinator accesses a component and the functionality wrapped inside a component. Knowing the six access types, the orchestration cycle introduces further context on their usage, without the need to fully detail the access patterns. The coordination of a component comprises several individual steps, the relation between and the order of those steps is described by the *orchestration cycle* pattern. This cycle is used to realize the skill-level behavior models. Dependent on the technique to realize the skill level models, the individual steps within the cycle can be executed in a single skill only, or can also be performed during an iterative refinement of skills, e.g. if the skill models are realized in a hierarchical approach. A particular case for a not fully executed orchestration cycle is the reconfiguration of a component during the operation of the component. In this case, the activation step is skipped, as the component has been activated beforehand. The decision on how the coordination of a component is performed is defined by the skill level behavior models realizing the orchestration of the components. The concept of the orchestration cycle is sketched by the author in [Lut+14].

Example

A robot equipped with a manipulator and two cameras, one mounted on the manipulator and one on a PTU on top of the robot, is used for pick-and-place tasks. Figure 6.6 illustrates this example. To detect the objects to grasp, the robot features a software component providing the functionality to recognize (identify and localize) objects using vision data. This object recognition component is at the

core of the example, for the sake of simplicity, other components and functionalities required to perform the task are omitted or sketched only. The object recognition component can detect different types of objects, dependent on the configuration of the component. Object types to be recognized need to be configured and known by the component beforehand. The component can be used to detect objects cyclically or in a one-shot mode, running one detection at a time only. The detection result is used for two different purposes, for coordinating the robot's actions on robotics behavior coordination level and by other coordinated components to realizing different functionalities, e.g., manipulation planning. Different kind of information is required for the respective case. For behavior coordination, the coordinator needs to know what kind of objects and how many of them are visible in the current scene, e.g., to grasp a detected object if required for a certain task. Other skill level components need more detailed information about the recognition results. The manipulation planning component, for example, requires the pose of the objects or how the objects are related to each other (standing on a table-top etc.) to perform manipulation planning within real-world scenes. In this example, the object recognition component is used in two different contexts, first using the camera mounted on the PTU to detect the overall scene, and second to recognize an individual object to gain further information using the camera mounted on the manipulator, taking a closer look. Further details regarding the object recognition example can be found in [SLS12].

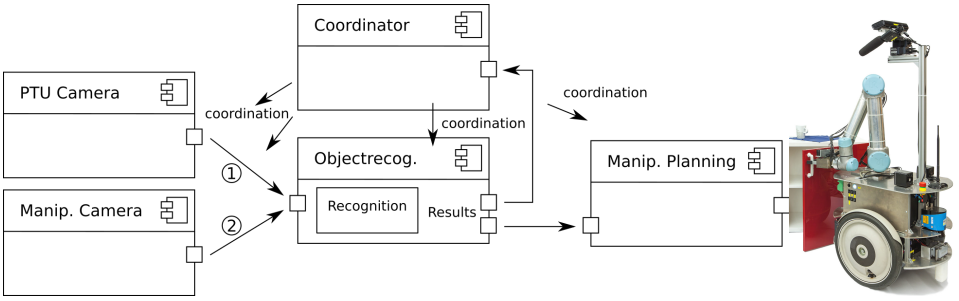

Figure 6.6.: Robotic object recognition coordination example.

Context

The orchestration cycle pattern can be used to enable coordination of software components, as is typically done to coordinate a component in use with other software components in a system. The entities to be coordinated need to be at a component granularity with a clear encapsulation and a limited sphere of influence and need to feature a dedicated coordination interface. The interface needs to enable connection management, configuration, activation, information query, and asynchronous sending of results for coordinated components. The systems to apply this pattern typically follow a 3-tier control architecture, with a reactive

sequencing layer as the orchestrating component. The pattern is, however, not limited to this control architecture. It can be used in any control architecture where the separation of coordinating and coordinated components is given.

Problem

The coordination of robotics software components in a layered architecture requires different steps to be performed. While the proposed coordination interface introduces how to realize the individual steps, such as configuration or activation, the overall usage and the interplay between the individual steps is important and is yet to be defined. The problem addressed by this pattern (*orchestration cycle*) is the sequence of actions and the relations between them required to coordinate a component. The sum of all coordination actions are necessary to realize the skill level behavior models, thereby raising the level of abstraction from functions inside the component to service by the coordination interface and finally making them accessible to the behavior coordination via the skill level models.

Solution

The orchestration cycle connects the individual steps needed to realize the coordination of an individual component in the context of the orchestration of multiple components in the system. Figure 6.7 illustrates the presented approach. The approach is called the *orchestration cycle* as the steps introduced are executed over and over again every time a component is coordinated.

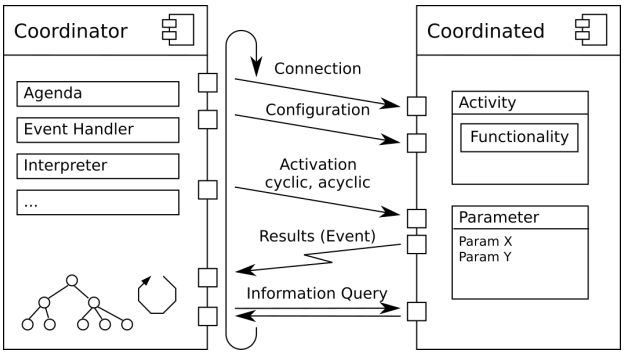

Figure 6.7.: Robotics Component Coordination Interface usage, *orchestration cycle*.

Connection/Dataflow Handling - Step number one is to set up the data flow between the coordinated components. Regardless of the used communication paradigm, connection-oriented or connectionless, the component needs to be set up to receive or request the data required to realize functionalities and to send data generated by the functionalities to or from other components. From where or to which component the data is received or send is defined by the context the component is used within and is modeled on task and skill abstraction level. In the introduced example, the recognition components are set up to use either the

camera mounted on the manipulator or the camera on top of the robot mounted on a PTU as a data source. The decision which to use is taken on behavior level when the component is used to recognize a tabletop scene or to recognize details about an object, e.g., the filling quantity of a package (further details regarding the object recognition example can be found in [SLS12]).

Configuration - Once the data flow is set up correctly, as a second step, the configuration of the component can be performed. It is rational to have the configuration second since the configuration might already use communication to set up the component or to validate the applied configuration. The configuration of the component comprises all necessary information to later execute the activity in a component that is driving the functionality to be realized. In the object recognition example, the component is configured with many different options, the most obvious being the mode the component is running in, either detecting objects in a scene or detecting object properties. This includes, for example, the configuration of which objects to inspect further or in the scene detection mode which object types to recognize at all.

Activation - The third step is the activation of an activity within a component. With the connections, and thereby the data flow of the components input and output ports set, and the configuration of the component set according to the current context and desired behavior, an activity driving the functionality within the component can be started. There are two different use cases for the activation: first, one-shot activities, halting the further operation once performed. Second, cyclically running activities, which need to be deactivated once not needed anymore. In this example, the object recognition component is activated for a one-shot run, whereas a camera component might be activated for a continuous operation to adapt to the current light conditions even if only a single image is used for recognition.
The one-shot activities can further be refined into two groups, those which are of type fire-and-forget and those which provided results to the coordinator. The fire-and-forget actions are used for short-running activities that cannot fail, sometimes related to setting up the component for operation, e.g., clearing a goal buffer or loading the recognition model/data for a new object type. The long-running one-shot activities are thought to run parallel to the coordinator, and the coordinator is asynchronously awaiting the results. The recognition component, for example, is, in many cases, with respect to the reactiveness of the coordination, a long-running activity being activated one-shot.

Results (Events) - Past the activation of a component, the coordinator waits for the results of the activated functionality as the fourth step. The results of an activation send to the coordinator, if any, are communicated asynchronously. The results sent to the sequencer are of reduced and abstract symbolic manner, as the coordinator (at least in a 3-tier architecture) does not deal with the interpretation of data, e.g., sensor data. The information sent to the coordinator is close to symbolic

information as the role of the coordinator is to bridge between subsymbolic and symbolic information, between continuous and event discrete processing and system states.
Not all activations feature results sent to the coordinator, e.g., sensory components such as the camera component typically do not provide results to the sequencer related to the activation. Such components or activities are designed not to produce any results or to produce any errors in normal usage. Occurring errors related to such activities (e.g., hardware failure, etc.) are then best handled by using the component lifecycle, such that the overall component is set into an error state and the coordinator is notified.

Information Query - The last and fifth step is used to receive further information from the coordinated components. With the results of the components' activated functionality available, the coordinating component does, in some cases, require further information about the results, which might not be included in the asynchronously provided information. Therefore, the information query step can be used to receive further detailed information about the results. A common usage of this case is to report identifiers as direct results, e.g., detected objects in the scene and to use those identifiers in the information query to fetch detailed information about the entity referenced using the identifier. Within the example, the object recognition component sends identifiers of the detected objects directly to the coordinator. Those can be used to query further information, e.g., the pose or the reconnection probability of the recognized objects in question.
The information query can also be used to communicate required information directly between other coordinated components. The manipulation planning component, for example, requires detailed information of the object, e.g., the pose and the relation to other objects, which the component can receive directly from the recognition component using identifiers. In this example, the manipulation planning component is provided with the objects of interest (with relation to the current context and task) during the configuration of the manipulation planning component. The identifiers of those objects received by the coordinator as results of the activation are sent to the manipulation planning component. Using those identifiers, the manipulation planning component is able to request the required information. A third use case the information query can be used for, is to receive information from components running continuously and do not require activation, e.g., the numeric position of the robot provided by a mobile robots base server.

Publish Subscribe - Argument The usage of classic publish-subscribe communication within the behavior interface is not desirable. Within the level of the sequencing or behavior execution, the level of information is raised to a symbolic and event-based representation. Most of the publish-subscribe communication is typically used for communication of continuous low-level information such as sensor data. The usage in cooperation with a sequencer is further counter-intuitive

since the sequencer's execution is driven by events, either internal or using events from skill components. The integration of classical publish-subscribe communication services is therefore not very reasonable since the sequencer would have to receive and check all published information to check if any relevant information has been published, which is not the desired behavior of an event-driven mechanism.

Consequences

The *orchestration cycle* relates the individual parts of the component coordination interface by applying the interface parts to realize skill level behavior models. The application of the pattern combines the separated steps and concerns required to coordinate a component in the context of system orchestration. The orchestration cycle enables the separation between the reactive sequencing layer and skill component layer, by guiding how the coordination interface can be used to coordinate the component and how to orchestrate the system. It shows how to provide the individual entities with the required information, while keeping the layers separated, e.g., not sending sub-symbolic information via the sequencing level to another component.

The pattern helps to realize the principle of subsidiarity at the interface between the skill and the sequencing layer. The coordinated components try to achieve their functionalities as long as they can do so according to the assigned decision space. If not possible, they can asynchronously inform the coordinator. This is important to realize the coordinator on the sequencing level capable of coordinating multiple parallel running functionalities. The pattern follows the idea of a clearly defined coordination architecture, with defined responsibilities. The pattern should not be understood as a fixed way where the coordination steps have to be executed accordingly in any case, but more as the typical way of being able to cope with most of the situations that arise during system orchestration. In some cases, e.g., reconfiguration during operation might require a deviation from the described pattern. The relation between the described steps stays, however, unchanged.

Implementation

For the realization of the coordination cycle, the individual parts of the component coordination interface are used. The implementation-specific details about the individual steps can be found at the according parts of the coordination interface.

6.3. Software Component Parts - A minimal Component Model

From a behavior coordination perspective, a robotics software component encapsulates functionalities. These functionalities need to be coordinated to enable flexible and real-world capable robotic systems. This section will illuminate those parts within the software component directly relevant for robotics behavior coordination. Other

parts, such as the middleware connection of the component parts, for example, are omitted. Therefore, this section introduces a minimal component model. The presented approach is further realized using the full-fledged SmartSoft component model, SmartMARS [SSS09; Ser]. The model is presented as an Ecore meta-model but could be defined in other modeling languages as well, as only a small set of modeling elements is used.

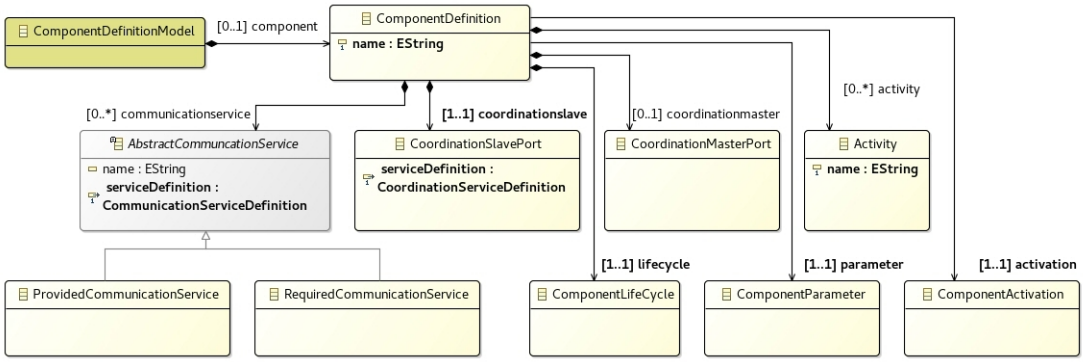

Figure 6.8.: Minimal component definition meta-model.

6.3.1. Coordinated Software Component Parts

The first subsection will illustrate the coordinated software component parts, being the more interesting and important one, as it concerns all skill level coordinated components. The other side, the coordinating component part, typically affects the sequencer component only. Besides the central class defining the component and its name, the following elements are contained in a meta-model:

Activity - <computation>

Activity is the entity driving the functionalities inside a component, it provides computational resources for the functionalities. An *activity* subsumes cyclically/continuously executed functionalities as well as acyclically/one-shot executed ones. The component developer realizes the functionalities within or attached to activities, for example, by reusing existing algorithms encapsulated within libraries. Glue code is required to bind the robotics framework independent algorithms to the robotics framework dependent component (component model).
The component developer needs to be able to access the component parts, e.g., for configuration or component-to-component communication from an *activity*. For coordination of the robotic system, the usage and thereby the activation of the functionalities needs to be accessible coordination-wise.

Component Service - <communication>

Component Services are the parts for component-to-component communication. A

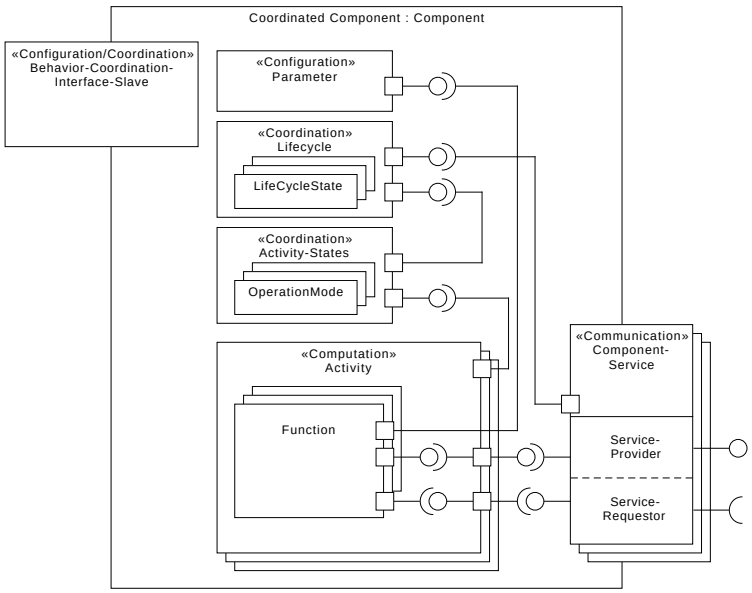

Figure 6.9.: Robotics Software Component Parts relevant for coordination access via the Coordination Interface.

robotic system is a complex cyber-physical system consisting of many functionalities encapsulated in components. In many cases, the algorithms realizing the functionality inside the components require the input of other components, e.g., sensory data, or offer some data to other components in the system. A component is able to feature multiple *component services*, each of them either being a service requestor or a service provider. Decomposing the overall problem into reusable and more manageable parts, namely software components, helps to deal with the overall complexity of the system and is also crucial for a robotics business ecosystem. Establishing communication between the separated parts enables a working system. Component-to-component communication is therefore a major requirement for robotics software components.
From a robotics behavior coordination view and the according control architecture, the connected components form data flow chains. The coordinating component needs to be able to manage the connections and thereby the data flow among the components during run-time. In case of a connection-oriented approach, the coordinating component is configuring which service requestor is connected to which service provider of a component. A good example of the usage is an object recognition component working with sensory data from cameras mounted on different locations on the robot, e.g., pan-tilt unit or the manipulator. The connection of the data source is run-time configurable depending on the context, e.g., recognition of a whole scene vs. recognition of specific object properties.

Lifecycle - <coordination>

The component *lifecycle* is the entity to realize a basic and generic automaton to enable the control of the "state" of the component during run-time. The automaton is used to control the components' overall lifecycle state. This includes two important use cases. First, the component state is configurable from the outside of the component, e.g., the coordinator is able to reset, disable, or shut down a component. Second, for component-induced state changes, the coordinator needs to be informed about, e.g., a fatal error or a crash of the component. As a consequence, all activities within the components are connected to the components' *lifecycle* state, e.g., if a component is shutting down, the activities within the component needs to be stopped in a coordinated manner. The coordinating component needs to be able to connect to the component *lifecycle* and to deal with both mentioned use cases.

Activity States - <coordination>

The *activity states* are used to coordinate cyclic activities within a component. The *activity states* are connected to the generic *lifecycle* automaton, the activity states operate while the component is in an operational lifecycle state. The cyclic activities within a component are bound to the activity states. This enables the control of the activities from a coordinating component. The internally used *activity states* are mapped to the external cyclic activation modes via the activation model. In addition to the pure coordination of the activities, the *activity states* are also used to save resources, e.g., by disabling not required activities and functionalities. The deactivation of a camera component during laser-based navigation, for example.

Parameter - <configuration>

The component *parameter* is the entity for the configuration of the component. The component *parameter* acts a unifying entity for different component configuration use-cases. This covers most important the two typical configuration use-cases for components, start-up, and run-time configuration. As a unified entity, it is used by the component developer to realize the configuration of the functionalities within the software components. From a coordination perspective, the coordinating component needs to be able to access the run-time changeable parameters.

Behavior Coordination Interface - slave

The slave part of the *Behavior Coordination Interface* is the entity wrapping all the services used to coordinate a robotics software component. The interface consists of two parts, a coordinating (master) and a coordinated (slave) part. The parts the *Behavior Coordination Interface* consists of and the use-cases realized by them are explained in detail in the following sections.

6.3.2. Coordinating Software Component Parts

This second subsection illustrates the coordination-relevant parts of a coordinating software component. This component is typically a sequencer component (3-tier architecture) and deals with the orchestration of the robotics software system using the robotics behavior models or a derived or generated version of them. This component follows the same component model as the components it is coordinating; it, therefore, features the same elements as described before. The additional elements described within this section are added on top and are tailored to the task the component is performing. The detailed realization of the coordination component depends on the used coordination approach (e.g., state charts). The following present the minimal set required to make use of the coordination interface.

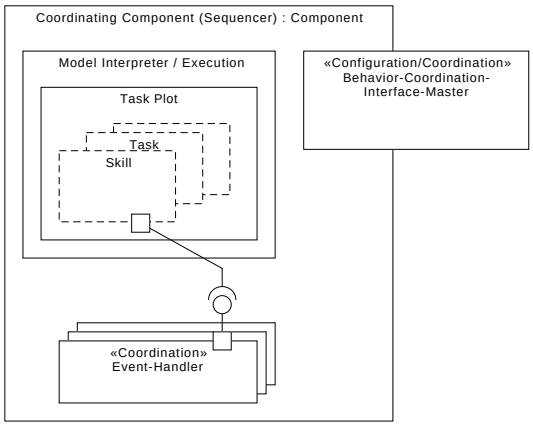

Figure 6.10.: Robotics Software Component Parts of the coordinating component relevant for coordinating software components via the Coordination Interface.

Task Plot - <coordination>
A *Task Plot* consists of task and skill blocks and defines how to achieve a certain goal by ordering the task and skill blocks in a sequence. The skill-level behavior blocks use the master part of the robotics behavior coordination interface to coordinate the individual components. In addition to the master part of the coordination interface, the skill-level behavior models make use of the *Event Handlers* to receive the results of the activated functionalities provided by the coordinated components.

Model Interpreter - <coordination>
The *Model Interpreter* uses the behavior models for system coordination. Dependent on the approach, the way the entity is processing the behavior models may differ: either interpreting the behavior models or executing generated or derived code of the behavior models.

Event Handler - <coordination>

The *Event Handlers* are the entities to provide access to the asynchronously send results from the coordinated components. They are used within the skill-level behavior models. As interface between the skill models and the master part of the coordination interface they could be seen as adjudged to the coordination interface and not the coordinating component parts. In addition to the components' results, the event handlers deal with the lifecycle changes of the component-induced lifecycle state changes, e.g., a coordinated component entering an error state not being able to proceed with operation any longer.

Behavior Coordination Interface - master

The master part of the *Behavior Coordination Interface* is the entity wrapping all the services used to coordinate robotics software components. The interface consists of two parts, a coordinating (master) and a coordinated (slave) part. The parts the *Behavior Coordination Interface* consists of and the use-cases realized by them are explained in detail in the following sections.

Coordination Interface Pattern - Full Approach

The following six sections of this chapter describe the previously sketched parts of the coordination interface in detail. They introduce each part of the coordination interface as a pattern. Each section presents the use-cases the pattern addresses, provides a robotics coordination example, and describes the context the pattern can be applied. The patterns introduce structures and semantics and how they related to the overall business ecosystem vision. Each pattern describes the connection between the roles (horizontal) and software layers (vertical) it is related to. The relation among the individual parts has already been described within the previous section by the orchestration cycle pattern. The Application Programming Interface (API) part of the patterns is described in the appendix to keep this chapter focused on the structures.

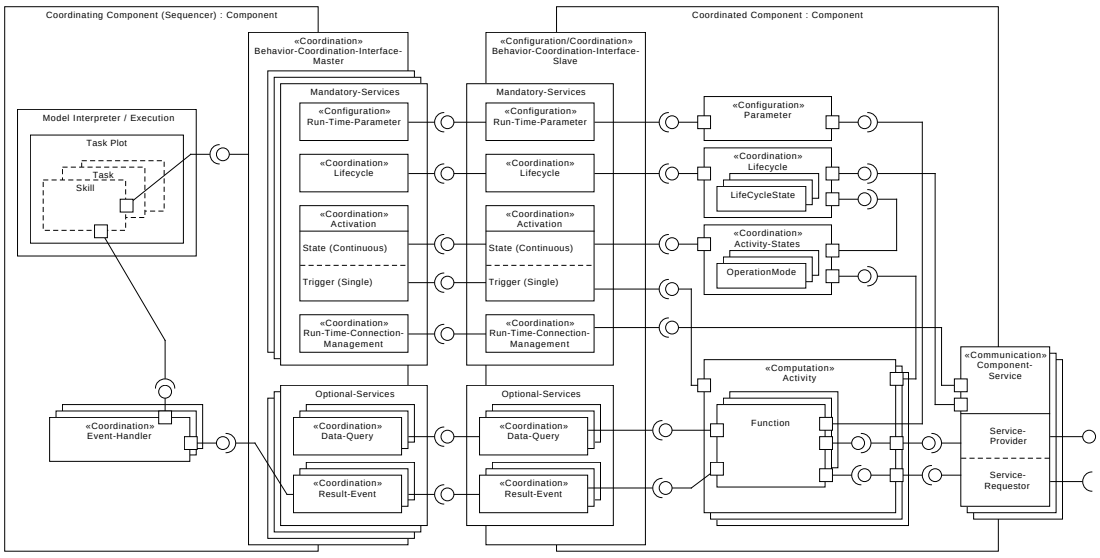

Figure 6.11.: Robotics Component Coordination Interface structures, coordinating and coordinated side at a glance.

6.4. Configuration

Component configuration can be split into two major use cases, dependent on the time and the role performing the configuration. The first use-case is the configuration of the component before or during the start-up of the component. The second use-case is the configuration of the component during run-time. Both use cases improve the reusability of the component in different settings, reuse during system composition and reuse during run-time in different environments and contexts. Without the ability to configure a software component, the reuse of the component would be rather complicated and therefore limited. Reusing closed software components in different settings without the ability for systematic configuration would require changing the component. This would not only destroy the idea of closed software components, as one would need to open the component to understand and change the internals of the component. It would even worse result in many different variants of the same component. In the context of a robotics business ecosystem, this would be a deal-breaker. Systematic configuration of software components is a curial requirement to enable reuse in a robotics business ecosystem and thereby to enable a working ecosystem overall.
Speaking of configuration for reuse in different applications and scenarios, it is sufficient to consider the configuration of the software components as an off-line configuration (static at run-time). The full robotic system could be designed, implemented, and integrated and the configuration of the software components could be fixed before starting the system. To enable the reuse of software components, the configuration during run-time is not strictly necessary.

The second, and for this work especially imported use case for component configuration is to create an interface for run-time configuration of software components. Extending the idea of configuration for component reuse, components need to be configured during run-time to cope with the challenges and the complexity that originates from an open-ended environment a robotic system is working in. The run-time configuration of software components can further improve the robustness of the realized functionality, exploiting the context the component is working in from a coordination side. While for static or simple examples, no run-time configuration is required, for most real-world robotics applications, the configuration of the components to fulfill the desired behavior or task is necessary. In applications where the robotic system has to perform different tasks, react on deviations and errors, etc., coordination and, therefore, component configuration is mandatory. While the consistency considering off-line configuration can be dealt with step by step, consistent component configuration during run-time is more difficult, often also dependent on the environment a system is operating in. If the configuration of individual components depends on the configuration of other components, the consistency of configuration spans across system parts. In many cases, the persistent usage or configuration of hardware components abstracted by their software counterparts needs to be ensured. A robot gripper can not be used to

push buttons while holding a cup of coffee, for example.

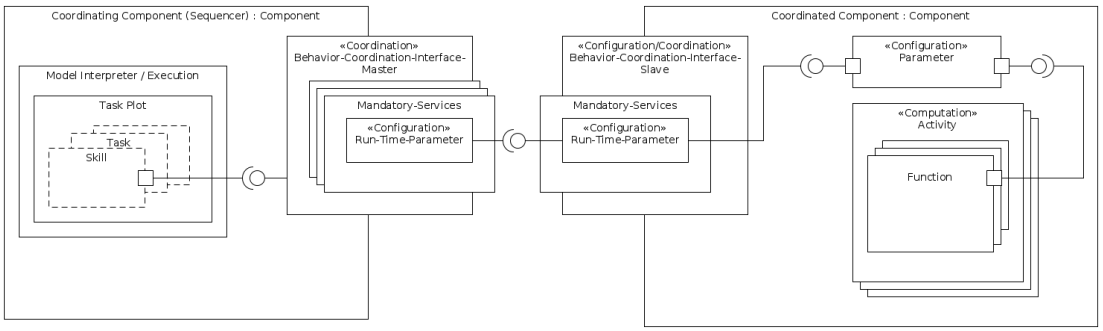

Figure 6.12.: Robotics Component Coordination Interface, configuration of components.

To realize the run-time configuration of software components, the robotics behavior coordination interface uses an *run-time-parameter* pattern. As the name suggests, the pattern is used for the run-time configuration of software components within the context of robotics behavior coordination. The pattern helps to realize a uniform robotics behavior coordination interface, allowing the consistent and flexible configuration of components.

6.4.1. Configuration Pattern Description

Example

A collision avoidance component used together with other components to realize robotic navigation is used in different contexts. It is used to either move the robot in an empty and long hallway or a cluttered and small room. Dependent on where the robot moves, the collision avoidance component needs to be configured with an appropriate velocity window and a smaller or larger robot shape to add some safety clearance. The velocity window and the size of the configured robot shape need to match. A false configuration renders the collision avoidance component none-operational. The coordinating component configures the collision avoidance component with both a new shape and a new velocity window while the component is running and the robot moves.

Context

This pattern can be used to enable run-time configuration of software components, as typically done to coordinate the component in use with other software components in a system. The entities to be configured need to be at a component granularity with a clear encapsulation and a limited sphere of influence of the parameters. The systems to apply this pattern typically follow a 3-tier control architecture, with a reactive sequencing layer as the coordinating component. The

pattern is, however, not limited to this control architecture. It can be used in control architectures with a separation of coordinating and coordinated components.

Problem

Developing a robotics software system using composable software components that encapsulate functionalities per se prohibits fine granular access to the component. Only coarse granular access to the component is granted, e.g., via service-level interfaces. Robotics behavior coordination, however, requires access to certain fine granular configuration options to influence the operation of the component. While some components might work without any (run-time) configuration, e.g., a symbolic planner or a simple sensor component, many robotics components require configuration. Some components won't even work without configuration, e.g., a navigation planner won't work without a goal configuration (given that the goal is set as a configuration option). The (run-time) configuration of a software component also adds to the reusability of a software component. A configurable component is more versatile in use and reuse, which is one of the major goals of an envisioned robotics business ecosystem. To contribute to the ecosystem, the different roles involved need to be addressed, and the handover of the artifacts among the roles needs to be organized.
Run-time configuration is also required to cope with the challenges originating from open-ended environments. The configuration of a component, e.g., by exploiting the context the robot is working in, helps to solve otherwise far more complex problems. An object recognition component used in a kitchen environment might perform better if configured to recognize kitchen-typical objects only. The run-time configuration of the software components further needs to cope with incomplete and transient configurations. The example mentioned above shows a simple case where the configuration of different options over time might lead to invalid component configurations. Those incomplete and transient configuration states should not be visible to the operational functionalities inside the component.

6.4.2. Top Level Objectives

Given the above-stated problem description and the use cases for component configurations, as well as the scope of robotics behavior development in an integrated workflow, the following "top-level" objectives for component configuration can be stated:

The component configuration should be explicated such that changing the values of the configuration does not result in changing the components' implementation. This adds to the reusability of software components.
While a software component encapsulates content in a black-box manner, the configuration for coordination opens the component and explicates otherwise hidden configuration options.

The component configuration should consider start-up and run-time configuration.
The component configuration should support and not hinder the reuse of the components, considering the different roles that might be involved during the development and the run-time of the robot.
The component configuration should enable the management of variability.
The component configuration should ensure valid configurations during design-time and run-time and should therefore enable the validation of configurations.
The run-time configuration needs to consider transient configuration states, ensuring that no incomplete configurations are used during operation.
The configuration interface should be open to being used by different coordination technologies and implementations.
The use of the currently valid configuration within the component should be simple and consistent across the component.

6.4.3. Configuration Pattern Approach

This pattern, in general, opens the closed software components by explicating variation points, in other words, configuration options. The pattern covers two use cases: start-up component configuration and run-time configuration for coordination. Both cases require the explicated definition of configuration options. For the run-time configuration, however, the configuration options need to be accessible to other elements, the behavior skill models, and to a possibly separated role. Both use cases further require the usage of an applied configuration within the software component, usable by the component developer to realize the functionality a component is providing. The description of the configuration pattern is split into two parts, the communication part with data and communication semantics and the modeling part defining and using the configuration options. This subsection will introduce the modeling part first defining those structures that enable the ecosystem vision. This chapter's first part illustrates the separation of roles and the way the roles contribute and collaborate. The second part introduces the communication part of the pattern, using the structures and models defined within the first part. Within the second part, the interface the component developer will use to implement the glue code using the configuration options is presented.

Involved Roles

For software component configuration, three different roles are involved. Figure 6.13 illustrates the roles and their contribution. To enable the separation in usage and the realization of the configuration options, the definition of the parameters is separated from the realization within the component. The role of the domain expert defines the run-time configuration options. Thereby, the role captures and consolidates the domain knowledge about the configuration options of components, e.g., for a mapping component. The separation of the run-time configuration options is necessary to avoid

binding the skill models to a concrete component definition. The start-up parameters are not required to be defined separated from their realization, as they are not used by other roles. The start-up parameters are therefore defined by the component developer when defining the component itself. The role responsible for developing the skills, which could be the component developer or a separated one, makes use of the definition of the configuration options when realizing the skill models. The last role involved is the system integrator, during the system integration step where building blocks are composed to a system. The role uses the definition of both types of parameters (run-time and start-up) and refines their values to match the system's needs. The default map size for a mapping component could, for example, be refined when composing a system operating in private households, different from systems operating in factory shops.

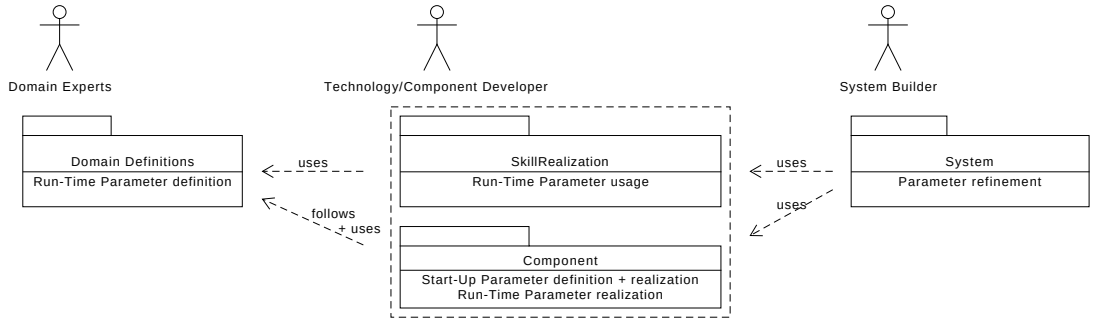

Figure 6.13.: Roles involved in the configuration of software components, for both start-up and run-time configuration.

Vertical Decomposition - Software Layers

The pattern influences the configuration of software components on different layers. While the previous section describes the horizontal relations between the roles, this section describes how and why the pattern contributes to which software layer. Figure 6.14 illustrates the different levels the pattern is contributing to.

Meta-Model Layer At the topmost layer, the parameter meta-models define the structure of the components' run-time parameter itself. This lifts the components' configuration from code to model level, explicating and formalizing the components' configuration. While the meta-models describe the structure, the code generators using the meta-models as well as the instantiated models realize the components' configuration. From a structural and layer perspective, all the aspects of the components' run-time configuration could be modeled in one single meta-model and could be realized in a single instance per component. To enable the reuse of the components' parameter interface and to make component configuration comparable at a model level, the separation of the

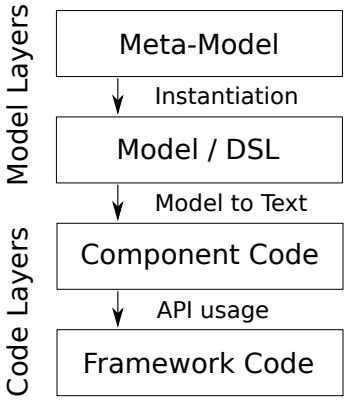

Figure 6.14.: Software layers, the pattern contributes to.

components' parameter definition and the usage is reasonable. Realizing the definition and the usage of the components' run-time parameter also supports other aspects, the context of the development workflow and the principle of separation of roles.

Model Layer The model layer, which instantiates the meta-models, enables the different tools and DSLs for the different roles to work together. The models are split into three parts. First, the run-time parameters are defined in an abstract and component-independent way within a parameter model belonging to the domain models. There, the configuration options with their properties (e.g., data types) are defined. The component configuration, being one of the services a component offers, using an independent parameter model that could be used by another component as well, makes the two components exchangeable at a component service level. Further than making the parameter reusable, the independent definition of the components' parameter further increases the reuse of the skill realizations using the parameters for run-time coordination. Multiple components following the same run-time configuration interface definition enable the reuse of skill models, dependent on the interface definition only, see Chapter 5. Modeling component start-up parameters in an abstract, independent, and reusable way separated from the component is conceivable. However, in most cases not very reasonable. Those parameters are thought, and in the majority of components used, for component implementation-specific parametrization. Neither are these "reusable" by other components, nor are they relevant during run-time configuration for coordination.

The most important part of the layer of the component parameter model is the usage of an abstract and independently defined parameter model. The usage is realized using a model reference. Apart from using a parameter model, the refinement of the defined parameters is possible on this layer. A local extension of the parameter models is driven by two factors. First, as in the above paragraph already mentioned – the local start-up parameters. The start-up parameters are defined at the level of the component model as an extension of the component model. Since those parameters are thought to enhance

the reuse of the component by explicating configurations of the concrete implementation of the components, from code to the model level, the fixed tie to the component by extending the component model is very reasonable.

The second factor that motivates a possible further extension of the abstract and independent definition of the components' parameter is the possibility to further extend the abstract parameter definition locally. The local extension of the component parameters is mainly driven by the idea of implementing specific run-time configuration options. Allowing a local extension of the run-time parameters by the role of the component developer does, however, break the composability of the components with the skill behavior models. If one would bind the skill realizations to the component implementation, which is possible, this huge drawback is resolved as the skill realization could make use of the locally extended interface only. The local extension has been realized and used for several years in earlier versions of the pattern. In favor of better composability and the possibility of separating the skill realization from the component definitions, it is the author's opinion that local advantages are not worth the drawbacks. The presented approach does, therefore, not contain local (component) extendable run-time parameters. This also reduces the complexity on meta-model and code level.

Component Code Layer Following the layers one step further down, the implementation of the component uses the modeled components' configuration. A model-to-text transformation is used to generate the code required by the component developer to implement the component using the configurations, both run-time and start-up. In contrast to the communication objects models, the parameter model is only generated within the components using a specific parameter. The generation of a standalone library that is linked to the executable components is possible but not reasonable. In contrast to the communication objects, all the necessary code to perform the component configuration can be generated from the model and is used only within the parameterized component. The generated code of the communication objects can, however, be extended by the user with its own methods to implement data-bound methods reusable near the data it is applicable to. Generating the code into a standalone library would further increase the complexity visible to the developers.

Framework Code Layer Most of the pattern realization is implemented on a software framework layer accessible via a stable API. The user-accessible part of the pattern within the component is, however, fully generated code, based on the software models, driven by the need to manage the interface between the different roles. Those parts of the pattern not dealing with those management aspects are not necessarily realized using code generation based on software models. The communication mechanism the pattern is defining needs to provide a stable semantics and interface to the using side only. The realization of this can be dependent on the realization technology, e.g., using different communication middlewares. While, in theory, this could also be generated

from models, it is not necessary. The realization of those parts does not affect the boundaries between different roles and, therefore, there is no need to manage this interface in a computer analyzable way, i.e. software models.

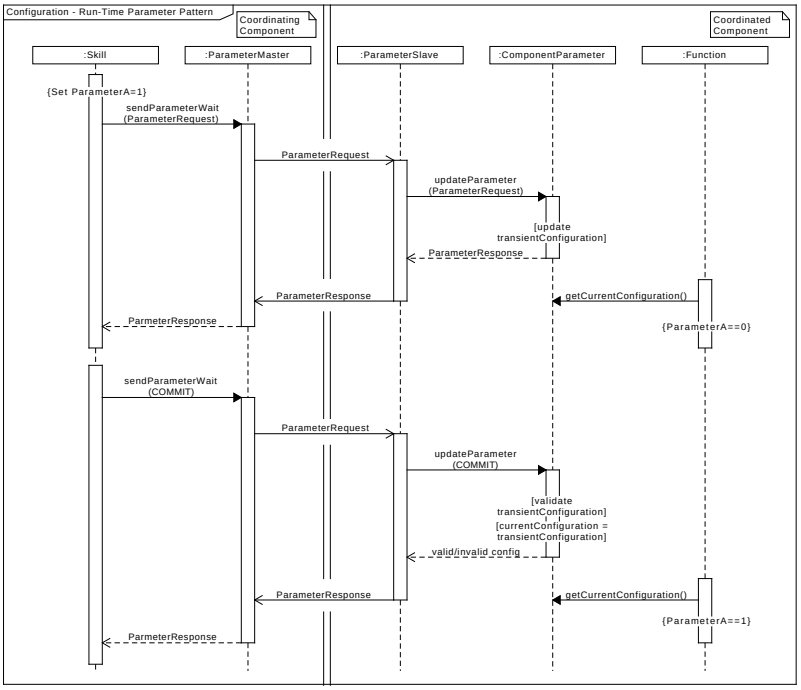

Figure 6.15.: Robotics Component Coordination Interface, run-time configuration of components.

6.4.4. Configuration Structures - Meta-Models

The fundamental structures the pattern introduces are defined using meta-models. The approach introduces three connected meta-models, each focusing on the structures dedicated to the needs of a role and the contribution the role is providing.

The first meta-model *ParameterDefinitionModel* defines the basic structures for the definition of the configuration options, Figure 6.16 shows the meta-model. The domain expert defines the parameters as run-time configuration options independent of the components. A parameter features a name and a list of attributes as key-value pairs. The class *ParameterDefinition* represents the definition of one parameter. It is a named element and can contain multiple attributes. The class *AttributeDefinition* defines one of those attributes, featuring a name and a type. Attributes as configuration options are allowed to be of primitive type only; arrays of the same type are possible. The cardinality of the attributes is stored within the *AttributeDefinition*. Complex type parameter values would add complexity and hinder the realization of the pattern spanning across different

realization technologies as for example, programming languages. The introduction of complex types would increase the risk of misuse of the pattern for other than configuration purposes. The class *AbstractAttributeType* abstracts a simple type system consisting of primitive data types only. The enum data type is important, as for configuration, typically, a limited set or range of values is used. The parameters are grouped logically within sets, the class *ParameterSetDefinition* represents such as a named set. The set introduces a correlation along the contained parameters and further adds a namespace to avoid naming conflicts. Same as for the skill definitions, the parameter sets are contained within domain repositories *ParameterSetRepository*.

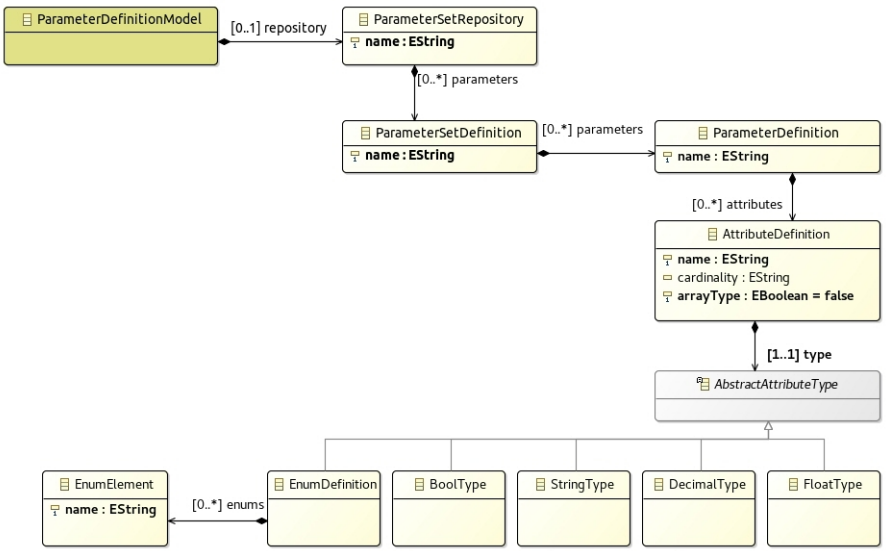

Figure 6.16.: Parameter definition meta-model, the definition of run-time configuration options for coordination.

The second meta-model *ComponentParameterModel* defines the realization of the configuration options within components, Figure 6.17 shows the meta-model. The role of the component developer instantiates the meta-model. The class *ComponentParameter* is the root element of the meta-model and links the component model and the parameter model. Therefore, the class contains a reference to an instance of the *ComponentDefinition* class. To enable a bidirectional link the class *ComponentParametersRef* contains a reference to the *ComponentParameter* and to the component model class *CoordinationSlavePort*. This link is not required structure-wise, it does, however, enable the access to the parameter model from the component model. It simplifies the realization of the DSLs and tools and is therefore contained within this pattern. The *ComponentParameter* class further contains all configuration options of a component. This could either be a reference to a run-time parameter, defined in the parameter definition model, or the definition of start-up parameters itself. The class *InternalParameter* models the start-up parameters not accessible to the coordination and uses the same key-value type system as the parameter

definition model. For the realization or instantiation of the defined parameters the two classes *ParameterSetInstance* and the *ParameterInstance* are used. Both contain a reference to the parameter definition model. When realizing the DSLs and tools, care has to be taken that only parameter instances within the defined parameter set can be instantiated. Both run-time and start-up parameters need to be initialized with a default value. This is required to get a component in an operational state if no explicit coordination or start-up configuration is performed. Those values can be refined by other roles, e.g., during system composition. The class *AbstractValue* subsumes all the type-specific realization of the default values. The types are aligned to those types used within the parameter definition model. The meta-model further contains an abstract class *AbstractComponent-Parameter* which subsumes both types of parameters realized within the component parameter model. The class is used to simplify the handover to the system builder.

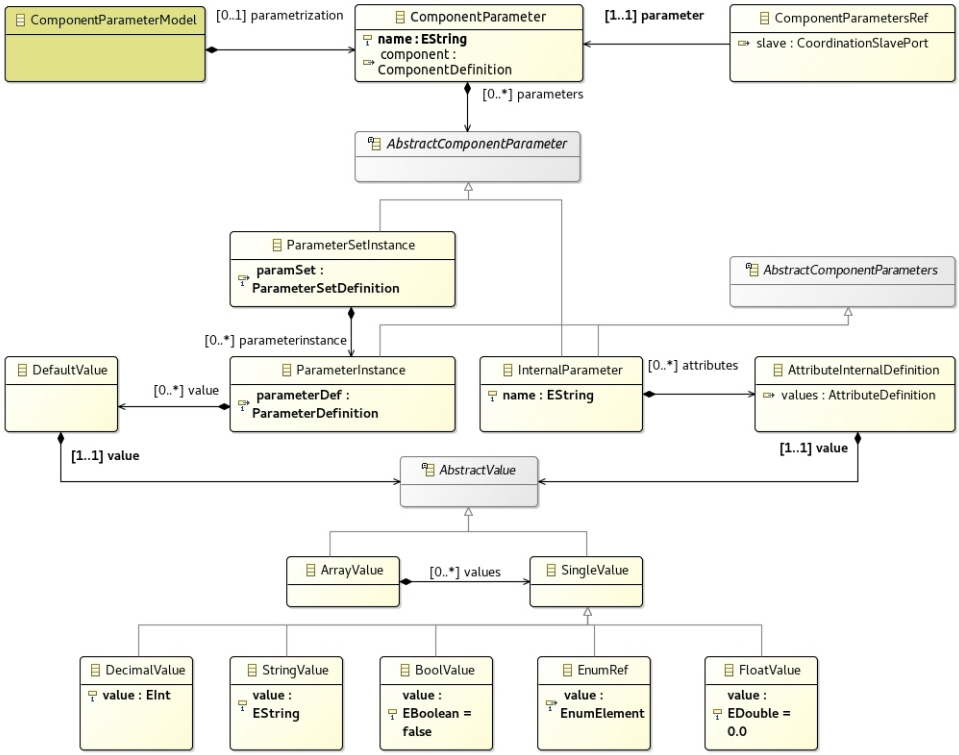

Figure 6.17.: Component parameter meta-model, the realization of run-time configuration options and definition of start-up parameters.

The third meta-model, the *SystemParameterModel* is driven by the system builder and is used when composing a system from building blocks, Figure 6.18 shows the meta-model. The configuration in this step takes configuration options defined by other roles and finally fixes or refines them. The start-up parameters are finally fixed as the next step past the system composition is the start-up of the components. The

run-time parameters are refined, as they can be changed for system coordination. The main class in the model the *ComponentParameterInstance* contains a link to the class of the *ComponentParameter* containing all component parameters. The class further contains a link to the instance of the class *ComponentInstance* representing an instance of a component modeled by the system builder. This link is required to allow for the individual configuration of each component instance. Finally, the class contains the instances of the *ParameterRefinement* class representing the refinement, either iteratively or finally, of the configuration options. The link to the definition of the configuration option within the component parameter model is realized using the abstract class *AbstractComponentParameter*. Each instance of *ParameterRefinement* is linked to exactly one parameter within the component. The class *SystemAttributeValue* models the value refinement itself. The class contains two subclasses, a link to an *AttributeDefinition* as the key element to reference a certain parameter attribute and an instance of the *AbstractValue* class to define the value refinement.

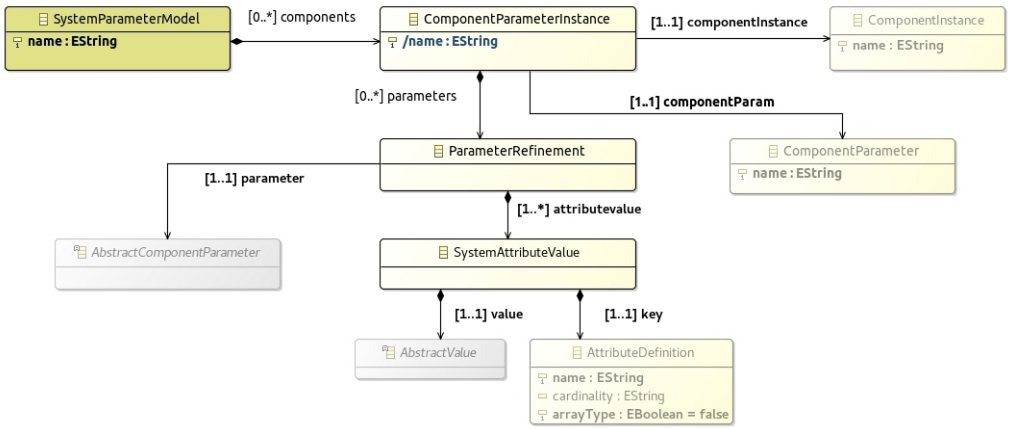

Figure 6.18.: System parameter meta-model, refinement of configuration options.

6.4.5. Configuration Communication

This sections describes the communication part of the parameter pattern, which follows the structures defined within the meta-models. The primary direction of communication is defined by the role of the coordinating component, typically few or one orchestrator (sequencer) and many coordinated components, from one master to N slaves. In addition to the parameter data send from master to slave, a response is sent back from the slave to the master acknowledging the application of the transmitted parameter. Figure 6.19 illustrate this communication semantics. Further details about the pattern realization and the component developer API are described in Chapter A.2.

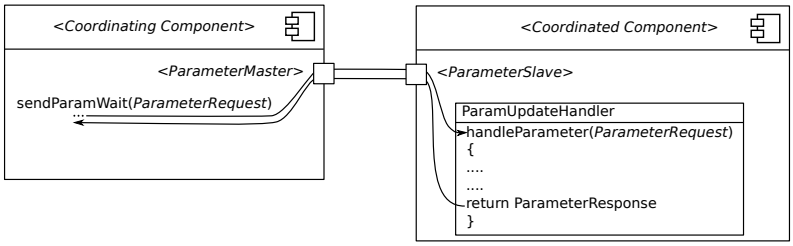

Figure 6.19.: Parameter pattern, communication between master and slave.

6.5. Activation

The *activation* part of the robotics behavior coordination interfaces is used to activate functionalities within software components. The interface part for the activation is split into two major parts and use-cases, the activation of continuously running activities and the activation of single activated or one-shot activities. The one-shot activation can further be separated into two different use cases, short running activities not intended to fail and longer running activities which can fail or produce a result.

One-shot activities are implemented in handler upcalls made available to the user. Their usage can be divided into two typical use-cases: the execution of short activities to prepare or set up the main activity of the component. Deleting a goal in a goal stack or loading a reference image for object recognition from file. This use case is related to the component parametrization. The separation of the parametrization is mainly motivated by the different most important use case, with the parametrization being focused on a set of component parameter variables. With this use case (the short execution time) in mind, the execution can be synchronous to the execution of the sequencer.
The second use case for one-shot component activations is the more obvious activation of long-running (in relation to the execution of the sequencer) activities. The execution of an object recognition procedure or a one-time calculation of a manipulation trajectory are typical example use-cases. The required time that might be necessary to perform such activities forces their execution to be performed asynchronously to the sequencer keeping the sequencer responsive.
The second large group of activities to deal with are those being performed cyclically. Many robotics algorithms are performed cyclically while the robot is running. Collision avoidance or the tracking of persons or objects are typical examples of such use-cases. The cyclic execution and the long time a cycle might take require the cyclic activities to run in parallel to their activation (sequencer).

6.5.1. Activation Pattern Description

Example

A collision avoidance component is used together with other components to realize navigation. The component wraps one cyclic activity, processing sensor, and goal

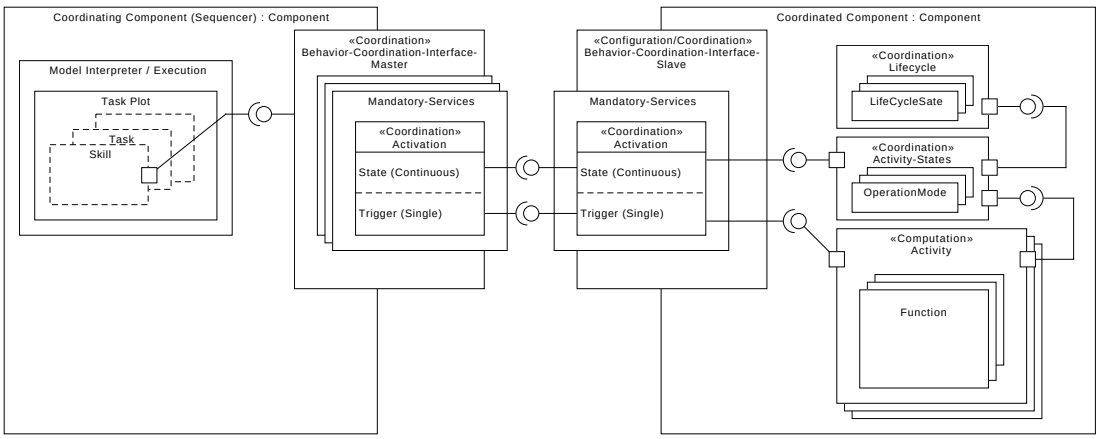

Figure 6.20.: Robotics Component Coordination Interface, activation of functionalities within components.

information as input to calculate safe navigation velocities. Before activating the component, the coordinating component configures and connects the involved components. Once everything is set up correctly, the coordinating component activates the collision avoidance component, which then runs continuously until the robot reaches the goal and the coordinating component deactivates the component. Apart from the coordination purposes, controlling which component is performing what and when, this is also used to save resources. Components not required in the current context (e.g., camera component during laser-based navigation) can be disabled to save resources (computational, energy, etc.).
Another example highlighting the use case of a one-shot activated component is the example of an object recognition component. In contrast to the collision avoidance component, the object recognition component is not executed continuously but one-shot. The activation needs to be different in this case, it is not about activating and deactivating, but about triggering the activity and functionality.

Context

This pattern can be used to enable the activation of software components, as typically done to coordinate the component in use with other software components in a system. The systems to apply this pattern typically follow a 3-tier control architecture, with a reactive sequencing layer as the coordinating component. The pattern is, however, not limited to this control architecture. It can be used in control architectures with a separation of coordinating and coordinated components.

Problem

Developing robotics software systems using software components that encapsulate

functionalities raises the need to activate those in a coordinated way. Robotics behavior coordination requires access to activate those functionalities, both continuous and one-shot performing ones. While components with constantly running component activities are possible for some simple systems, a complex cyber-physical robotics system requires the coordinated activation of component functionalities, also saving precious resources. For some activities, the activation of different functionalities needs to follow a certain sequence defined by the context and the coordination of the system. Closing a gripper on a manipulator typically needs to be done once the arm is in position.
The one-shot activities can further be divided into two typical cases. First, as given in the object recognition example above, activities that take a significant amount of time, and second short running activities or supplementary activities to prepare the execution of the main activities. The distinction between the two cases is required to consider the chosen communication semantics for the pattern (synchronous or asynchronous).

6.5.2. Top Level Objectives

The activation of functionalities within robotic software components should consider the following properties and objectives:

The activation of functionalities should consider both one-shots and continuous activities.
The approach should be able to deal with both short and long-running activities.
The approach should enable a consistent and deterministic activation of functionalities within components. The order of activations send to one component needs to be maintained.
The activation should explicate the nature of the activity (one-shot or continuous) and their execution policy (synchronous or asynchronous).
The approach should support and not hinder the reuse of the components, considering the different involved roles and their separation.
The approach should avoid a fine granular and business logic interwoven activation logic.
The approach must add to the component coordination interface as is used by the skill behavior models to coordinate the components.
The use of the approach should be simple and consistent across the component.

6.5.3. Activation Pattern Approach

The activation pattern structures the activation of functionalities encapsulated within closed (black box) robotic software components. The pattern splits the activations into two groups of functionalities or activities to be started. Cyclic activities are typically

performed over an extended period of time, while acyclic or one-shot activities very often involve only a short period of time.

Involved Roles

The pattern contributes to the coordination interface of a robotics software component. The two roles involved are illustrated in Figure 6.21. To enable the separation in usage and realization of the activation access, the definition of the activations interface is separated from the usage and the realization within the component. The domain experts define the activation options, thereby capturing the consolidated domain knowledge of what and how the activation of the functionality encapsulated by the component within a certain domain should be designed. This holds true for both cases, the activation of one-shot and continuously activated functionalities. The second role involved is the component developer. The role realizes the activation options as defined by the domain experts. The role responsible for developing the skills, which could be the component developer as well but also a separated one, makes use of the definition of the activation options when realizing the skill models. The skill level behavior models use the activation options to activate the functionalities encapsulated and provided by the components. By using the activation options different from the configuration pattern, the role of the system developer is not related to the activation pattern.

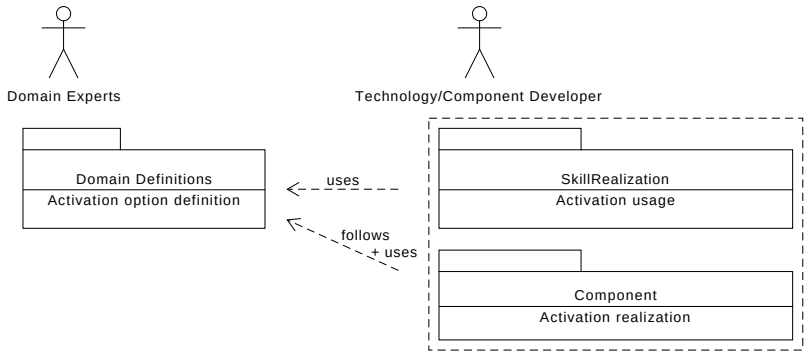

Figure 6.21.: Roles involved with the activation of functionalities within software components.

The acyclic one-shot activation (trigger) of activities uses a handler-based interface. The communication for activation between the coordinator and the coordinated component is synchronous. Depending on the activity's nature, the synchronous call will wait till the end of the activity or just until the activity is queued. The user of the pattern can add an active queue to the pattern, such that longer running activities do not block the coordinator. For short running activities, asynchronous execution with the coordinator waiting till the end of the activation can be used.

Vertical Decomposition - Software Layers

The pattern influences the activation of the functionalities within the components on different layers. While the previous section describes the horizontal relations between the roles, this section describes how and why the pattern contributes to what and on which software level.

The topmost layer, the *Meta-Model Layer* models those structures that enable the separation and the collaboration of the different involved roles. Lifting the level from code to model layer requires the formalization defined on the meta-model layer. On this layer, the separated definition, realization, and the usage of the elements are defined and thereby structured. The instantiation of the meta-model on the *Model Layer* enables the realization of DSLs and tools to support the involved roles. The DSLs and tools enable the usage of the pattern without the need to understand the structures behind the pattern. The realization of the pattern on the code level is separated into three parts. The communication parts of the pattern are realized on *Framework Code Layer*, as it is generic and used the same way in all software components. The component or domain-specific part of the pattern is mapped to the *Component Code Layer*. Within this layer, the user can access the interface of the pattern from the inside of the component hull. The structures required on this layer are generated, following the modeled definitions. The code generation on this layer is reasonable as the content depends on the domain models and is thereby not generic for all the components. The continuous activation option, e.g., "enable video image" within a camera hardware abstraction component is different from the one-shot activation of an object recognition component, e.g., "recognize objects".

6.5.4. Activation Structures - Meta-Models

The fundamental structures the pattern introduces to enable the separation and collaboration of the different involved roles are defined using meta-models. The pattern introduces two connected meta-models, both centered around a specific role. The meta-models are used to realize DSLs and other tooling later on. The DSLs are not presented within this section as they are following the meta-models structures and mainly add none decisive syntax. The DSLs are, however, visible in the experiments Chapter 8.

The first meta-model, the *ActivationDefinitionModel*, defines the structures to express the activation options of components, Figure 6.22 shows the meta-model. The role of the domain expert defines the activation options independent of the component. The options are logically grouped within an *ActivationRepositiory*, e.g. "navigation-activations". The class contains both types of activations. The one-shot acyclic activations are modeled by the class *TriggerDefinition*, contained within a *TiggerSetDefinition* to further group the triggers which are semantically correlated, e.g. "start object recognition" and "capture image of scene". The *TriggerDefinition* is a named element as it is referenced by other roles. The class further contains and defines the execution semantics of the modeled trigger. Two different semantics are possible, *SYNC* as synchronous activation, with

the coordinating component waiting for the activity to be finished, and the *ASYNC* as asynchronous activation, the coordinating component is not waiting for the end of the activity. Explicating the semantics here is important to enable composable skill behavior models, as they rely on the semantics of the interface. Featuring both activation semantics types is not strictly necessary but does, however, drastically reduce the coordination effort required for simple and short running activities. There are many use cases where short running activities are used, ranging from setup activities to short file system actions (saving a map to file), for example.
Each trigger can contain any number of additional attributes to parametrizes the activation. The class *AttributeDefinition* expresses those attributes, as name-value pairs. The data type model-part is omitted, as it follows the configuration data model, see 6.4. The second part of the *ActivationDefinitionModel* are the continuous activations modeled by the *ComponentModeDefinition* class. The class expresses modes used to enable and disable activities within software components. The modes are named elements as they are referenced by other roles. The *ComponentModeDefinition* class is contained by the *ComponentModeSetDefinition* to group correlated modes in the same way as the *TriggerSetDefinition* does.

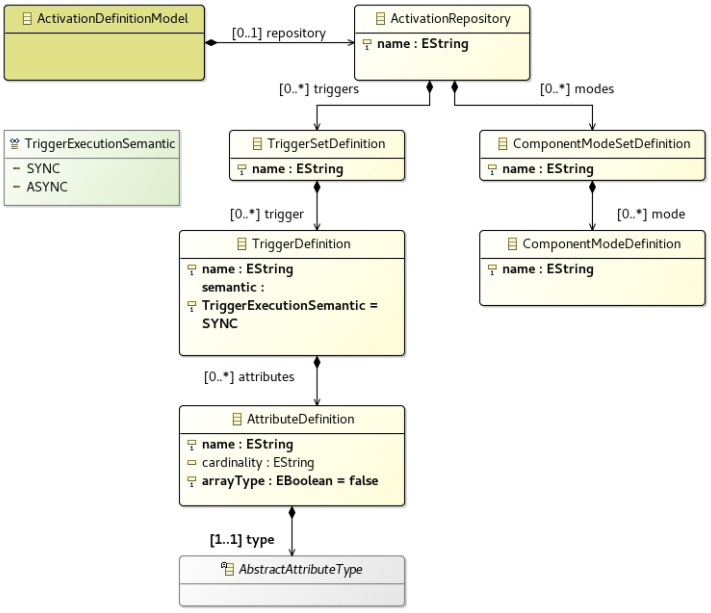

Figure 6.22.: Component activation definition meta-model, defining the structures used by the role of the domain expert to model the activation options of a component.

The second meta-model the *ComponentActivationModel*, explicates the usage and realization of the activation definitions, by the component developer. Figure 6.23 shows the meta-model of the component activation. The central class in the model is the

ComponentActivation, connecting the activation model with the component definition model via the contained reference to the *ComponentDefinition* instance. As well as the *ComponentActivationRef* linking the *CoordinationSlavePort* with the *ComponentActivation* instance for component side access to the activation model. The class contains all further model elements. The classes *TriggerSetInstance* and *TriggerInstance* explicates which triggers are realized within the component, both classes contain a reference to an instance of their definition counterpart. The same holds true for the *ComponentModeInstance* class. The mapping of the components' cyclic activities (typically threads) with the activation is done using the *InternalOperationMode* class. The mapping via the component internally used *InternalOperationModes* to the *ComponentModeInstance* is done via the containment within the *ComponentModelInstance* class.

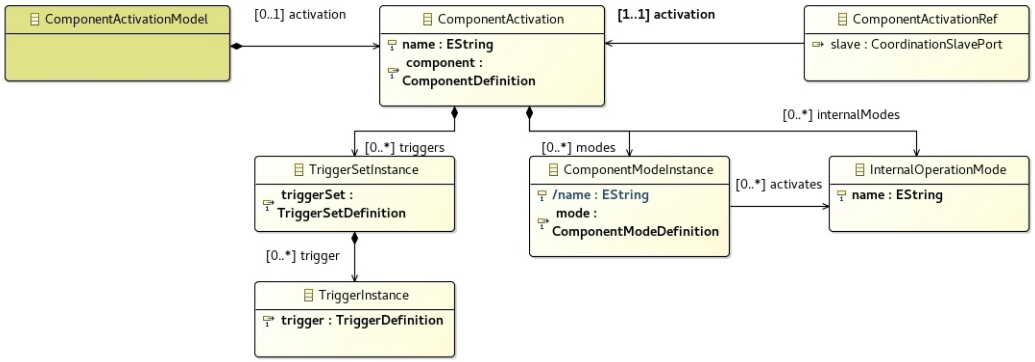

Figure 6.23.: Component activation component meta-model, defining the usage of the activation definitions and linking to the components elements.

6.5.5. Activation Communication - Trigger and State

This section describes the communication semantics and the user view on the usage of the pattern. The described structures are realized on the code layers (component and framework). The realization is split into two parts, the activation of the one-shot and the one of the cyclic activities, due to their different interface on the coordinated component side. Both types feature the same communication direction, sending data from a coordinating component to the coordinated component. Figures 6.24 and 6.25 illustrate the usage and the communication semantics of the pattern. The continuous activation part of the pattern is based on the SmartSoft state pattern. The here presented pattern tailors the state pattern to the coordination use case.

The communication data for the one-shot trigger activation part follows the same type of meta-model as the one used for component run-time configuration and can therefore use the same generic communication data construct. Further details about the pattern realization and the component developer API are described in Chapter A.2.

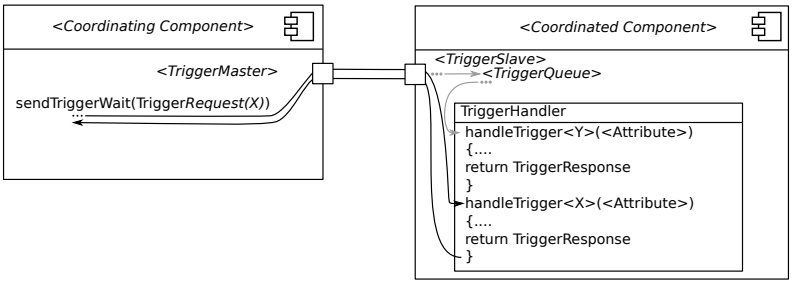

Figure 6.24.: One-Shot Activation, communication between master and slave.

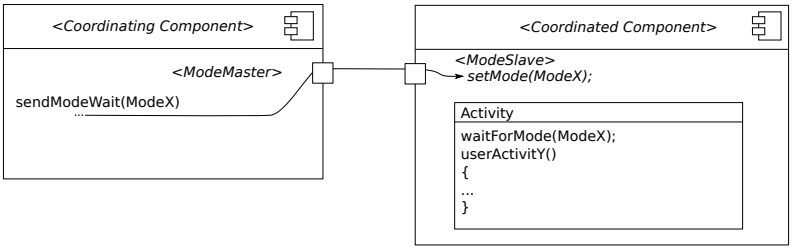

Figure 6.25.: Continuous Activation, communication between master and slave.

6.6. Connection

The run-time connection handling is required to make use of components in different settings in combination with different other components. The connection handling pattern, which is part of the component coordination interface, is rather simple and straightforward. The connection handling part is easiest to use if a connection-oriented component model is used. For other component models not using direct component connections, the data flow between the components has to be managed accordingly. The connection handling part manages the data flow between provided and required component services. This can also be done without direct connections, e.g., via a blackboard pattern mechanism, e.g., using unique topic names. The important part is the possibility to change the data flow of a component from the outside of a component.

6.6.1. Connection Pattern Description

Example

An object recognition component is used to detect objects in a domestic kitchen environment. The component can be used to detect objects on top of a kitchen counter using a robot-mounted camera overseeing the kitchen counter at a glance. The recognition component can further detect the state of previously recognized objects, e.g., the filling degree of containers. Thereto the component requires sensor data showing the inside of a container, which can be recorded using a manipulator mounted camera. To use images from this camera, the data flow

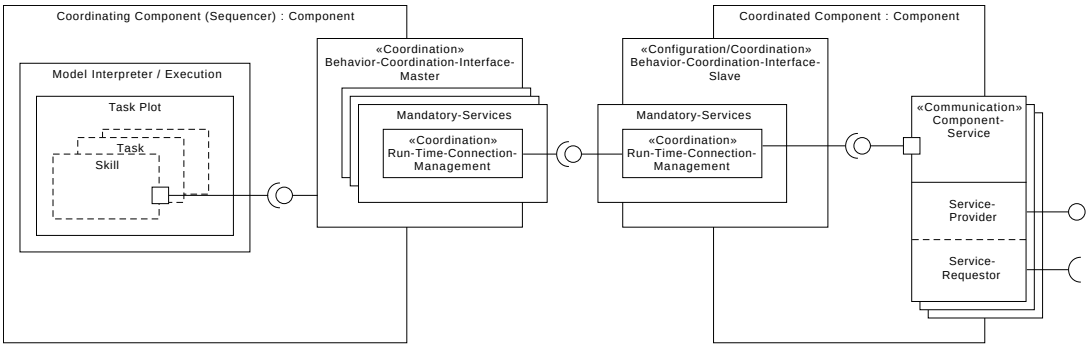

Figure 6.26.: Robotics Component Coordination Interface, run-time connection of components.

between the sensors and the recognition component needs to be reconfigured.

Context

The pattern can be used in the context of software components, to manage the data flow between them. It is used to coordinate software components in a robotic system. The systems to apply this pattern typically follow a 3-tier control architecture, with a reactive sequencing layer as a coordinating component. The pattern is not limited to this control architecture. It can be used in control architectures with a separation of coordinating and coordinated components. The patter needs the ability to run-time manage the components communication interfaces.

Problem

The development of robotics software systems using closed software components operating in open environments requires the adaptation of the system to cope with the challenges the environment or changing tasks poses. In some cases, it is necessary to use software components together with varying other software components. In those cases, the data flow between the components needs to be reconfigured. Therefore, the closed software components need to be able to be commanded from the outside to change the data flow to or from other components.

6.6.2. Top Level Objectives

The following properties and objectives should be considered when managing the connections of components:

The approach needs to be able to connect and disconnect the component services.
The approach must add to the component coordination interface as it is used by the skill behavior models to coordinate the components.
The use of the approach should be simple and consistent across the component.

6.6.3. Connection Pattern Approach

The pattern enables the connection management and thereby the control of the data flow between the components. It adds the possibility to connect and disconnect the required services of a software component from the outside. Following the overall concept of the coordination interface, it is split into a coordinating and a coordinated part.

Involved Roles

For the run-time management of the component interconnections, the only involved role is the role developing the skill level behavior models. During the development of the component itself, no further steps need to be taken. As the role of the component developer is, however, the one modeling the skill level behavior models as well, this role is the only one involved, see Figure 6.27.

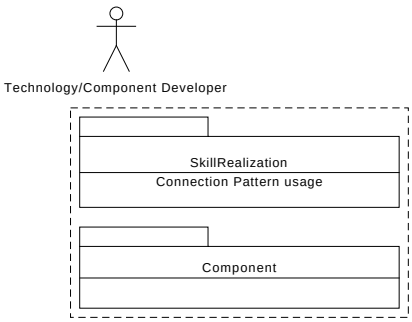

Figure 6.27.: Roles involved with the run-time connection management of software components.

Vertical Decomposition - Software Layers

The connection pattern does not require too many structures. On the meta-model layer, the pattern only introduces the element of the service for the connection handling as an element for the coordinating and the coordinated component itself. As this component coordination service is a mandatory part of the component coordination interface, one could even argue that no meta-model element is required. On the model level, the user does not need to model any element at all. This does not mean that on the model level, no connection management is done; obviously, this is done at the skill level. To enables this, no further model elements are necessary. For the user code on the component level of the coordinated components, no pattern dedicated actions are required. The user should, however, take care that it can cope with connection errors and reconnects. On the framework level, the communication services of the component need to be able to perform connections and disconnections during component operation, past the start-up

phase of the component. The framework further needs to add the slave part (coordinated component) of the connection service to every component and offer the possibility to add the master part to a component. Since no asynchronous response is required, a master/slave and thereby a 1 to n connection type is applied.

6.6.4. Connection Structures - Meta-Models

The pattern adds little to no meta-model elements. This is mainly due to the fact that the pattern does not introduce any interaction between different roles. The pattern only introduces the connection service as part of the coordination service itself.

6.6.5. Connection Communication

The communication semantics of the pattern follows the SmartSoft dynamic wiring pattern as described by Schlegel in [Sch04]. No further structures than those described by Schlegel are required. Further details about the component developer API concerning the pattern are described in Chapter A.2.

6.7. Results (Event)

The *results* part of the robotics behavior coordination interface is used to asynchronously communicate results or messages from the coordinated components back to the coordinating component. Thereby, the pattern realizes the main response channel for coordination in a 3-tier robotics control architecture. The pattern establishes the connection from the sub-symbolic, continuously running coordinated components to the event-driven or event discrete execution of the sequencer, in this context, the coordinating component. Thereby, the pattern describing the results part of the interfaces is named the event coordination pattern. The set of configuration, activation and event pattern form the three most important coordination patterns. They realize the minimum set of coordination interactions for robotics behavior coordination, realized with the proposed coordination interface. The pattern enables the usage of the subsidiarity principle for the coordination of the software components. With the usage of asynchronous results, the coordinating component can delegate the execution of specific functionality to the coordinated component. The coordinated component can pursue the execution of the delegated functionality, providing results to the coordination if necessary, e.g., when finished or if the execution of the functionality has failed.

6.7.1. Event Pattern Description

Example

A set of navigation components is used to realize collision-free navigation of a robot. The components are configured, providing a navigation goal, the context (e.g., the map

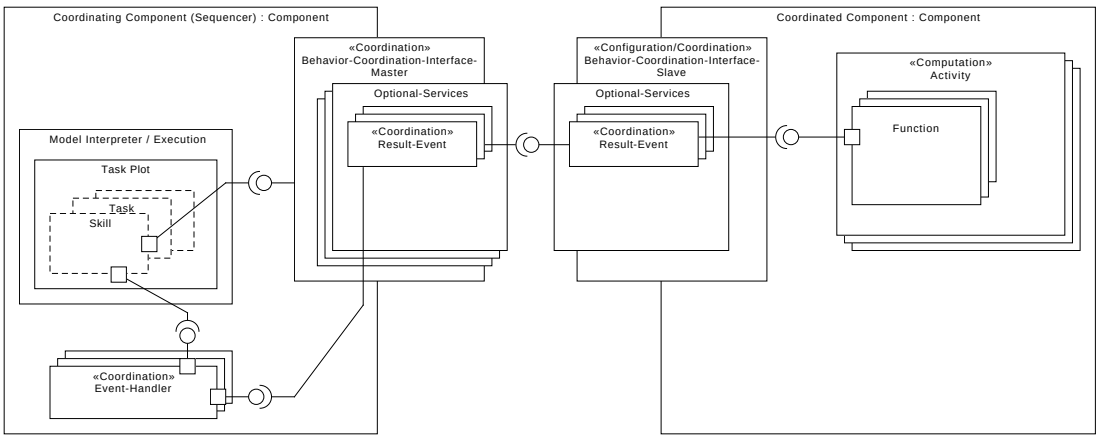

Figure 6.28.: Robotics Component Coordination Interface, results (events) send back from the coordinated components.

to use) as well as the boundaries (e.g., max robot speed) the navigation should work within. By setting the configuration and activating the navigation components, the coordinating component delegates the functionality to pursue (moving the robot to a place) and the boundaries (e.g., max speed) to the coordinated components. All the coordinating component has to do is to await the execution results or to continue the coordination of other components. The coordinating component needs to be notified once the coordinated component leaves the delegated corridor. During the navigation execution the collision avoidance component notifies the coordinating component that it is no longer able to pursue normal execution as the robot is blocked by an obstacle within the safety perimeter of the robot. The coordination component receives this event and activates a recovery behavior freeing the robot. Once the robot reaches the navigation goal, the navigation components send the event signaling the successful execution of the functionality to perform. The coordinating component can now deactivate the involved components and carry on with the execution of the task plot.

Context

The pattern can be used for robotics software coordination to receive asynchronous notifications from coordinated components. The system to apply this pattern typically follows a 3-tier control architecture, with a reactive sequencing layer as the coordinating component. Within this architecture, the pattern describes how to establish the connection between the continuous running sub-symbolic components and an event-driven sequencing layer. The pattern is not limited to this control architecture. It can be used in control architectures with a separation of coordinating and coordinated components.

Problem

The coordination of closed software components requires the asynchronous notification of the coordination from the coordinated components. Software components encapsulate the functionalities realized in the coordination of the robotic systems requires to receive results of the execution of the functionalities. This access needs to be harmonized to allow for uniform coordination of software components, enabling the composition of building blocks. From an architectural perspective, the problem the pattern addresses differs from the other parts of the coordination interface address. With the pattern, no direct coordinating control to components is applied. The active part (significant communication) is further the coordinated component, not the coordinating one, as with the other patterns. The coordination architecture with one coordinating component (e.g., sequencer) coordinating multiple other components raises the problem of how to coordinate multiple functionalities located in different components in parallel. This primarily motivates the need for asynchronous notifications.

6.7.2. Top Level Objectives

Given the above-stated problem description and use case for result communication within the scope of robotics behavior development in an integrated workflow, the following "top-level" objectives for the pattern can be stated:

The pattern should enable the uniform communication of information from the coordinated component to the coordinating.
The possible results send by a coordinated component should be explicated and managed according to the different roles involved.
The communication semantics of the pattern should be asynchronous.
The data communicated needs to be at a high level of abstraction to be used for robotics behavior coordination on skill behavior model level.
The pattern should be usable from the skill level behavior models.
The pattern should support the reuse of the components, considering the different roles that might be involved during the development and the run-time of the robot.
The interface should be open to being used by different coordination technologies and implementations.

6.7.3. Asynchronous Results Approach

The event pattern, in general, formalizes the results of activities sent from the coordinated components to the coordinating component. It thereby explicates the responses used for coordination as events. The description of the pattern is split into two parts, first the communication part, communication semantics, and data. Second the modeling part with the definition and the usage of the results. The modeling part will be introduced first defining those structures that primarily enable the ecosystem vision.

Architectural Considerations

Connection Multiplicity The event pattern used to coordinate software components could be defined and realized as a pattern using a one-to-one connection. For coordination, at each point in time, a defined coordination hierarchy needs to be defined. Thus a single coordinator for a specific component is defined. The pattern can also be used for none coordination interaction among components. Another component can make use of the information provided by the event pattern. This is especially interesting for different robot control architectures or to shortcut some coordination interaction between components. Some coordination actions involving multiple components normally routed via the coordinator can be handled with direct interaction. Within a set of navigation components, a path planner could, for example, make direct use of an event sent from a collision avoidance component signaling a robot blocked event without the need to logically pass the information via the coordinator. The event server needs to be able to deal with multiple activations from the coordinator. Thereby the infrastructure for one to N communication is already present. Considering the different usages the connection multiplicity is best defined with one to N. Thereby, the coordinated component offers the activation of events to multiple destinations.

Communication content The content the pattern communicates is defined by the main use case of the pattern. As the pattern's name (event) indicates, the communicated information is used to bridge the gap between continuous processing and event-driven discrete operation on the skill and task abstraction level. Therefore, the pattern primary transports symbolic state change information, such as e.g., goal reached, robot blocked, objects recognized. While this is the main information, the symbols can be accompanied by further data. This reduces the communication and interaction effort required to coordinate components. In many cases this eliminates additional communication, e.g., information query. The additional information very often consists of identifiers used to query specific information from a component. This is done in two different settings, first to request further information for coordination, and second to request information among the coordinated components. The later case avoids the communication of data required by another coordinated component via the coordinating component. See [Lut+14] for further information and an illustrating example.
An intuitive example for this use case is a mobile manipulation scenario, where manipulation and recognition components need to work together. Once the recognition is finished, the component sends an event to the coordinating component containing identifiers and object types. The coordinating component uses the reported object types and the IDs to coordinate the manipulation components to grasp a specific object. Thereby, the ID of the concrete object is sent to the manipulation planning component, which in turn uses the ID to query the required information from the recognition component directly. In this case, information such as object poses, dimensions, or even point clouds do not pass the coordinating component, reducing overhead and complexity. While

generic communication data expressing a symbolic event might be sufficient to cover the minimal use case, the extension of the pattern to add user-defined data increases the utility of the pattern drastically. The pattern should therefore cover both use cases, containing a symbol as well as additional data.

Communication semantics With the connection multiplicity and content set, the communication semantics of the pattern can be defined. The main purpose of the pattern is to asynchronously send the results of activations back to the coordinating component. The pattern, therefore, defines a publish-subscribe communication semantics. To accomplish the event nature of the coordination component, the pattern should only communicate the changes within the event once. This is in contradiction to a typical publish-subscribe communication semantics where the data is communicated each time. To provide this semantics, the component developer needs to be supported, providing an easy-to-use interface.
The communication semantics required is exactly the one of the SmartSoft event pattern presented by Schlegel in [Sch04]. The user interface of the SmartSoft event pattern is tailored to the needs for component coordination, removing some of the user access methods. This enables the MDE tooling automation and a more easy-to-understand user interface. Communication semantics wise the pattern rests on the SmartSoft event pattern, providing the necessary properties. The pattern realizes a one to n publish-subscribe communication, with the possibility to filter the individual message per activation. It will not lose any data and communicates the messages in order.

Involved Roles

The pattern involves two different roles. Figure 6.29 illustrates the roles and their contribution. To enable the separation of usage and the realization of the events within the coordinated components, the definition of the events is separated. The domain expert defines the events (what is communicated), capturing and consolidating domain knowledge, e.g., the events send from a collision avoidance component used for navigation. The separation of the definition of the events is necessary to not bind the skill level behavior models to specific component definitions. The role developing the skill level behavior models, typically the component developer, uses the event definitions to model the skills, using the coordination service also containing the events. Thereby, the role lifts the level of access from the events being offered as coordination service to skills used by tasks for the tasking of the robot. The component developer is primarily responsible for realizing the functionality provided within the component. The role attaches the functionality within the component to the events following the definitions expressed by the domain experts. Thereby, the roles lift the abstraction level from functional to the service level the events offer.

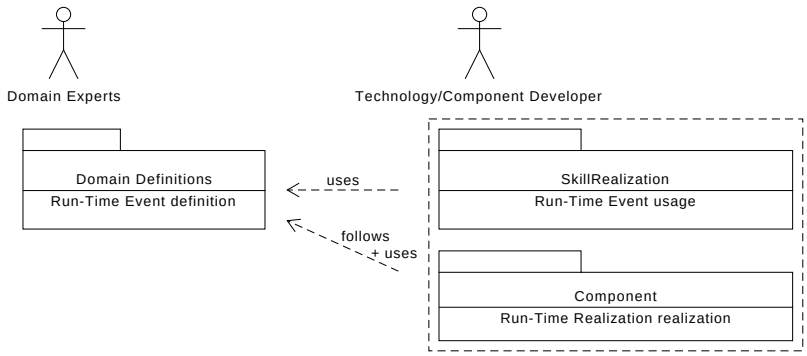

Figure 6.29.: Roles involved with the events send as result from the coordinated software components.

Vertical Decomposition - Software Layers

The pattern influences the development of robotic systems on different software layers. While the previous section describes the horizontal relations between the role, this section describes how and why the pattern contributes what and on which software layer.

Meta-Model Layer At the topmost layer, the event pattern contributes to those structures required for the composition of the elements and the realization of the separation of roles, and the management of the collaboration of the separated roles. The layer is the foundation to lift the results being sent back from the coordinated components from code to model level. The pattern defines the two sides of the interface, master, and slave on the meta-model layer. Thereby, the pattern follows the structure of the coordination interface. The pattern further defines the separated definition, usage, and realization of the results respective the structure (data) of the events. On the coordinating component side, the asynchronous handler interface could be defined on the meta-model level. Since the handlers are implicit and do not contribute to the management of the involved roles, they are not defined on the meta-model level.

Model Layer The model layer follows the meta-model layer and contributes the parts required to manage the separated roles. This is foremost the domain-specific models, capturing the domain knowledge within. Thereby, the services and the data objects, symbols, and accompanying data are defined and named. The user can make use of the tooling provided to realize the models.

Component Code Layer On component code layer, the patter's user interface needs to be accessible. If an MDSD approach is applied, some boilerplate code can be generated

to support the user with the realization. The interface on the coordinated component side consists of a single method only. Therefore no further structures are introduced on the component code layer. For the coordinating component side, the interface consists primarily of the asynchronous handler accompanied by utility methods, e.g., activation. The pattern introduces no further structures on the component code layer.

Framework Code Layer The structures required to provide the user interface and those to realize the communication semantics are realized on the framework code layer. The pattern is accessible to the user in the role of the component developer via the framework API this layer is presenting. The communication mechanism the pattern is defining needs to provide a stable semantics and interface to the using side only. The realization of this can be dependent on the realization technology, e.g., using different communication middlewares. While, in theory, this could also be generated from models, this is not necessary. The realization of those parts does not affect the boundaries between different roles and, therefore, there is no need to manage this interface in a computer analyzable or accessible way, e.g., using software models.

6.7.4. Asynchronous Results Structures - Meta-Models

The fundamental structures the pattern introduces to enable the separation and collaboration of the different involved roles are defined using meta-models. The pattern introduces meta-models used to realize DSLs and other tools. The DSLs are not presented within this section as they are following the meta-models structure and mainly add none decisive syntax. The DSLs are, however, visible in the experiments Chapter 8.

The meta-models define the separation of definition, usage, and realization of the events data types. Figure 6.30 shows an excerpt of the *ServiceDefinition* meta-model including the *CoordinationServiceDefinition*. The structure follows those of the SmartSoft event pattern introduces by Schlegel in [Sch04]. The named instances of the *EventPattern* define the usage within the *CoordinationServiceDefinition*. Each instance contains three communication data types. First, the *activationType* used to subscribe to an event sent from the coordinating component. Second, the *eventType* which defines the actual data send from the coordinated component to the coordinator. Third and last, the *eventStateType* which is used within the coordinated component to decide if a specific event activation should receive an event. Each of the three attributes is of the same type *EventCommunicationObject*. The class contains the attribute *event* of the type *Enumeration* to contain a symbol, which identifies the event. The class *EventCommunicationObject* further contains the attribute *comObj* of the type *CommunicationObject*, representing additional data send alongside the event symbol. The *ServiceDefinitionModel* is used by the role of the domain expert to express a part of the coordination interface, thereby capturing domain knowledge. The event definitions used to coordinate an indoor wheeled-based collision avoidance component can be harmonized abstracting different realizations and implementations.

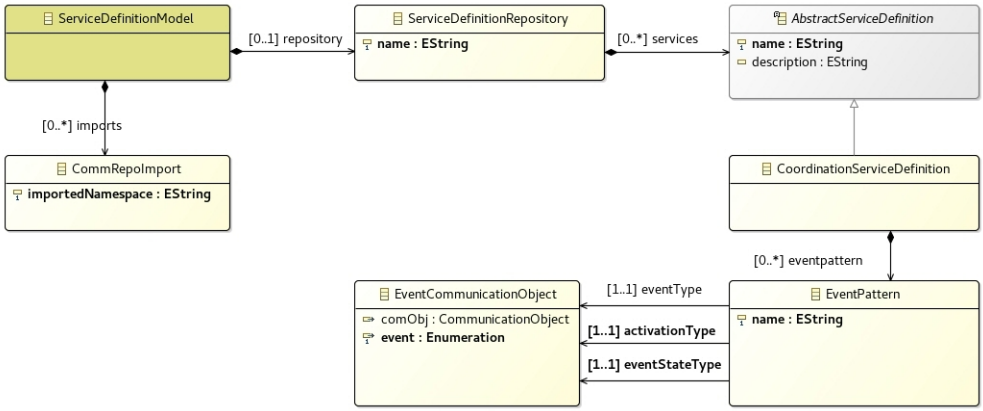

Figure 6.30.: Service definition meta-model, containing the event pattern and the definition of the data communicated used for component coordination.

As for the second role involved the component developer, there are no specific meta-model elements on the coordinated components side. The coordinated component will follow a coordination service definition, thereby realizing all the defined services, including the events. On the coordinating component side, the pattern introduces a callback event handler. This could be done on the meta-model level, but as described in section A.1 the run-time usage of the coordination side does not require the elements to be defined within the meta-model.

6.7.5. Asynchronous Results Communication

The communication semantics of the pattern follows the SmartSoft event pattern as described by Schlegel in [Sch04]. The communication pattern described by Schlegel offers further possibilities to make use of the pattern in a wider spectrum of use cases. The the here presented event pattern tailors the SmartSoft event to the specific use case described, thereby reducing some of the options to choose from. The component developer API and the usage of the pattern are described in Chapter A.2.

6.8. Information Query

The information query pattern belongs to the optional part of the component coordination interface. As the name suggests, the pattern is intended to query information for the usage of system orchestration. Several use cases during coordination require the request of information from coordinated components for system orchestration. In some cases, communication services used for component-to-component communication on skill component level are used; in other cases, new request-response services are modeled and realized.

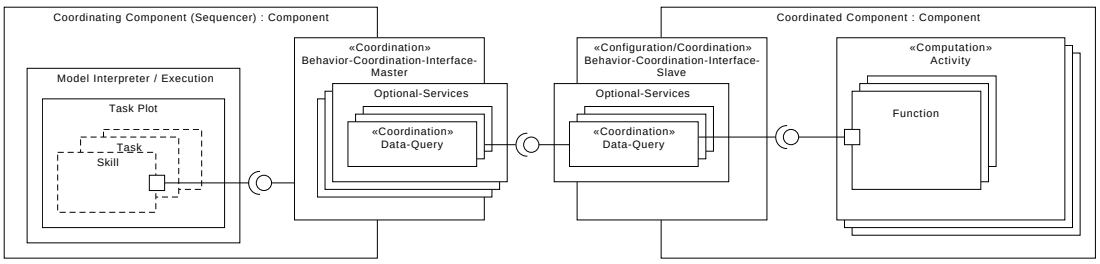

Figure 6.31.: Robotics Component Coordination Interface, information query to the components.

6.8.1. Information Query Pattern Description

Example

An object recognition component is used for the recognition of objects in a domestic environment. The component can detect additional properties of objects, e.g., the filling level. The component is used together with a set of manipulation components to clean up a table. In the scenario, some of the empty objects can be stacked into each other. The coordinating component uses the coordination interface of the recognition component to configure, activate and receive the results of a finished recognition. The results sent from the recognition component to the coordinator do contain the most important information only, such as an identifier and the object type. Other components can request further specific information about the recognized objects, such as the filling level, the location of the object, or the uncertainty of the classification result. While the manipulation component requires the pose to grasp an object, the coordinating component requires the filling level to decide which objects are stackable. Therefore, the coordinating component sends a parametrized request to get the required information, in this example, the filling level.

To efficiently clean up the table, the objects can be stacked into each other. The order and the decision on which objects to stack can be modeled within the behavior models. However, dependent on the number of objects,, this is a huge effort due to the combinatory possibilities. To avoid this, the coordinating component can make use of an external expert component. In this case, the coordinating component uses an external symbolic planner component to plan an efficient order of actions to clean up the table. The planner requires the information to solve the problem. Therefore, the information about the environment of the robot, also containing the filling level of the objects, is aggregated to a world model within the coordinating component and the knowledge base. The planning problem to solve is then sent to the external expert component using a request-response communication. The answer sent to the coordinating component is then used to efficiently clean up the table by performing the planned actions.

Context

The pattern can be applied in the context of the coordination of closed software components. It is used to synchronously request information from coordinated components. The components to request information from should feature a clear encapsulation to limit the sphere of influence of the request. The duration of the request should be limited to a short amount of time concerning the reactivity of the coordinating component. The systems to apply this pattern typically follow a 3-tier control architecture, with a reactive sequencing layer as the coordinating component. The pattern is, however, not limited to this control architecture. It can be used in control architectures with a separation of coordinating and coordinated components.

Problem

The coordination of closed software components requires the requests for specific information provided by coordinated components. Closed software components encapsulate functionalities that provide information for the coordination of software components. The access to this information needs to be harmonized to enable the composition of coordination approach and coordinated component.

6.8.2. Top Level Objectives

Given the above-stated problem description and the scope of robotics behavior development in an integrated workflow the following "top-level" objectives can be stated:

The approach should enable the request for information required for system orchestration.
The approach should support and not hinder the reuse and composition of components, consider the different roles and their separation.
The approach must add to the component coordination interface used by the skill behavior models to coordinate the components.
The usage of the approach within the component should be simple and consistent across the component.

6.8.3. Information Query Pattern Approach

The pattern structures the request for information from coordinated components. It adds the possibility to synchronously request dedicated information from components. Following the overall concept of the coordination approach, the interface is split into a coordinating and coordinated part.

Involved Roles

The information query involves the role of the domain experts as well as the component developer, see Figure 6.32. The domain expert defines the request-response service. This includes the name as well as the data that is used for request as well as for response. The definition captures domain knowledge and contributes to the definition of the coordination service. The component developer realizes the defined request-response service, following the definition of the domain experts. The component developer further models the skill level behavior models using the modeled and realized coordination service definition. The roles thereby use the realized coordination request-response communication to lift the level of abstraction from service to skill level, providing the functionality encapsulated in components to the task level behavior coordination models.

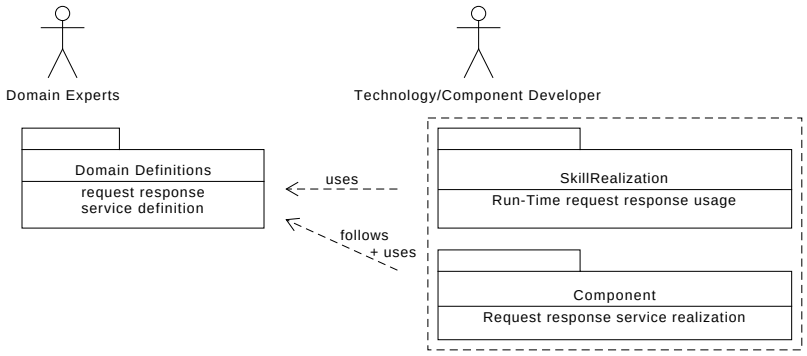

Figure 6.32.: Roles involved in the coordination using the information query to software components.

Vertical Decomposition - Software Layers

The pattern influences the software components on different layers. While the previous section describes the horizontal relations between the roles, this section describes how and why the pattern contributes to what and on which software level.

Meta-Model Layer As the topmost one, the meta-model layer is used foremost to express those structures that manage the interfaces between the different roles. Thereby, the interface is lifted from code to model level, enabling the development of DSLs and tools to work on models. The pattern primarily introduces the structures for the content communicated for coordination.

Model Layer At the model level, the meta-model elements are used and instantiated. Thereby, the request-response service is named and the data types are modeled and assigned. The request-response service is further assigned to the coordination interface.

Component Code Layer On component code layer, the patterns user interface needs to be accessible. If an MDE approach is applied, some boilerplate can be generated to make the user interface tailored to the needs. As the pattern introduces a rather simple handler-based interface, no complicated structures are contributed to the component code layer.

Framework Code Layer Most structures the pattern is introducing on the code level are realized on the framework level. The pattern is accessible to the user in the role of the component developer via the framework API this layer is presenting. The communication mechanism the pattern is defining needs to provide a stable semantics and interface to the using side only. The realization of this can be dependent on the realization technology, e.g., using different communication middlewares. While, in theory, this could also be generated from models, it is not necessary to do so. The realization of those parts does not affect the boundaries between different roles and, therefore, there is no need to manage this interface in a computer analyzable way, e.g., using software models.

6.8.4. Information Query Structures - Meta-Models

The structures and communication semantics of the pattern are used the same for general-purpose component to component communication. Therefore, the pattern could use already existing services without reimplementing the service for the component coordination interface itself. If no existing service can be reused, the request-response service needs to be modeled.

Figure 6.33 shows an excerpt of the *ServiceDefinition* meta-model including the *CoordinationServiceDefition*. To keep the modeling simple and uniform, the meta-model matches both cases the same way. The request-response service definition, also containing the data types (*CommunicationObject*), is done separated from the coordination interface. The coordination service definition is referencing this request-response service definition, done by the class *CommunicationSeviceUsage*. The meta-model of the communication data is represented by the class *CommunicationObject* and contains a data type model not detailed here. The realized meta-model follows the RobMoSys meta-model for communication objects [Sch+18a].

The pattern further extends the *ComponentDefinitionModel*, Figure 6.34 shows an excerpt of the component definition meta-model. The pattern adds the corresponding ports as well as the slave side *RequestHandler*. The mapping of the coordination service realizing the definition used in the coordination service is realized with the class *CommunicationSeviceUsageRealization*. When realizing the tooling, care has to be taken to check for matching service of the referenced *ComponentPort* and the *CommunicationSeviceUsage*.

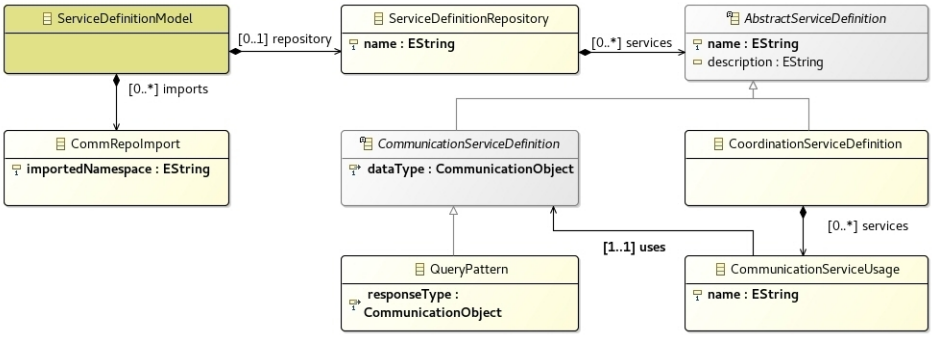

Figure 6.33.: Service definition meta-model, containing the request response pattern used for component coordination.

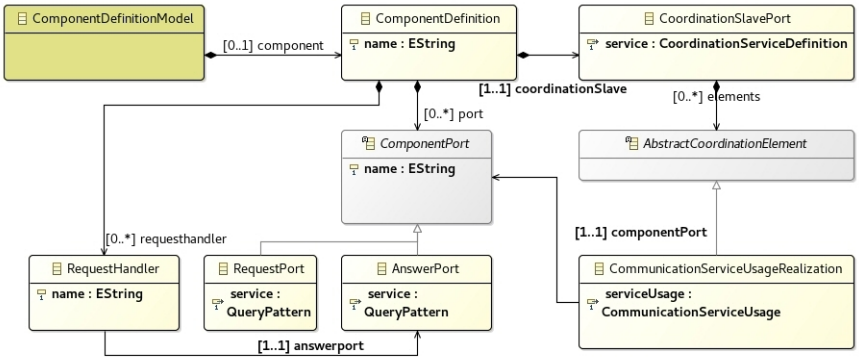

Figure 6.34.: Component definition meta-model, containing the request response pattern parts used for component coordination.

6.8.5. Information Query Communication

The communication semantics of the pattern follows the SmartSoft query pattern as described by Schlegel in [Sch04]. No further structures than those described by Schlegel are required. The component developer API and the usage of the pattern are described in Chapter A.2.

6.9. Component Lifecycle

To support consistent management of the components' resources, the functionalities and activities within a component, and services as the interfaces to other components, every component should feature a generic component lifecycle. This lifecycle is used to coordinate the component internals during the start-up and shutdown of the component, including the management of fatal unrecoverable errors. From a robotics behavior coordination perspective with a strong focus on robustness and error management,

the usage and the connection to the component lifecycle is crucial. The details about the coordinated component side structure of a component lifecycle, not related to the interface, are considered out of scope for this thesis. They depend on the used component model and the implementation of it. More details about the structure of such a component lifecycle and how it can be implemented can, for example, be found in [SLS11]. The following will focus on the behavior model and coordination relevant aspects of the component's lifecycle, namely the connection between the component lifecycle and the behavior models and the coordination and thereby the contribution to the robotics behavior coordination interface. The pattern assumes a minimal component lifecycle to show the different use-case but is not limited to it.

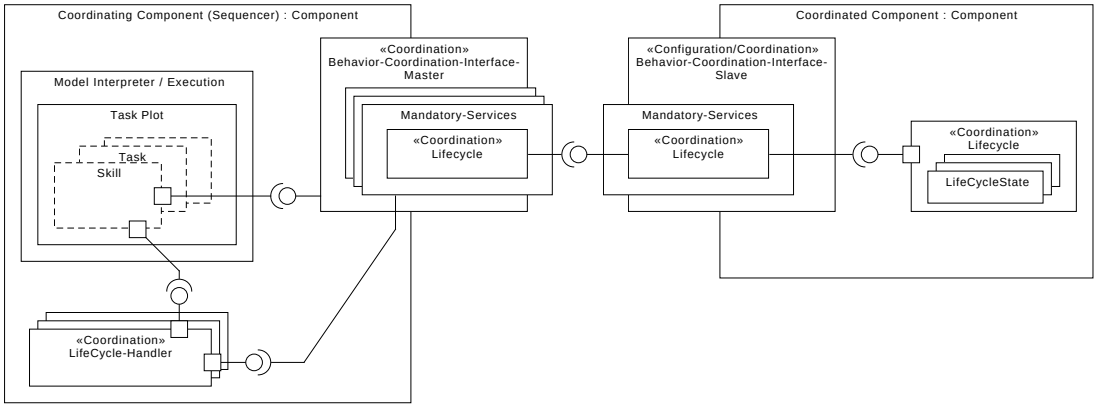

Figure 6.35.: Robotics Component Coordination Interface, lifecycle coordination of the components.

6.9.1. Lifecycle Coordination Pattern Description

Example

In this example, a coordinated software component, a path-planning component, working with other components in a robotic system, switches through its different run-time lifecycle states. The coordinating software component requires access to the component, controlling the lifecycle state, and needs to know in which state the coordinated component is. During run-time, the coordinating component decides to start the path-planning component. Before the system can use the component, for operation and coordination, the coordinator needs to know if and when the component is fully initialized and has entered the matching lifecycle state. The other way around, the coordinator requires to be able to gracefully shut the component down. In the same way, as with the start-up, the coordinator needs to know if and when the component is gone. During the operation of the path-planning component, the component is coordinated to load a broken map file. The component cannot continue normal operation, providing the offered services

and functionalities, and enters an error state. The coordinating component needs to be informed of this state change to be able to react accordingly.

Context

The pattern can be used in the context of the coordination of software components featuring a run-time lifecycle. It is used to coordinate the lifecycle of the components. In contrast to most of the other parts of the coordination interface, the direction of information flow is partially inverted. This is since most of the components' lifecycle changes originate from the coordinated component. The systems to apply this pattern typically follow a 3-tier control architecture, with a reactive sequencing layer as a coordinator. The pattern is not limited to this control architecture. It can be used in control architectures with a separation of coordinating and coordinated components.

Problem

The development of robotic software systems using closed software components featuring a run-time lifecycle requires the connection to the coordination of the system. The connection features two different directions, first the control of the lifecycle from the outside by the coordinating component, and second informing the coordinator of state changes induced within the coordinated software components itself. Robotics behavior coordination needs to be able to perform both actions in a unified way. Only if the coordinator is aware of the components' lifecycle state, e.g., fatal error, the overall system can deal with the situation according to the current context.

6.9.2. Top Level Objectives

Given the above-stated problem description and the scope of robotics behavior development in an integrated workflow the following "top-level" objectives for the components' lifecycle coordination can be stated:

The closed software component needs to explicate a uniform interface to allow coordinating access to the components' internal run-time lifecycle.
The lifecycle of the software components needs to be controllable from the outside of the component.
The robotics behavior coordination approach needs to be informed of component-induced lifecycle state changes.
The approach needs to add to the overall robotics coordination interface.
The approach should be open to being used by different coordination technologies and implementations.
The approach should be open to different components' lifecycle state automatons.

6.9.3. Lifecycle Coordination Pattern Approach

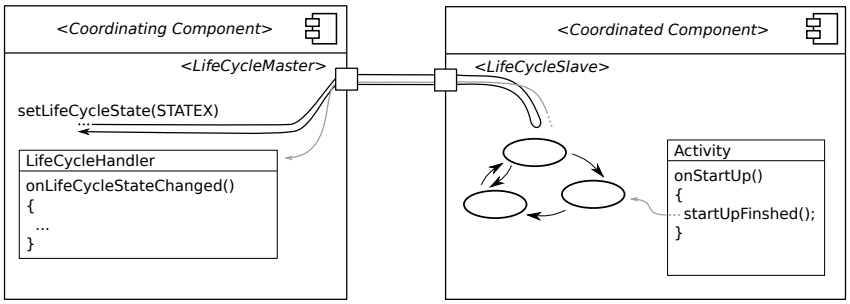

Figure 6.36.: LifeCyle pattern, coordination of the run-time lifecycle of software components.

Before introducing the structure and realization of the pattern, this section introduces a minimum component lifecycle to work with. This is followed by illustrating the pattern's relation to the involved roles and some architectural considerations. In general, the component lifecycle pattern adds to the realization of the fail cognisant principle, where the coordinated component can detect errors, and this information is made accessible to the coordination.

Minimum Component Lifecycle

Each component should feature a harmonized uniform lifecycle. The parts of the component, e.g., the services, should have a defined behavior concerning the life cycle. Otherwise, the composition of components-to-system requires to understand the different implementations or even changes in the implementation. This section introduces a minimum component lifecycle, motivated from a coordination perspective. The presented pattern is not limited to this lifecycle. Other component lifecycle states are conceivable. The semantics of the state and if or how they might interact with the coordination needs to be defined. However, the described minimum covers the most urgent use cases every robotics software component features: Startup, Shutdown, and Error State of the component. The minimum lifecycle is aligned with the RobMoSys component model [Sch+18b], see also Figure 6.37.

The components lifecycle has to have an explicated start-up and initialization procedure. The coordination needs to know if and when a component is fully initialized and ready to be coordinated. Once started, the component switches from “init" to the “alive” state on its own, not controllable from the outside. In most cases, the coordinator will wait for a component to be fully initialized, to reach the “alive” state. An asynchronous notification of the component start-up is conceivable but untypical and rare use case.

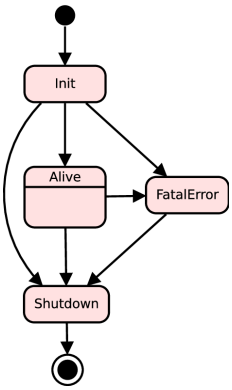

Figure 6.37.: Minimum component run-time lifecycle, figure from [SLS11].

On the other side of the components' lifecycle, the coordinating component needs to be able to shutdown the component in a coordinated manner. The component is not allowed to shut down on its own, the shutdown can only be commanded from outside of the component (coordinating component or a system signal, e.g., SIGINT), in case the component cannot continue operation any further. It can, however, enter an error state. As the coordinator is the only one setting the shutdown state of the component, no notification of the state change is necessary. The successful entering of the state can be achieved through blocking state change. So much for the theory; in practice, things are a bit more complicated. Other external events can trigger the components' shutdown, such as systems signals (e.g., SIGINT). This is done to shutdown components running without coordination which is not intended. To fetch those state changes as well, there is the need to inform the coordination of a state change into the shutdown state, similar to those use cases where the component is responsible for changing the state on its own.

In between, the coordinating component needs to know if the component is running and if and when a component has entered a fatal error state, indicating that the component is not able to operate (e.g., to provide the offered services and functionalities) any further. The component is responsible for entering a fatal error state on its own, not controllable from the coordination side. Based on the information of reaching the fatal error state, the coordination can react to the error, e.g., by restarting the component.

The developer is able to add user-modeled activities attached to the component lifecycle automation. These activities are accessible within the lifecycle state "alive". The component can not activate the user-defined activities on its own. The activation is controllable from the outside only, see Section 6.5 activation.

Involved Roles

The only role involved with the run-time lifecycle coordination is the component developer, see Figure 6.38. The role is involved by making use of the lifecycle from a coordination perspective by realizing the skill level behavior models and realizing the components themselves. As the components' lifecycle is fixed, no other roles, e.g., defining further state, are involved. Thereby, no handover between different roles occurs.

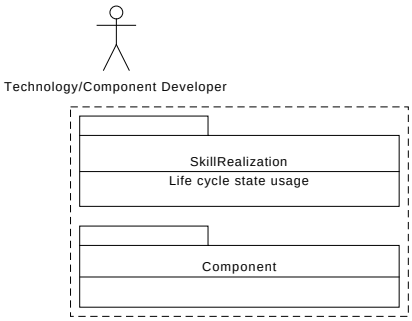

Figure 6.38.: Roles involved in the coordination of the run-time lifecycle of software components.

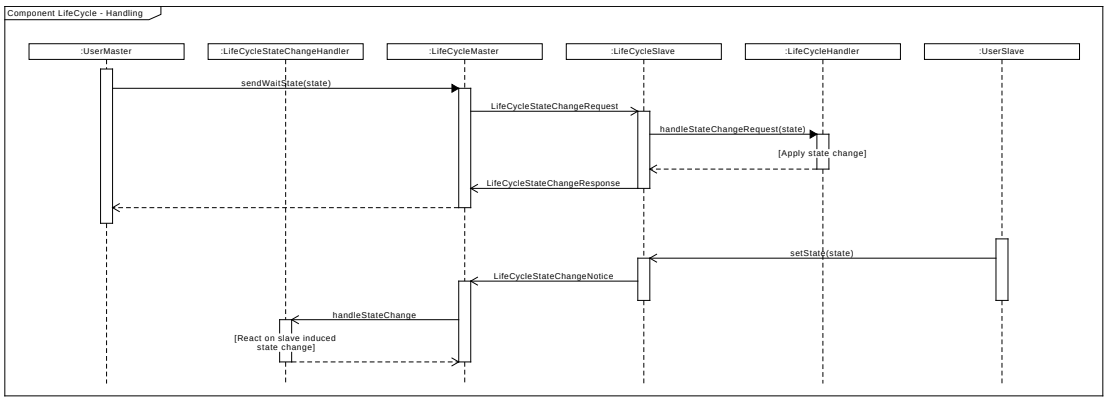

Figure 6.39.: Sequence diagram showing the typical use of the pattern.

Architectural Considerations

To be able to react to changes in the components' lifecycle state, the coordination needs to be informed about the change in the state. The information about the components' lifecycle state change can be transmitted using different mechanisms at different points in time. The most important difference between all approaches that could be taken is the point in time the sequencer gets the information about a component lifecycle state

change and is, therefore, able to use it. The coordinating component could either be informed about the change of the component's lifecycle immediately, or the information could be held back until it naturally interacts with the corresponding component within the coordination cycle.

The decision at which point the sequencer receives the information influences the required structures on the component side and the interface on the sequencer side and the behavior models using the accessible information. If the information is delayed until a natural interaction between both happens, the already existing communication mechanisms could be used to transport the information. An asynchronous and separated notification requires an extension of the communication between the sequencer and the skill components.

Late Notification Holding the information of state changes back until the coordination interacts with the component, as is modeled in the skill level behavior models, follows the principle of locality. The behavior models could incorporate an adequate reaction or policy on how to deal with the component lifecycle changes given the current context, the robot's task, its state and its environment. If, for example, a speech recognition component fails, once it has been activated, the sequencer could restart the component and fix the error by asking the user to repeat a spoken command, for example.

Keeping the lifecycle response close to the "normal" coordination interaction, all coordination patterns need to consider the changes in the components' lifecycle state. For example, the parameters need to be rejected in case of a fatal error state, or the registered event handlers need to be triggered with a corresponding event indicating the changed component lifecycle.

However, this approach of holding back has its drawbacks. The most important one is that non-coordinated components (e.g., in the current context) required by other components could change their lifecycle state (e.g., to a fatal error state) undetected or detected only through errors communicated by other components using this component. This would make the reaction of the coordination to such a change in the components' lifecycle very complicated or break the decoupling of the component, in case components would report errors that originate in other components. Even if one would argue that no component should be without a connection to the coordination, no coordination of such a component could be required in the current context, leading to the same problem.

Another rather drastic downside could be the time delay until the coordination is able to react to a state change. With the information propagation held back until the next natural interaction, it could take a long time for the coordination to be informed. For some applications, it might be helpful, if not crucial, to get the information of such a drastic error as early as possible, e.g., if the robot requires the failed component to finish a job, it might be better to start it, knowing not being able to finish it.

Asynchronous Notification The opposing approach is to forward the components' lifecycle state changes to the coordination directly, separated and asynchronous. The main advantage of this approach is obviously the asynchronous immediate notification of the coordination. This offers the possibility of reacting to errors early and keeping an updated self-model of the components' state on the coordination side. The usage of this information on the coordination side during the execution of the behavior models could be handled in several different ways, dependent on the behavior coordination approach. The coordinator could decide to keep the information till a natural interaction occurs, miming the same interface as above. The architectural decision and the underlying structures influence the possibilities of how the behavior models and approaches can react to lifecycle state changes. Therefore, the following will examine the rough usage of the information in reflection of a possible interface without going into details of the behavior models.

The most simple and direct approach to use the asynchronously provided information is to offer dedicated handlers triggered by components' lifecycle state changes of individual components. This way, the developer can model the system's reaction to certain changes in the components' lifecycle state, independent of the current robot's context. In some cases, this unified reaction is a simple and yet sufficient approach to deal with error, e.g., if a system critical component fails fatally, the only reaction could be to enter a safe error state and to stop the further execution of the system, independent of the current system state. With the components' lifecycle states propagated and handled on the coordination level, a system-wide safe error state could be reached in a safe and coordinated way. As the coordination knows the current system state, including the current active components and configurations, the cancellation of the currents task could be performed using the modeled information in the behavior models. This is fundamentally different from handling such cases in the components locally only. In contrast to the individual components, limited to the given solution space, the coordination is, even with this single and global approach, able to use the task knowledge expressed in the behavior models to react.

However, the obvious limitation of this approach is that the current task context cannot be directly and fully taken into account when dealing with component state changes in the life cycle. In some cases, local and task-dependent handling is more flexible and enables the modeling of local error handling. Despite the limitations of this rather simple approach, the possibility to express a valid global reaction to certain lifecycle changes supports the developers with a simple yet powerful tool.

To deal with the changes in the system lifecycle in the full context of the task, the information of a component lifecycle state change could, on the coordination side, also be used to fetch the orchestration calls from the coordinator to the component, similar to the first approach. This offers the advantage of having both local task-dependent reactions as well as the possibility to react to the state changes in a unified context-free manner. How to deal with this information with the unified handlers and the access to the local task coordination, depends on the chosen behavior coordination approach.

Verdict The drawbacks of the first approach, holding back the information, outweigh the additionally introduced complexity and the new communication mechanism required by the second approach by far. The second approach using asynchronous communication to notify the coordination of a component lifecycle state change offers the required flexibility that enables the developer to deal with components' lifecycle state changes appropriately.

The asynchronous notification of a components' lifecycle state change requires a new communication direction and semantics in addition to the default coordination communication direction, also used to control some of the lifecycle states of the components from the outside. For this control, a simple synchronous request-response communication is sufficient, with the coordinating component being the initiator of the communication. For the asynchronous notification an inverted communication channel is required. The coordinated component needs to send the information of a component-induced lifecycle state change to the coordinating component. The communication in this direction is rather simple and can use plain one-way communication. There is no need to send any response from the coordinated component, the knowledge of a successful transmitted state change is sufficient. No filtering of the communicated messages on the coordinated side is required as the communicated data consists of the state changes initialized by the coordinated component only and is, therefore, very spars and will only be transmitted to the connected master, typically the sequencer. The pattern needs to support an on-activation semantics to transmit the current lifecycle state of the coordinated component on the initial connection.

The pattern could feature different communication semantics. A single master could be used to set the states of several state slaves, which is one master one to N slave. This realization requires the connection to the orchestrated component each time a state is changed, which results in a small communication overhead. The realization of a sequencer orchestrating many components is, in terms of required state masters, more lightweight as only a single one is sufficient to control the states of all components. The asynchronous responses of the components initiated lifecycle state changes, however, requires with this one to N architecture more effort on the state master side. The messages from the slave clients would have to be encoded to correctly associate the messages to the sender on the master side.

From a user perspective, the coordinating component needs to be able to coordinate multiple components and therefore needs to distinguish the asynchronous state changes originating from different components. The one-to-one approach with one state master for one state client would simplify the problem on the pattern side, however, leaving the user to solve part of the problem himself. Overall, therefore, the one to N approach seems to be the more balanced approach, solving the problem for all users being the main argument. With one to N communication architecture, the life cycle state pattern should have one request-response channel for sequencer-initiated state changes and a second one-way communication channel in the reverse direction. The receiver side for the component initiated state changes should feature a handler-based interface

managing the up-calls from the different coordinated components.

Vertical Decomposition - Software Layers

The pattern can be realized on different software layers. While the previous section describes the horizontal relationships between roles, this section describes how and why the pattern contributes what and at what software layer.

Meta-Model Layer At the topmost layer, the meta-models describe the structure of the components' run-time lifecycle management itself. Lifting the components' lifecycle coordination from code to model level explicates and formalizes these parts of the components' run-time coordination. While the meta-model describes the structure, the code generators using the meta-models as well as the instantiated models realize the components' run-time lifecycle coordination. As the components' run-time lifecycle coordination aspects are the same for all components and not extendable by the user, the introduced meta-models elements can be kept at a minimum. No management of the interaction between different roles or workflow aspects needs to be performed.

Model Layer The model layer, which instantiates the meta-models, enables the different tools and DSLs for the different roles to work together. As the component lifecycle is a mandatory part of the coordination interface, the explicit modeling of the element is not necessary. The modeling does not introduce any customization or configuration to the lifecycle, nor does it manage any interface between different roles.

Component Code Layer On component code layer, the patterns user interface needs to be accessible. If an MDSD approach is used, some of the boilerplate code can be generated using the model and meta-model information. Again as the model layer does not introduce further information and the meta-model only contributes unified elements, most of the pattern is realized on the framework software level.

Framework Code Layer Most of the pattern realization is implemented on a software framework layer accessible via a stable API. The communication mechanism the pattern is defining needs to provide a stable semantics and interface to the using side only. The realization of this can be dependent on the realization technology, e.g., using different communication middlewares. While, in theory, this could also be generated from models, this is not necessary. The realization of those parts does not affect the boundaries between different roles and, therefore, there is no need to manage this interface in a computer analyzable way, such as software models.

6.9.4. Lifecycle Coordination Structures - Meta-Models

The pattern is limited to the component developer's role and does not manage the interaction between different roles. The structures the element introduces are limited to an extension of the component model, adding the coordination slave and master ports, and the coordination service definition. As the pattern is mandatory for all components and requires no configuration or refinement, no further meta-model elements are required.

The component lifecycle itself can be modeled using the simple state machine meta-model shown in Figure 6.40.

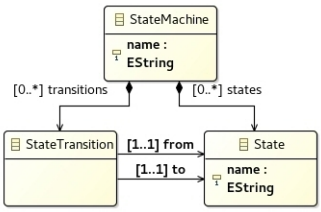

Figure 6.40.: State machine meta-model, to base the component lifecycle on.

6.9.5. Lifecycle Coordination Communication

The communication semantics follows the SmartSoft communication patterns as described by Schlegel in [Sch04]. The component lifecycle pattern combines two communication directions with different communication semantics. The coordination part controlling the coordinated component lifecycle from the outside follows the typical master-slave communication direction and semantics, see Figure 6.36. This matches the communication semantics of the SmartSoft query pattern. The second part of the pattern consists of an asynchronous notification of the master in case of a coordinated component-induced state change. This matches the communication semantics of the SmartSoft push (newest) pattern. The component developer API and the usage of the lifecycle pattern are described in Chapter A.2.

7. Behavior Development in a Robotics Development Workflow

This chapter presents a workflow for the development of robotic systems and applications. The workflow makes use of the proposed approach and structures presented in the previous chapters. The proposed workflow relates the involved roles and their contributions to the ecosystem they are part of. This chapter closes the bracket around the central chapters opened by the corresponding counterpart, chapter 4 (Composable Behavior Coordination in Robotic Systems).

This chapter is not proposing a development process. No single development process fits the many different contexts in which robotic applications are developed. The development of robotic applications within a research lab poses different challenges and requirements than the development of a mass product in industry, or a custom solution developed by an SME. Therefore, the development processes used likely reflect these contexts and requirements.

The ecosystem idea will benefit from different participants, which develop systems or system parts using their own instance of a development process. Introducing a traditional fixed development process is, therefore, neither desirable nor achievable, as also summarized by Bosch [Bos09]. In the light of this, the obvious question arises, why propose a workflow at all.

The presented approach introduces structures to organize the development of parts and systems in a ecosystem, especially the handover of the developed parts between different roles. Thereby, the roles and their contributions relate to each other, featuring dependencies (direct or indirect). For the development of systems and applications in this ecosystem context, those interactions and dependencies among the roles and their contributions need to be organized. The proposed workflow represents one way of achieving that, without dictating the development process to use.

7.1. Influences from the Ecosystem Vision

A distinguished aspect of advanced robotics is to deal with open ended environments. That comes with additional complexity as the system as a whole needs to be coordinated to act and to react to changes in the environment to fulfill its task or to be able to perform different tasks. Not all of the eventualities that might occur during run-time can be foreseen and dealt with explicitly during design-time, as this would result in a combinatorial explosion.

The coordination of the components at run-time is thus required to realize a task or a goal the overall robotic system has to achieve. The presented approach enables the combined development of functionalities e.g. wrapped in software components and robotics behaviors, while keeping independent roles separated.

Robotics with its many cross cutting topics, needs to use the expertise of many different experts in different roles. To generate a robotics software business ecosystem the individual roles in the ecosystem need to be able to collaborate. The individual developed software parts need to be composable to a working system. The separated development of composable software parts requires guiding structures.

MDSD as technology enables the development of tools that make the structures necessary for composition and separation of roles usable with little effort. Models as computer processable central elements enable the consistent handover of artifacts between the different roles.

Given the overall motivation and background, the following sections will introduce this proposed workflow. It will sketch the workflow in an overview, followed by a detailed description of the involved roles and their relation to the workflow, and workflow phases.

7.2. Workflow overview

The following presents the fundamental roles and steps which are involved in robotic application development and how they are connected or related to each other. Figure 7.1 illustrates an overview of the proposed workflow, it shows a linear execution of the workflow, there is, however, no need to execute the phases linearly. Each building block, component, skill or task, can be developed in separation, the order of composition is therefore not fixed. The order and execution of the steps or the roles involved may depend on the pursued development process, the application to develop, or the environment (company, university, community group) where the development takes place. In many cases, the way how robotic applications are developed involves a tailored process, or is even done somewhat improvised. This typically involves an iterative approach of the steps, which is not in conflict with the here presented workflow.

In contrast to the composition of components to robotic systems, the development of applications includes the behavior (tasking) of the robots as well. This shifts the perspective of the development. The composition of the components can be achieved by focusing on connecting services of components as first class elements, as is proposed by Stampfer [Sta18]. With the application and the tasking of the robot in focus, the functionalities required to realize the applications are the most important elements. Those functionalities are provided by the components and their skill realizations. The services used for inter-component communication steps into the second row. The development workflow is therefore centered around the behavior development for the application.

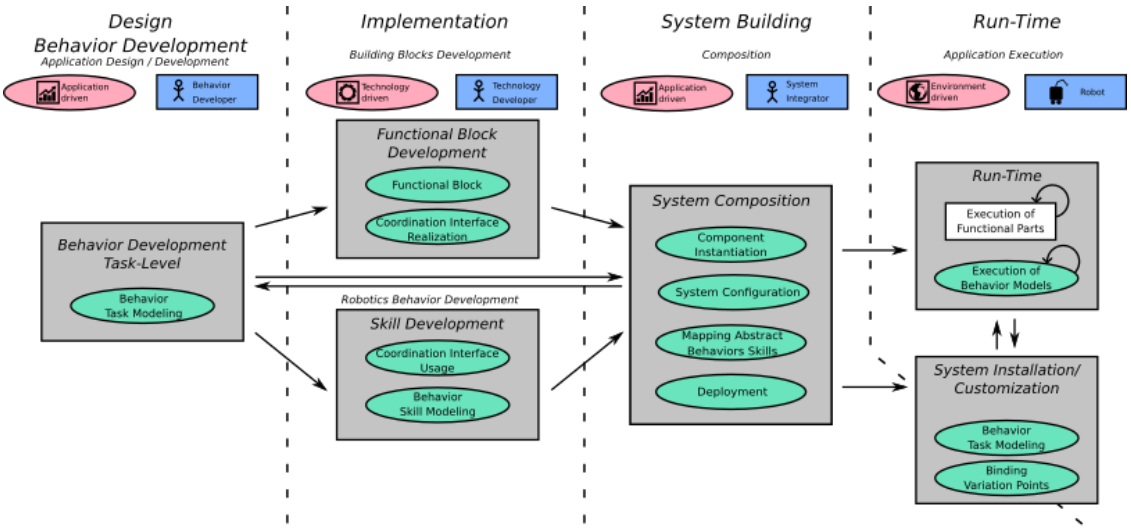

Figure 7.1.: Overview over the robotics development workflow, from design over implementation and integration to runtime.

The development workflow of a robotic application can roughly be grouped into four phases containing different steps:
The **Design Phase** focuses on the abstract design of the application. This phase is not concerned with how the application is realized, it strictly focuses on the high abstraction building blocks and their interfaces. The phase utilizes robotics behavior tasks and skills.
The **Implementation Phase** focuses on the implementation of the individual functional building blocks, as well as the skill realizations, both not being bound to the application to develop.
During the **System Building Phase** the design of the system as the result of the design phase is realized by choosing the functional building blocks developed during the implementation phase. The functional building blocks together with the skills will be composed to the developed robotics behavior such that they fit the requirements of the application.
At the end of the proposed robotics development workflow stands the **Run-Time Phase**. During this phase, all the results, the developed models and realizations are executed and left-open variability is closed.

During the different phases, feedback towards the system design is possible and reasonable. Especially during the system building phase, where all the building blocks are composed together, the need for changes that could be applied iteratively to the system and behavior is quite common.

The presented workflow realizes a top-down development, from high abstraction task models, via skills to the realizations of the functionalities within components, c.f. Figure

7.2. A bottom-up development, starting with the functional build blocks (components), is also possible with the presented approach and the structures. The other way around, driven by the application, is however more convenient, at least when starting from scratch. The required functionality can be searched via matching skills provided by the components within the ecosystem. With an existing H/W system, at least some parts of the software system are forethought, due to the H/W matching components, bottom-up might sometimes be more reasonable.

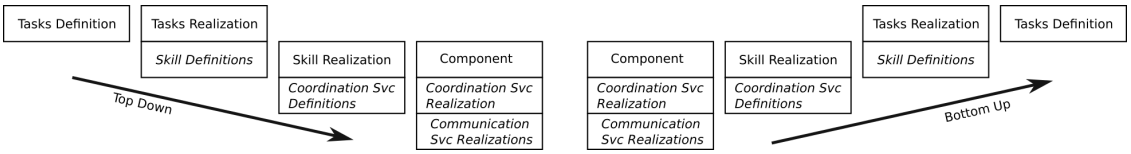

Figure 7.2.: Robotics behavior development, top-down or bottom-up development.

The overview presented so far does not show the ecosystem interaction. Each role can, however, interact with the ecosystem, contributing to and benefiting from it. The following sections will introduce the roles in detail and discuss the ecosystem interaction, respective the interaction among the roles.

7.3. Roles in a Robotic Development Workflow

The workflow can be explained from at least two sides, stepwise and roles wise. The next section will describe the workflow following the steps. Before that, the roles involved are described below.

Behavior Developer

The *Behavior Developer* is responsible for defining the coarse architecture of the application and is iteratively developing the robotics behavior (models on task level). Given the idea of the application to realize and the functionality that is required to realize the application, the role partitions the application and defines the coarse tasks. Top down the role iteratively refines the realization of the behavior following the needs of the application to build. This includes also the usage of the skills required by the application. The skill definitions used are either reused from the ecosystem or defined by the role itself.

The *Behavior Developer* needs to have good procedural knowledge about the application to build as the role is designing how the system acts and reacts within the environment and given the service it should provide. Using the functionalities provided by the technology experts, which are represented on skill level, the task expert composes tasks to fulfill the needs of the application in a specific domain.

The role of the *Behavior Developer* develops system parts on the abstraction levels of TASKs with the expertise to know how the applications need to fit the domain's

specific requirements. The tasks capture the application logic on a high abstraction level.

Technology Developer

The *Technology Developer* is responsible for the development of the components, containing the functionality the system is composed of. The software components are required to enable the usage of the functionality in a software business ecosystem. The *Technology Developer* uses the interface definitions provided by the domain experts via the ecosystem, to realize the components according to them, for communication and coordination. The *Technology Developer* has a white box view of the component. The role designs and implements all internals of the components such as threading, handlers or which algorithms or libraries to use, to realize the functionality. In addition to the components the technology developer also develops the skills associated to the component. The role thereby raises the level of abstraction from the pure functionality to a skill level accessible to robotics behavior tasks, to enable the coordination of the functionalities the components provide.

The technology developer defines the software model of the components, including services (required or provided), the skills behavior blocks, and the variability the component is providing. The role is able to develop the component as a building block without the need to know in which specific application it is used.

The role of the *Technology Developer* contributes system parts on the abstraction levels of:

SKILLs which the role develops for the usage of the provided functionality on task level and the coordination and configuration of the components.

SERVICEs which the roles makes use of to develop the component. The component uses services offered by other components or provides services to other components.

FUNCTIONs as this is the technology experts main expertise, the role is expert in a specific domain, e.g. localization, object recognition etc..

System Integrator

The role of the *System Integrator* is composing the system out of the developed building blocks. The role uses the components and behavior models developed by their related roles and composes them to a working system and application. The integration is done in a composition manner, as the role selects the fitting components, with their behavior skill models from either custom developed or ecosystem available blocks. Additionally, the role binds some of the left-open variation points to tune the components to the application's need. The role of *System Integrator* requires knowledge of the application to build, as well as knowledge about the

building blocks the *System Integrator* needs to chose for the application. While in theory, the interfaces matching should be sufficient, in practice the selection of the "right" building blocks from the ecosystem, requires some expertise in technology to integrate and, to some extend, also in the application domain. Further than meeting the requirements of the application, those of the composed components need to be fulfilled as well. The role needs to manage the provided and required services of the components to enable the required inter component communication.

The role of the system integrator contributes to the software system on the abstraction levels of:

SERVICEs, by selecting the components with the services matching the applications' design requirements.

SKILLs, by selecting the components to fulfill the needs of the tasks that require them.

TASKs, by selecting the abstract tasks required to fulfill the needs of the designed application.

Robot

The *Robot* as the executing role in the workflow is responsible for executing all software system parts and closing the left open variation points at run-time. As an important difference to other software systems, this is necessary due to the challenges the open ended environment poses to the system. Not all the possible cases can be foreseen or even modeled during design-time. The role requires to "know" about the possible options and variations and how to use them to provide the desired services. The component and skill models explicate those options and provide rules to make use of the variability during run-time.

In contrast to the other roles, one could argue that the robot contributes to, or working with, all levels as, obviously, all levels are present during the execution. There are, however, some abstraction level in the system which need to be specifically pre-paired to the run-time aspects of the application:

SKILLs and TASKs, on both skill and task abstraction level the coordinating component closes variability during run-time. Thereby the robot adapts the execution to the needs of the application in the context of the environment. For example, which coffee machine to choose to fetch the coffee from, given the current environment and state of the system, e.g. position, the length of the queues, the order etc.

System Maintainer/Installer

The role of the *System Maintainer/Installer* is not always present and equally

weighted role as the others are. The role is responsible for changing an existing system with respect to the service it is performing. It is e.g. configuring the system to match a specific setup or to perform a new slightly different task. The role is not meant to be understood in the same way as if an existing system is undertaken the full development cycle again. The *System Maintainer/Installer* does not need to know any details about the system, the role needs, however, a good understanding of the application. The changes introduced by the role are primarily on the level of the robotics behavior task.
The role requires tailored tooling to work with the system, e.g. RobotinoFactory App see chapter 8.

Framework Expert

The role of the framework expert is realizing the stable interface offered by the execution container on different operating systems and middlewares. The framework expert provides the ground for the development of the component as well as the development of a coordinating component executing the behavior models. The role is not included in a workflow, as it is responsible for proving infrastructure "only".

The role of the framework expert contributes to the development of the system on the abstraction levels of: Execution Container, by realizing the level with a stable interface independent of the underlying operating system and middleware.

7.4. Robotic Development Workflow - Phases

This section describes the proposed workflow phase by phase. The roles involved and their relation to the ecosystem is described. The development workflow can be grouped into four phases containing different steps:
Starting with the **Design Phase** where the abstract design of the application is developed. The **Implementation Phase** is focused on the realization of the individual functional building blocks. During the **System Building Phase** the system is composed of the building blocks, namely components, skills and tasks. At the end of the proposed robotics development workflow stands the **Run-Time Phase**, where the resulting system is executed and the robot binds the left open variability.

7.4.1. Design Phase

The first phase for developing a new robotic application is the design phase. Following the top down approach, the tasks the applications should execute are selected, and iteratively modeled. The required functionality to realizes the application is defined. This is done without the need of incorporating any specific realization or realization technology within a functional block. The phase can be split into two iterations, first, a pure design step where the application is sketched, especially defining the used functionalities (via

the CoordinationModule instances and skills). Second, a task realization step where the sketched tasks are fully modeled. Figure 7.3, shows design phase the involved artifacts and their relation to the ecosystem.

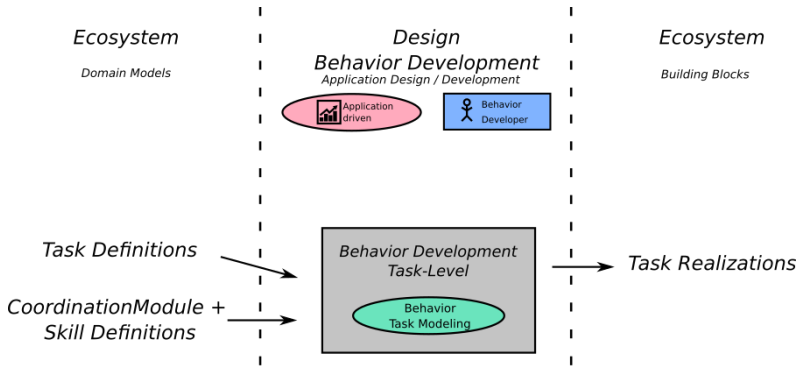

Figure 7.3.: Design and behavior development phase.

Example The example shown in Listing 7.1, illustrates the development of the task models in the design phase. The model is part of the robotic butler example, see section 8.2.6. To realize the task *pressButton* the behavior developer utilizes the skills (SkillDefinitions): *say* (DomainSpeech.TTSModule), *pressButton* (CommManipulatorObjects.manipulatorModule) and *grasp* (CommManipulatorObjects.gripperModule).

```
TaskRealizationModelTCL {
  AbstractCoordinationModuleInstance manipulator coordModuleDef
    CommManipulatorObjects.manipulatorModule
  AbstractCoordinationModuleInstance gripper coordModuleDef CommManipulatorObjects
    .gripperModule
  AbstractCoordinationModuleInstance tts coordModuleDef DomainSpeech.TTSModule

  (define-task-block (pressButton ?objectID)
    (rules (rulePressButtonFailed))
    (action ((let* ((obj (tcl-kb-query :kb-key '(is-a id)
                          :value '((is-a object)(id ?obj-id))))
                    (speech (get-value obj 'speech)))
        (format t "pressButton obj: ~a ?button: ~a ~%" ?objectID ?buttonNmbr)
        (tcl-push-plan :plan `(
          (tts.say ,(format nil "I'm going to press the button of ~a." speech))
          (gripper.grasp)
          (manipulator.pressButton ?objectID button-medium-offset)))
        '(SUCCESS ()))))))
        ...
```

Listing 7.1: Task block realizing the pressing of a button, utilizing a manipulator, gripper and text-to-speech functionality.

Involved Roles As this phase is driven by the needs of the application the role of the behavior developer is the main involved role within this phase. Depending on the nature of the developed application or the environment the application is developed, the behavior developer could also reach out to a broader audience such as e.g. a general assembly where the involved partners meet and define the coarse application's functionality.

Required Interfaces/Input From the idea of the robotics business ecosystem, the interfaces of the functional blocks required for developing the application might originate from the ecosystem. If no suitable interface definitions for a functional block are available the interface are defined within this phase. Using an existing interface definition from the robotics ecosystem enables the possibility to select existing building blocks from the ecosystem. If no matching functional block is available it enables the possibility to develop compatible functional blocks in the later development phases. During this phase, two different kinds of interface definitions are used. First of the CoordinationModuleDefinitions and the SkillDefinitions. They provide the access to the functional building blocks of the components. Second, the taskDefinitions, defining the interface of the tasks being realized within the current application. This enables the reuse of the developed tasks within other applications. Fixing the interfaces among the function blocks, also defining the behavior interface, enables the separated development of the functional blocks and the application specific robotics behaviors, as the separated roles can work on the defined interfaces.

Provided Results The results of the phase are two fold. First, the behavior developer or the design/behavior development phase as a whole, defines the architecture of the developed application in the sense that the interfaces of the functional blocks to the robotics behavior blocks are defined. Given the basic structural boundaries, the other development phases are able to work independently of each other.

Second, a developed robotics behavior with its tasks can be provided as an independent entity to other ecosystem participants. The tasks can be reused for the development of other applications, or the same application utilizing a different robot.

Driven By/From The design phase is mainly driven by the needs of the application. The role of the application architect is freed from the burden to think about the technological details functional parts feature. Iterative changes of the system architecture with input mainly from the system building phase is quite common, as the composed system faces the real-world after the composition.

7.4.2. Implementation Phase

The implementation phase, performed by the technology developer, realizes the functional building blocks, the components. Further than the functionalities within the

components, the technology developer implements the interfaces of the skills (SkillDefinitions) that provide access to the functionality for robotics behavior coordination. Figure 7.4, shows implementation phase the involved artifacts and their relation to the ecosystem.

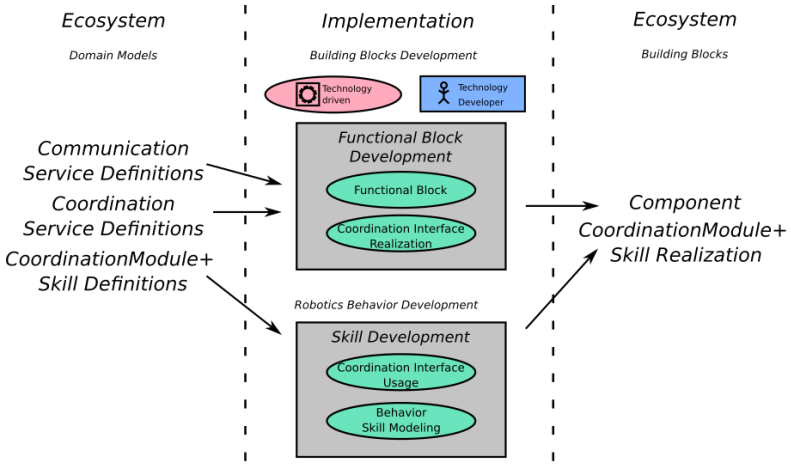

Figure 7.4.: Implementation phase, development of function building blocks and their skills.

Example The example shown in Listing 7.2, illustrates the development of the skill models in the implementation phase. The model is part of the robotic butler example, see section 8.2.6. To realize the skill *pressButton* the technology developer utilizes the coordination service *CommManipulatorObjects.ManipulatorCoordinationService*. The technology developer thereby realizes the skill definition defined within the domain models.

Involved Roles Within this phase, the technology developer realizes functional parts required to build any application. The role is only concerned with two things. First, and most important, the technology the role is an expert in and which the technology developer is encapsulating within a component. The technology developer is freed from the burden to think about how its parts are later on composed to a system realizing an application. Second, the technology developer needs to know what the interface to the functional parts are and what technology developer has to deliver.

Required Interfaces/Input The technology developer realizing the component's needs to know the interfaces for both the required services of the components require from other components and the provided services to other components. The role also needs to define the interface of the components towards the skills. Both interfaces can be obtained from the ecosystem's domain structure models. The interfaces and

```
SkillRealizationModel {
  CoordinationModuleRealization manipulator coordModuleDef CommManipulatorObjects.
    manipulatorModule uses {
    CommManipulatorObjects.ManipulatorCoordinationService instName manipulator
  }{

  (define-skill-block (pressButton ?obId ?buttonId)
    (skillDefinition  pressButton)
    (module "manipulator")
    (action (
      (format t "===>>> pressButton obj ~d ~%" '?objId)
      (let* ((obj (tcl-kb-query :key '(is-a id) :value '((is-a object)(id ?objId))
    ))
            (pose (get-value obj 'pose))
            (type (get-value obj 'type))

            (objClass (tcl-kb-query :key '(is-a type) :value `((is-a object-class)(
    type ,type))))
            (buttonOffset (get-value objClass ?buttonId)))

      (tcl-ci-state :server manipulator :state trajectory)

      (tcl-ci-activate-event :name evtPressButton
                             :handler handlerPressButton
                             :server manipulator
                             :service manipulatorevent
                             :mode continuous)

      (setf pose `(,(first pose) ;; x
                   ,(second pose) ;; y
                   ,(third pose) ;; z
                   ,(fourth pose) ,(fifth pose) ,(sixth pose)))
      (setf pose (eval (append '(transformPoseToPoint) pose buttonOffset)))
      (setf pose `(,(first pose) ,(second pose) ,(third pose) 0 0 0))

      (format t "button pose: ~s ~%" pose)
      (tcl-ci-send :server manipulator
                :service trajectory
                :param (append (list 'POSE) pose))
      '(SUCCESS ())))))
      ...
```

Listing 7.2: Skill block realizing the pressing of a button using the coordination service of a manipulator component.

thereby the contribution from the ecosystem the role makes use of are: Communication service definitions for horizontal inter-component communication. Coordination service definition, for the coordination access to the functionality within the component. The coordination module and skill definitions.

Provided Results The results of this phase are building blocks on different abstraction levels. For one, the components' developed within this phase realizing the functionality required to build the application. Second, the robotics behavior skill models that connect the functional blocks to the robotics behaviors and enable the coordination of the functional blocks. The skills are encapsulated with the coordination modules, which are realized and contributed as well.

Driven By/From The phase is driven by the needs of the technology to realize a certain functionality made accessible as a contribution to the ecosystem. In contrast to many other approaches in robotics software development, the robotics experts, typically the technology developer in this workflow, does not need to know how the application domain is ticking.

7.4.3. System Building Phase

The system building phase is the step where all developed parts come together to form the application. The integration of the separately developed parts is done in a composition-like manner. The individual parts are composed without the need to open them up again. No knowledge about the internals of the parts to be composed is required, there should also be no need to change the parts to make them fit to the applications need. In case change is required the according role (technology developer) is in charge of changing his building block. All required configuration options to make the parts fit the application needs to be modeled during the previous phase. Figure 7.5, shows system building phase the involved artifacts and their relation to the ecosystem.

Example The example shown in Listing 7.3, illustrates the mapping of the coordination modules in the system building phase. The model is part of the robotic butler example, see section 8.2.6. The system integrator maps the abstract coordination module instances used by the task models (*BehaviorButlerScenario*), with the realizations provided by the components, in this example the components *SmartURServerLegacy* and *SmartLoquendoTTS*.

```
...
  CoordinationModuleMapping {
    moduleInstance BehaviorButlerScenario.manipulator realizedby UR {
      interfaceInstance manipulator realizedby SmartURServerLegacy
    }
  }
```

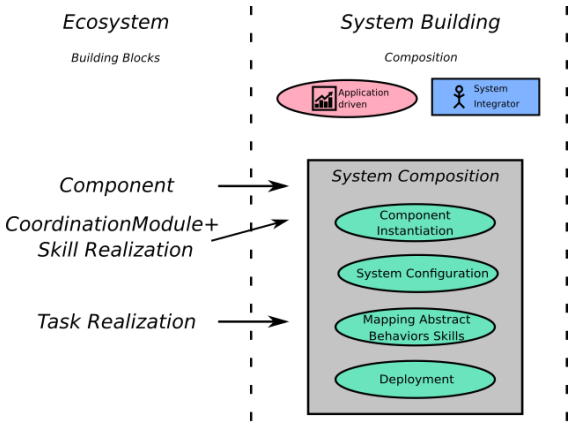

Figure 7.5.: System Building Phase, composing the building blocks to a system of components and a robotics behavior, both matching the applications need.

```
  CoordinationModuleMapping {
    moduleInstance BehaviorButlerScenario.gripper realizedby UR {
      interfaceInstance gripper realizedby SmartURServerLegacy
    }
  }

  CoordinationModuleMapping {
    moduleInstance BehaviorButlerScenario.tts realizedby LoquendoTTS {
      interfaceInstance tts realizedby SmartLoquendoTTS
    }
  }
...
```

Listing 7.3: Coordination module mapping is part of the system project and model. The excerpt shows the mapping of the module instances and the realizations.

Involved Roles The driving role of the system building phase is the system integrator. The role is able to compose the individually developed parts, functional or behavior to a working application. The robotics behavior on task abstraction layer, developed by the robotics behavior developer, acts as a blueprint in a top-down approach to select the matching functional and behavior building blocks.

Required Interfaces/Input This phase relies on the existence of three fundamental parts. First, the behavior developed during the design phase (sum of all tasks), as this is the blueprint to which functional building blocks the system should be composed out of. Second, the system integrator requires the components as functional building blocks to compose the system. Those building blocks could either be developed custom made for the application or being select from the ecosystem. Third, the system integrator requires the developed skill models and coordination modules, required by the tasks

and realized by the components. All of the three parts being composed during this phase typically require services or input from other building blocks, that needs to be connected prior to execution. At least, it should be clear that there will be a matching counterpart during run-time that matches the interfaces.

The components need input from other components and provide output to other components within the system. The (initial) connection of the components is set during this phase. In addition to the connection of the required services, the functional parts need to be configured to match the application's need. During this phase, the system statics or initial configuration options are being set. The configuration for coordination is done in the next phase during run-time.

The robotics behavior models, both task and skills, have so far been working with the abstract interface instances (coordination modules) to access the functional parts. Now, with the functional parts being selected by the system integrator, the abstract instances (e.g. from the task behavior models) needs to be mapped to the realizations (e.g. provided by the components).

Provided Results The system building phase realizes the application and provides a configured system consisting of the executables, the initial connections, and configured system parts. The robotics behavior models are connected to the selected functional parts and are ready for deployment to the system. As the overall workflow is typically executed iteratively, feedback is also provided towards the design and the robotics behavior development.

Driven By/From The phase is driven by the needs of the application only. The role of the system integrator is focused on the composition of the building blocks such that they fit the application's needs.

7.4.4. Run-Time Phase

Since a service robotics system is working within an open ended environment and with the challenges that arise from it, a robotics development workflow would not be complete if it does not cover the run-time aspects. Additionally, a versatile robotics system typically needs configuration during the installation of the application to make the system work within the new environment. In some cases, the robotic system might even be able to be reconfigured to perform a new task, without the need to come back to a full development cycle. Figure 7.6, run-time phase the involved artifacts and their relation to the ecosystem.

Involved Roles The main role that needs to be considered during the run-time is the robot itself. The robot needs to be able to handle the situations confronted by an open ended environment. The robot, or more precise parts on all different level, needs to be

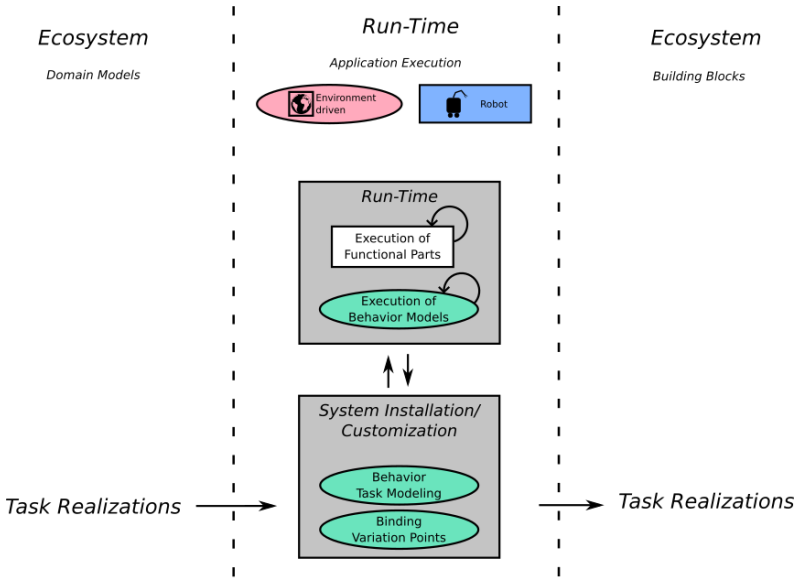

Figure 7.6.: Run-Time Phase, execution of the system and the behavior and installation and customization.

able to use its designed variability to work in an open environment. The second role involved during the run-time, or somewhere near the run-time phase, is the system maintainer/installer. The role is responsible for two things, first, the configuration of the developed system to the current working environment (e.g. maps, coordinate frames, objects, etc.), second realizing new tasks the system is capable to perform.

Required Interfaces/Input The main challenge to solve in robotics during the phase of run-time, at least from a software system development perspective, is robustness. Therefore, the system needs to be able to use design-time modeled variability at run-time. The components model variability via their behavior skills for run-time coordination. Those models together with the realization independent behavior task models form the required input for the system. Typically the sequencer component reacts to the challenges the environment poses the robotics system with. The role of the system maintainer/installer requires three things, first and obvious the full system to configure, second the configuration options (variation points) to be able to configure the system to the applications' needs. Those options are mainly defined within the implementation phase, explicated in component models and task or skill behavior models. Third, the role needs task or skill level behavior blocks to realize new task, without the need of going through a full development cycle involving other roles and changing the application architecture. Those blocks are handed over as well. The role of the system maintainer/installer typically makes use of tailored tooling, which transforms the models to easier to deal with representations, e.g. RobotinoFactory app see chapter 8.

During system customization new compatible robotics behavior tasks can be received from the ecosystem and composed to the system. While this might also be possible with function building blocks (components), the composition of the new components is most likely too challenging for the role of the system maintainer/installer, even with composable components. The selection of possible (explicitly defined) interchangeable alternative components is, however, more realistic and has been demonstrated.

Provided Results As the run-time of the system is the last step in the development workflow, the result is ultimately the running-robotic application. The role of the system maintainer/installer uses the system with its configuration options and different models to configure and to create new tasks the system can perform. As a result, the input system is modified and does now work given the current environment, or the newly enhanced system is capable of performing an additional slightly different task. This new task could be provided as contribution to the ecosystem as well, possibly being labeled as specialization.

Driven By/From While the other phases are driven by technology or the needs of the application this step is driven by the environment and the application. This includes other systems providing information or jobs to be executed, as well as the physical environment a robot is working within. The robot has to be able to react to those different influences from the environment it is working in, to provide a reliable service. The changes of new tasks the role of the system maintainer/installer might introduce, are driven by the application's need.

7.4.5. Summary

The proposed workflow organizes the handover between the involved roles. It relates the roles and their contributions to the development of applications and the ecosystem. The presented workflow illustrates how applications can be developed with separated roles in an ecosystem context. The tooling supports the roles in the development and their collaboration, the usage of the tooling, further than the provided example, is illustrated in chapter 8.

8. Experiments and Validation

The approach presented in the previous chapters has been applied to several systems and applications built with the SmartMDSD Toolchain. This chapter presents excerpts of chosen examples from existing systems to demonstrate the approach and to illustrate how it could be realized and applied. Furthermore, the benefits and the drawbacks while applying the approach to support the development of robotic systems and applications via composition in an ecosystem context are discussed.

The chapter first presents an overview of some applications that have been realized with the presented approach. The overall applications are outlined to illustrate the applicability of the approach to real-world systems. In the second section, excerpts of the applications are presented as experiments to evaluate and validate the approach.

The presented approach contributes to the idea and the foundation of a robotic business ecosystem, which enables the collaboration of the separated roles to realize systems and applications. The experiments and examples presented in this chapter illustrate this aspect. As the ecosystem envisioned is not existing yet, it is fair to state that most of the systems shown in this chapter have been developed by the SRRC group at the Technische Hochschule Ulm (THU). Even though developed by a single organization, different people have been involved in, collaborating, and contributing to the systems living the idea of separated roles. Some of the examples, however, also involve external partners from both academia and industry. Those examples demonstrate the approach and separation of roles on a larger scale, as has been envisioned. The following Table 8.1 outlines the purpose and content of the experiments presented within this chapter. Further technical experiments are provided in the appendix, section A.4.

Title:	**New Capability Building Blocks**
Section:	8.2.1
Content:	Composition of functionality with behavior coordination
Description:	The experiment illustrates the development of functional building blocks (software components) and the connection between them toward robotics behavior coordination. Both sides of the component coordination interface are demonstrated, the component wrapping the functionality as well as the skills coordinating the component.

Title:	**Behavior and Task Development**
Section:	8.2.2
Content:	Composition of skills and tasks
Description:	The experiment illustrates the development of the robot behavior by composing skills and tasks. The development of tasks separated from concrete functional building blocks is demonstrated.

Title:	**Composition of Behavior and System to Applications**
Section:	8.2.3
Content:	Composition of behavior and system
Description:	The experiment evaluates the composition of robotics behaviors with the system of components. This primarily includes the mapping of the used resources (coordination modules) by the tasks, with those provided by the components composed to systems.

Title:	**Modification of Existing Robotic System**
Section:	8.2.4
Content:	Composition of building blocks, same interface different realization
Description:	The experiment demonstrates the modification of an existing robotic system by swapping a functional building block, as well as the building blocks for coordination. The experiments demonstrate the ability to make use of the same tasks with different functional building blocks realizing the same coordination module and skills.

Title:	**Adding Tasks to Existing System**
Section:	8.2.5
Content:	Composition of tasks and system, resources and dependencies
Description:	The experiment demonstrates the ability to compose existing or newly created tasks to an existing system. The experiment further illustrates how the approach deals with the required resources and the dependencies among the system parts.

Title:	**Transferability of Tasks to other Robotic Systems**
Section:	8.2.6
Content:	Composition of behavior and system, transferability of behaviors
Description:	The experiment demonstrates the ability to transfer tasks to other robotic systems and thereby the independence of tasks from concrete functional building blocks by making use of the skills and the coordination interfaces.

Title:	**Skill Realization Dependencies**
Section:	A.4.2
Content:	Composition of skills, using foreign resources
Description:	The experiment demonstrates the realization of skills that make use of other skills contained within foreign coordination modules. Therefore, the experiments illustrate how the encapsulation of the coordination modules is contained while still being able to make use of resources (skills) provided by other coordination modules.

Table 8.1.: Outline of the experiments presented within this chapter.

8.1. Applications and Systems

8.1.1. Robotino Factory 4.0

The application is situated in an industry 4.0 environment and realizes the transportation of parts within small load carriers between modular production stations. The mobile robot platform used for the application is a FESTO Robotino in Versions 3 and 4.

Figure 8.1.: Fleet of robots providing a transportation service between modular production stations.

The foundation of the system, especially the coordinated fleet navigation, has been developed in the BMBF KMU-innovativ Projekt *LogiRob (Multi-Robot-Transportsystem im mit Menschen geteilten Arbeitsraum)* [Bil]. The system comprises three parts. First, a fleet of mobile robots; second, a fleet management system; and third, a configuration tool to set up the fleet and the application (RobotinoFactoryApp). There is an optional fourth part that uses a data analytics engine (Elasticsearch) to monitor and optimize QoS attributes of the fleet services.

The fleet management system and the software on the robots is realized using the presented approach. The system consists of composed software components and robotics behavior coordination models. The configuration tool is a conventionally made software tool interfacing the fleet via REST services. The whole application has been available as a commercial product since several years and are iteratively extended ever since. Figure 8.1 shows the system operating in an open environment, while transporting parts between FESTO modular production stations.

The fleet uses a coordinated fleet navigation system [LVS16] that is able to operate a fleet of robots in confined environments shared with humans. The system offers many functionalities where the same or similar functionality is provided by alternative implementations that are composed to form the system. Dependent on the context (e.g., which type of station to dock to), some of the options are selected during the start of the robot (e.g., dependent on the H/W configuration), while some of them are bound during runtime depending on the context (e.g. which type of station to dock to).

The software components the system is composed of have been developed by three different partners—Technische Hochschule Ulm, Robotics Equipment Corporation GmbH, and Festo Didactic SE. Figure A.42 shows the component architecture model of a single robot in the fleet. The system consists of 20–31 components, depending on the configuration, which, taken together, provide more than 100 skills available for behavior coordination. The model visible in Figure A.42 (appendix) shows a typical robot configuration (selected components) and the initial wiring between them. The connections

will be altered during runtime depending on the context and coordination realized by using the skill and task-behavior models.

8.1.2. Collaborative Order Picking

The application is situated in an intra-logistics warehouse environment with goods transported in boxes. The mobile robot used for the application is a FESTO Robotino v3. The system realizes a human-robot collaborative order-picking application. Human workers perform the picking of difficult to grasp individual items (e.g., flyer, folding boxes), while a fleet of mobile robots supports them in transporting the goods during the order-picking process. Figure 8.2 shows the robots operating in the warehouse.

The application makes use of the same software system as the *Robotino Factory 4.0* application. It extends the system by a few components, especially for human-robot interaction, such as the person following or a dedicated graphical user interface on the robot. The application is further extended by new behavior coordination task blocks specific to the application. The new task blocks make use of the same skills as the previous applications. Therefore, this application is a good example of how to modify an existing system to realize new applications by reusing components and skills. Listing 8.1 shows a newly created task block of the application. The new task uses existing tasks to deliver and fetch boxes from stations, as well as skills for navigation and to perform user interaction via a graphical user interface.

Figure 8.2.: Collaborative order-picking application in a warehouse environment. A fleet of robots assist human workers performing order-picking tasks.

```
(define-task-block (orderPickingJob ?dropOffStation ?dropOffBelt
                                    ?emptyBoxStations ?emptyBoxBelt
                                    ?parkingLocation)
  (taskDefinition CommNavigationObjects.orderPickingJob)
    (rules (ruleAbortJobErrorAckRestart))
    (action ( (format t "============= >>> orderPickingJob~%")
      (let*   ((empty-box-station (tcl-kb-query :kb-key '(is-a id) :kb-value `((
    is-a station)(id ?emptyBoxStations))))
              (empty-box-station-approach-location (get-value empty-box-station '
    approach-location))
```

```
9                (drop-off-station (tcl-kb-query :kb-key '(is-a id) :kb-value `((
    is-a station)(id ?dropOffStation))))
                 (drop-off-station-approach-location (get-value drop-off-station '
    approach-location)))
11       (tcl-push-plan :plan `(
           ;; fetch empty box
13         (localizationModInst.activateLocalization)
           (fleetNavigation.approachLocation ,empty-box-station-approach-location)
15         (mpsInst.mpsStationFetchFrom ?emptyBoxStations ?emptyBoxBelt)
           ;; park robot
17         (fleetNavigation.approachLocation ?parkingLocation)
           ;; wait for jobs from user
19         (logisticsGui.getUserInput => ?inputFormUser)
           ;; evaluate user input and execute job
21         (executeNextStep ?inputFormUser)
           ;; finalize jobs
23         (fleetNavigation.approachLocation ,drop-off-station-approach-location)
           (mpsInst.mpsStationPushTo ?dropOffStation ?dropOffBelt)
25         (localizationModInst.deactivateLocalization)))))))
```

Listing 8.1: New task block specific to the application, composed of existing tasks and skills, together with their corresponding coordination module instances.

8.1.3. Autonomous Order Picking

This application is again located in an intra-logistics warehouse environment. The fleet of robots consists of two different robot platforms—Robotino v3 robots for transportation and the UR5 equipped service robot Larry for mobile manipulation. The application has been developed in the context of the ZAFH Intralogistik - Center for Applied Research funded by “ERDF and the state of Baden-Württemberg" [ZAFHI]. The fleet of heterogeneous robots realizes an autonomous order-picking application. Figure 8.3 shows the system operating in a warehouse.

The application realizes the order-picking of pharmaceutical items and is split into two parts. Similar to the *Collaborative Order Picking* application, a fleet of Robotino robots is used to execute the transportation of the picked items. The service robot Larry executes the picking part of the application which was previously done by human workers. Larry recognizes, grasps, and puts the items into small load carriers on the Robotinos. The items to pick are located in a-frame style shelves organized as single instances or loosely contained in large cardboard boxes within inclined shelves (Figure 8.3). Alternative realizations of the same recognition skills are provided by two components to best match the different contexts of the inclined and the A-frame style shelves. Each approach exploits the given context to realize a robust recognition of the items to pick.

This application is a good example to show a large number of software components and skills that are shared among the systems, despite the different robots used, see Tables A.11 and A.12 in the appendix. The reuse of building blocks can be demonstrated. Figure 8.4 shows the component architecture model of Larry. Most of the navigation, task coordination, mapping, and localization components are the same as used on the

Figure 8.3.: Autonomous order-picking application in a warehouse. A fleet of heterogeneous robots perform order-picking tasks. A combination of manipulation robots (service robot Larry) and cheaper transportation robots (Robotino) is used to enhance the performance of the robot fleet.

Robotino robots. The system is, however, extended by a set of perception and recognition components, as well as those for the manipulation of objects.

8.1.4. SeRoNet - Gradual Automation of an Assembly Line

This application has been realized in the context of the BMWi PAiCE project *SeRoNet (Plattform zur arbeitsteiligen Entwicklung von Serviceroboter-Lösungen) [Wir17]* and the BMBF KMU-innovativ Projekt *LogiRob (Multi-Robot-Transportsystem im mit Menschen geteilten Arbeitsraum)* [Bil]. The application shows how to gradually automate a traditional assembly line. The SeRoNet project partners Technische Hochschule Ulm and Daimler TSS illustrate this by realizing the assembly process of a Telematic Communication Unit. The application is gradually changed to add further automatization, with Figure 8.5 illustrating the change in the process of the application. An increasing number of production steps are included in the automation assembly line, two staged assembly and inspection, automated commissioning, and parallel execution of lines. Within the process, a fleet of mobile Robotino robots automatize the transportation of products, while a stationary KUKA manipulator together with the transportation robots is used to perform the commissioning of the finished products to structured shipping boxes (video published by the project partners [Lut+19b]).

The application consists of four parts: a central fleet management, a fleet of Robotinos to provide transportation services, a manipulation system developed by TSS using a KUKA manipulator, and the Robotino Factory App for system configuration. A simulated MES provides the jobs to be executed by the system, distributed, and organized by the fleet management. Figure 8.6 shows an overview of the systems architecture.

This application is of particular interest as it targets the flexible rearrangement and

Figure 8.4.: Component architecture model of the service robot Larry. The model shows the components of the system with the most important connections in the initial wiring. The most important differences between the system model of Larry and one of the Robotinos are highlighted in red.

the extension of an existing application. This is what can be done most easily by reusing the existing task and skill blocks. Changing the behavior of the application only is one of the easiest ways to realize new applications and can be done by the staff on the premise. The composition of a new task or skill blocks is rather easy given the proper tool support, whereas the composition of new software components typically requires more knowledge of the system. Figure 8.7 shows an example task developed within the SmartMDSD Toolchain. The toolchain supports the behavior developer in modeling tasks with automatic code completion, syntax and semantic model checks, syntax highlighting, etc. The example shows a transportation task using navigation docking and localization skills and the related coordination module instances.

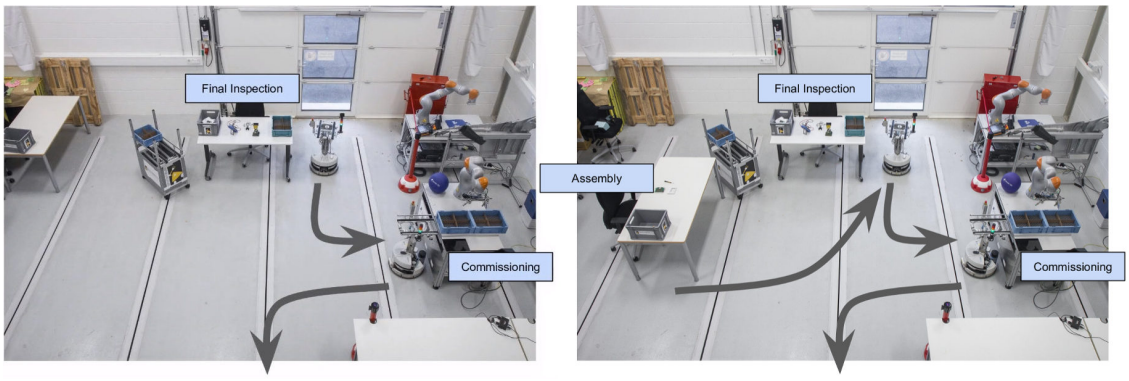

Figure 8.5.: Rearrangement of the scenario by the worker/staff on the premise. The grey arrows indicate the transportation of products throughout the process. The left image shows the initial setup with one inspection station only, while the right image shows an extension with a second station. Pictures from video published by SeRoNet [Lut+19b].

8.2. Experiments

This section presents excerpts of the previously presented applications as experiments to evaluate and validate the presented approach. In this section, many figures showing the SmartMDSD Toolchain are presented. This is to best provide an impression of how the presented approaches are realized in concrete tools, and how the users within their roles are addressed and supported. While in the previous sections textual models are printed as text, this section resorts in many cases to the screenshots of the SmartMDSD Toolchain. This allows to also present the tool support. To save space, some of the subsequent screenshots of the toolchain are reduced to the parts of interest.

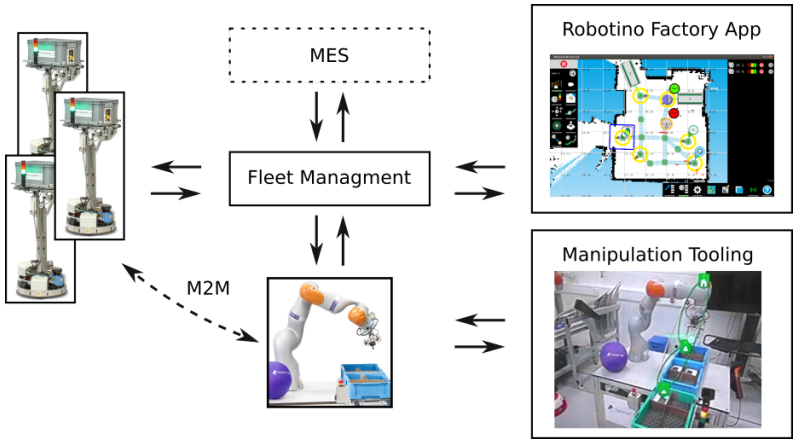

Figure 8.6.: Gradual automation of an assembly line, system overview. Besides the central fleet management, the mobile robots interact directly with the manipulation system developed by TSS. Setup and configuration of the transportation fleet is done using the Robotino Factory App. The manipulation system uses a dedicated tooling developed by TSS to set up the manipulation tasks.

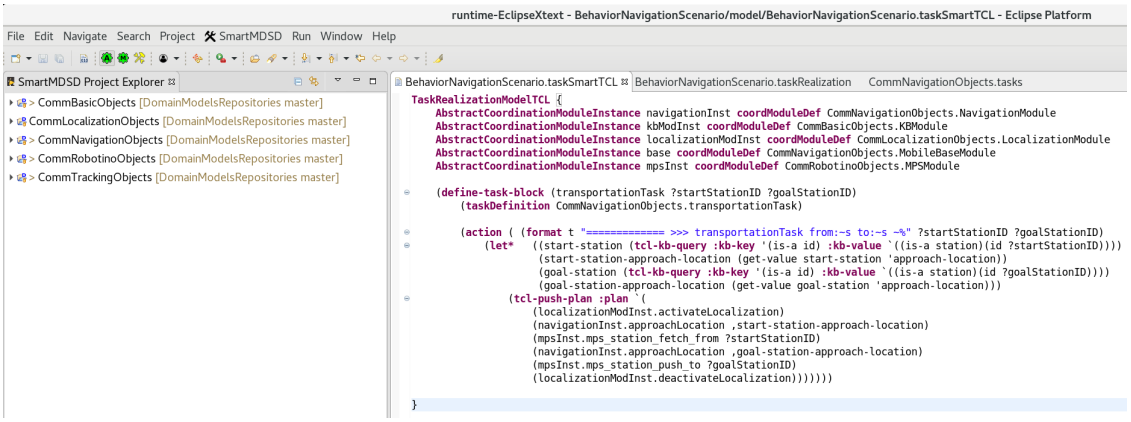

Figure 8.7.: Task development in the SmartMDSD Toolchain. It supports the different users with automatic code completion, syntax and semantic model checks, highlight and other tools. The SmartMDSD Toolchain offers a view tailored to the behavior developer, exploiting the features of the Eclipse Modeling Framework.

8.2.1. New Capability Building Blocks - Composition of Functionality with Behavior Coordination

This first experiment validates and illustrates the development of a new capability building block from a robotics behavior coordination perspective. The functionality to wrap by a component is the docking to a Festo MPS station, as is used, for example, in

the application *Robotino Factory 4.0*.

Wrapping Functionalities, Connection to Coordination - Component Coordination Interface

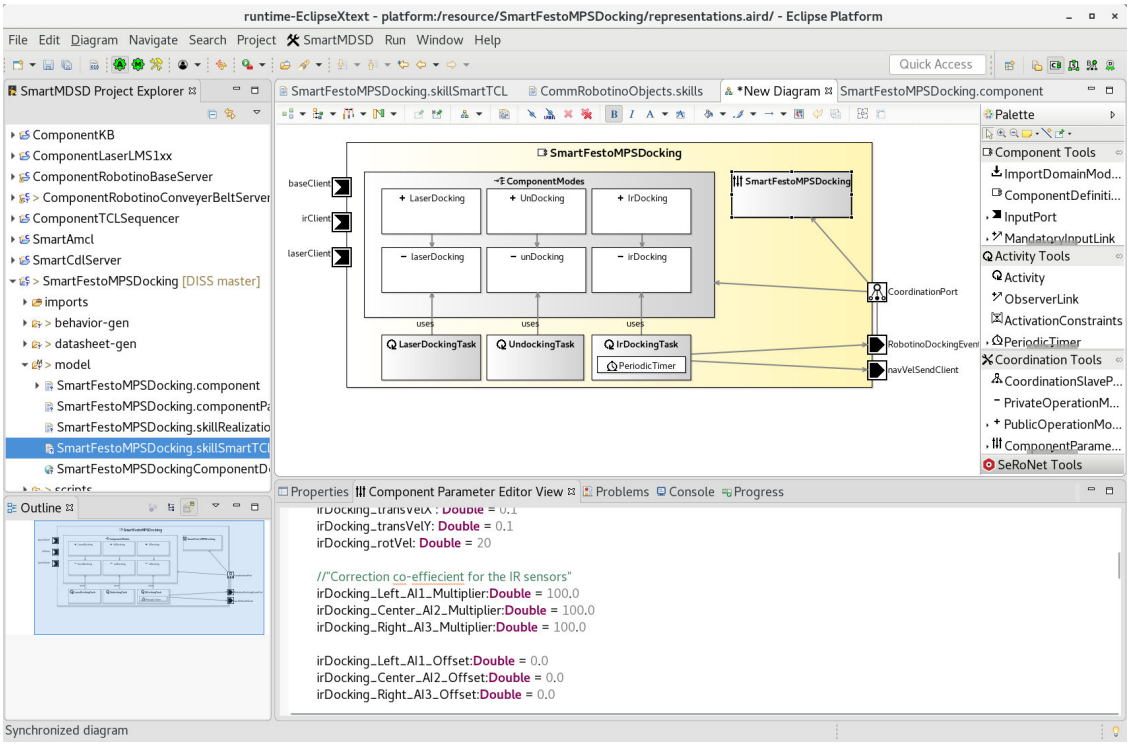

Figure 8.8.: SmartMDSD Toolchain, graphical representation of the component model, as is developed by the component developer. Left bar shows the components in the project explorer. The right toolbar contains the elements to be dragged to the model at the center. The bottom bar shows the parameter model of the component.

The component developer models the component following the interfaces defined within the domain models. The component developer chooses the services for component-component interaction (e.g. to access laser data) and coordination. The component model is accessible in a graphical and textual representation shown in Figure 8.8 and 8.9. Within the Figure showing the textual representation, the selected coordination service *CommRobotinoObject.FestoMPSDockingCoordiationService* is visible. The component developer selects how the individual parts of the services are realized. In this example, the services consist of the three parts, state parameter and event, as is visible in Figure 8.10 showing the definition in the domain models.

When developing the component, the component developer models three cyclic

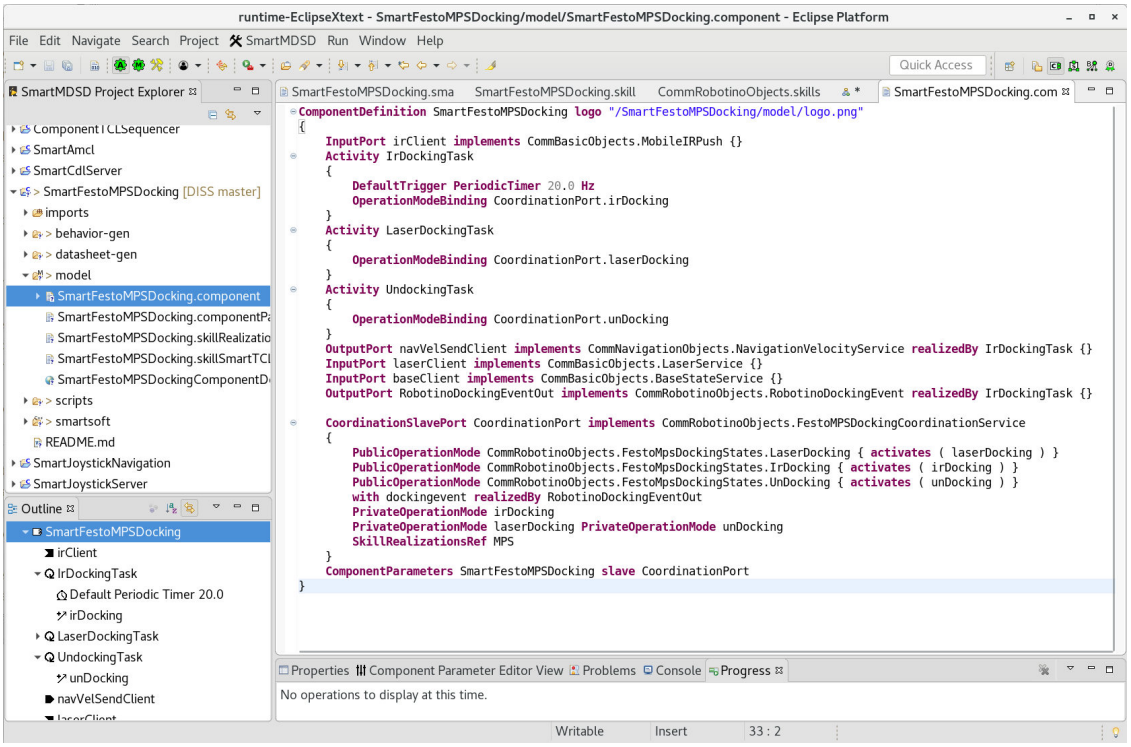

Figure 8.9.: SmartMDSD Toolchain, textual representation of the component model, corresponding to the graphical one, Figure 8.8. The used coordination services as well as further details are visible.

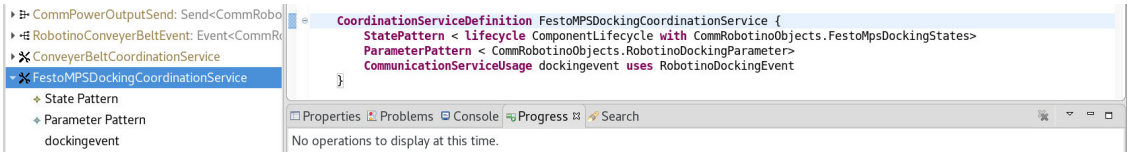

Figure 8.10.: SmartMDSD Toolchain, the definition of the coordination service with the domain models, here using the state parameter and the event pattern.

running activities, namely laserDocking, irDocking, and unDocking. This is where the component developer links to the realization of the functionality typically using libraries. Each activity is activated via the state pattern, and the connection can be realized graphically as well as textually, see the figures of the component model. The connection between the activity and the state is realized in the generated C++ code which is not visible to the component developer.

The parameter model of the component is visible in Figure 8.8 at the bottom. The component developer is able to define the use of the start-up as well as the runtime parameters within this model. The used runtime configuration domain model defines a single trigger among the parameters, see Figure 8.11. While the parameters are accessible

via getter methods, the triggers result in up-call handlers.

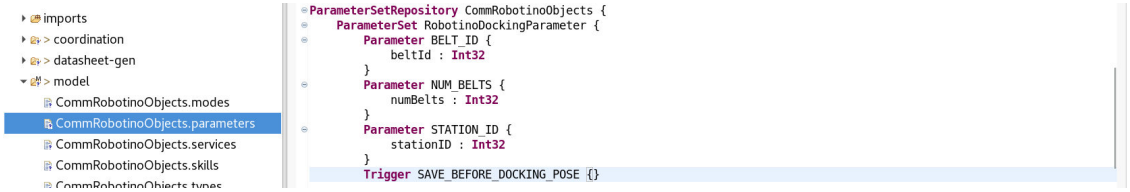

Figure 8.11.: SmartMDSD Toolchain, the definition of run-time parameters as well as the trigger activations in the domain models.

The *dockingevent* used by the coordination service is the event that is realized using the matching type service realization within the component, as is visible in both the textual Figure 8.9 and the graphical Figure 8.8 representation of the component model.

The component together with the related domain models are used to generate C++ source code for the implementation. The component developer is able to access the generated model elements within the C++ realization. In principle, the generation gap pattern [Vli98] is applied throughout the SmartMDSD Toolchain to decouple the implementation (user code) from the generated source code wherever reasonable.

The toolchain supports the different roles with code completion, syntax highlighting, structure outline, etc. The parameters' runtime and the start-up are accessible via a component wide singleton. Figure 8.12 shows an example of how to get a parameter via *COMP->getPrameters* a copy of all current valid parameters (commit protocol) is returned to get them in a consistent state. All the necessary classes, structures, and methods to realize the pattern, as described in Chapter 6.4, are generated from the model.

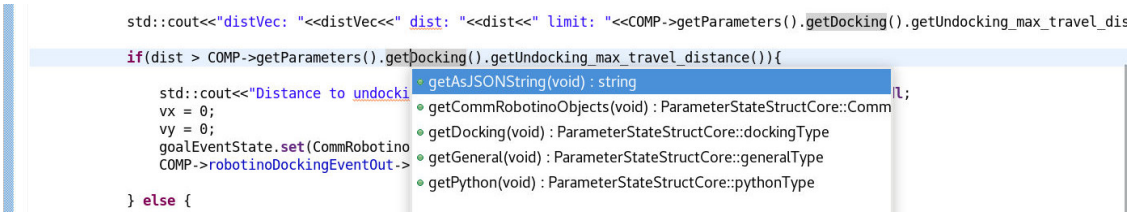

Figure 8.12.: SmartMDSD Toolchain, parameter access for implementation of functionalities within components.

The trigger modeled in the domain models results in an upcall-handler that the component developer needs to implement. Figure 8.13 shows the handler for the trigger *SAVE_BEFOR_DOCKING_POSE*. The example shows the use of the handler to memorize the current pose of the robot.

The use of all types, connections, definitions, etc. are bound to the corresponding definition within the domain models. There is no direct dependency between this component and others, which would break the idea of composition. This holds true for both

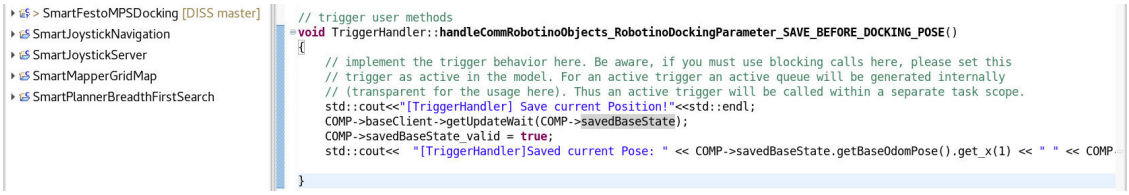

Figure 8.13.: SmartMDSD Toolchain, trigger upcall handler within a component.

functional realization and the coordination access to the component implementation.

Wrapping Functionalities, Tasking Access - Skills and Coordination Modules

The component developer additionally develops the skills, as defined by the coordination module, as well as the coordination and communication services defined within the domain models. This example makes use of the coordination module *MPSModule* containing several skills logically grouped around the interaction of the robot with the Festo MPS (Modular Production Stations). Figure 8.14 shows an excerpt of those domain model definitions within the SmartMDSD Toolchain.

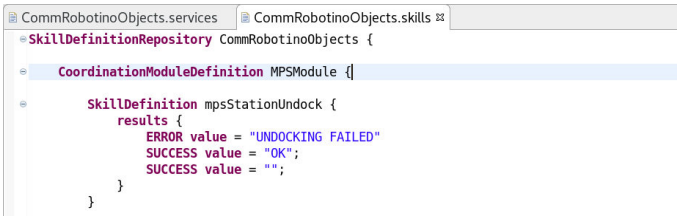

Figure 8.14.: Coordination module and skill definition, defined within the domain models.

Following the definitions, the component developer realizes the skills within the coordination modules. The SmartMDSD Toolchain supports the component developer with the development of the skills. Figure 8.15 shows the skill model realized the component developer. The first few lines in the model define the instantiation of the coordination module following a specific type. Within the header section, the role further defines which coordination services are used within this coordination module. This also includes the number of instances of the type. The example shows the module instantiation of a single coordination service only.

The skills are defined within the coordination module, the example shows the *mpsStationDockLaser* skill. This skill provides the functionality to dock to a station using data from a laser ranger and by tracking retro reflector markers mounted to the station's front. The realization of the skill is not bound to a specific component. It is only dependent on the coordination interface. More important than independence from the component is, however, independence from a specific application and the task realizing them.

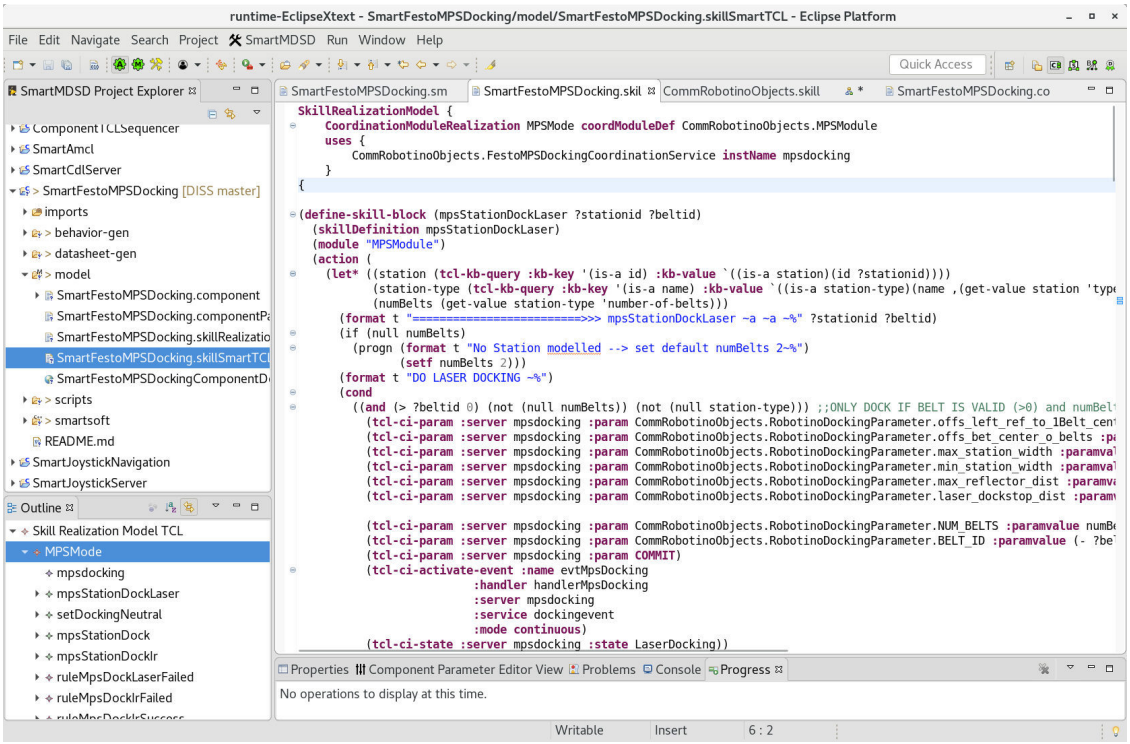

Figure 8.15.: SmartMDSD Toolchain, realization of skills in coordination modules, as seen by the component developer.

The toolchain supports the role in the development of the skills. Figure 8.16 shows the completion provided when accessing the activation of the component via the state pattern. In this example, the state *LaserDocking* is configured and used to activate the laser-docking activity within the component.

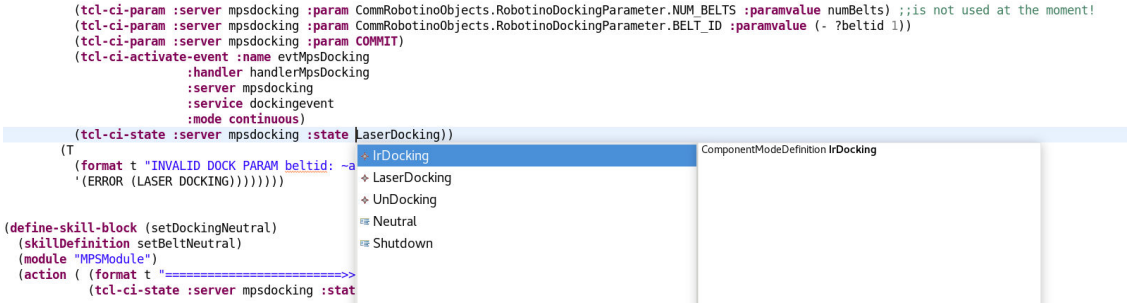

Figure 8.16.: SmartMDSD Toolchain provides support to realize the coordination of the components. In this example, the skill activates the component by selecting its state *LaserDocking*. The context menu shows all valid states, as defined within the domain models.

8.2.2. Behavior and Task development - Composition of Skills and Tasks

This experiment evaluates and demonstrates the composition of skills and tasks realize the behavior of the robot as well as the task development itself. The experiment ends with the developed behavior not including the composition of the behavior with a system of components, which is shown in the next experiment. The developed tasks are separated from any concrete component or sets of components. The behavior of the robot is developed by instantiating coordination modules and using the provided skills.

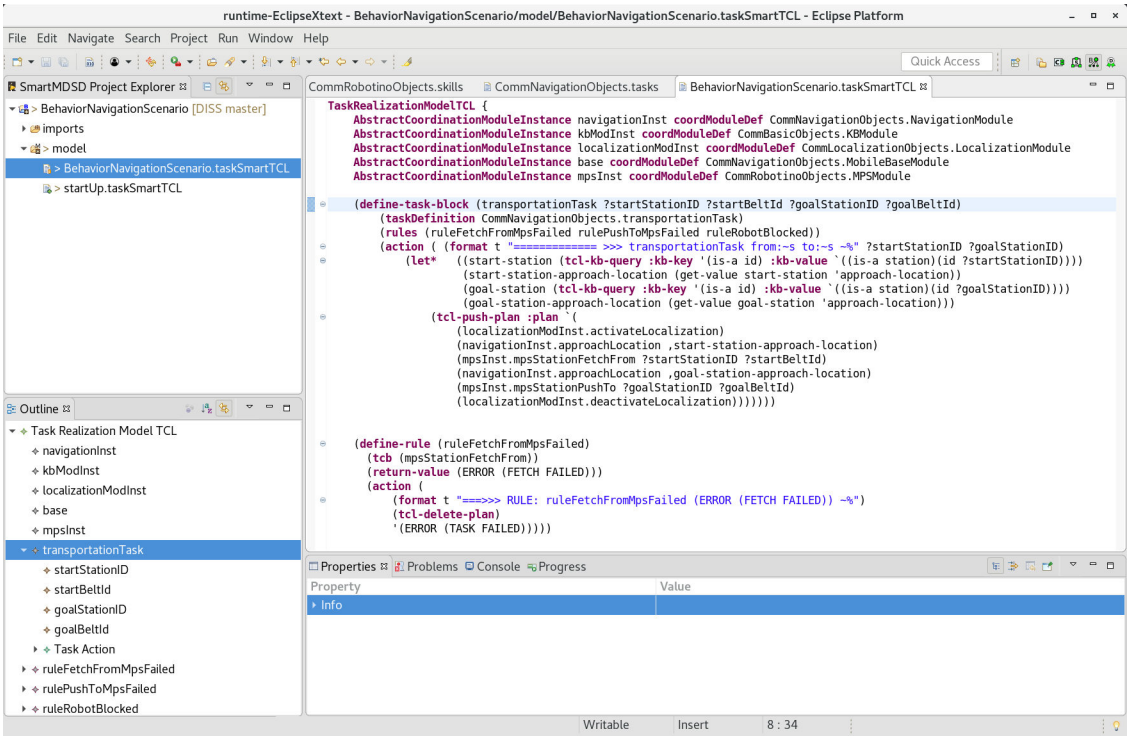

Figure 8.17.: SmartMDSD Toolchain, behavior project. A transportation task is modeled using skills offered by the coordination modules instantiated within the first few lines of this example.

The behavior developer is supported with dedicated views within the SmartMDSD Toolchain. Figure 8.17 shows a behavior project with task models being developed. Model checks, auto-completion, syntax highlighting, etc. are provided to ease the development of the tasks. Figure 8.19 shows an example where a wrong (not existing) skill is used, with the toolchain highlighting the error and proposing possible repairs. The toolchain suggests valid skills as defined by the coordination module as possible repairs. Additional to the order of skills, the business logic of the task resolves the connection between the station id and location via which the station is approached. The task uses the knowledge base to look up the required information. In the same way as with the skills, the tasks also feature a type definition to make use of the tasks developed

by others without the need to resort to the realization. Therefore, the interface of the tasks is defined within the domain models. Figure 8.18 shows the definition of the task type realized in this example.

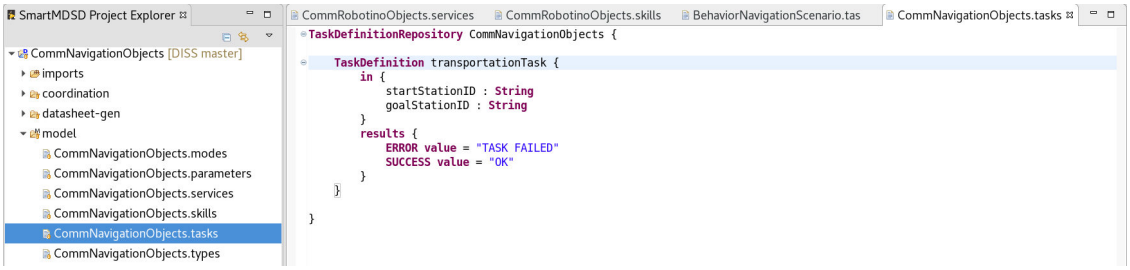

Figure 8.18.: Domain model definition of a task, binding the interface other tasks can make use of.

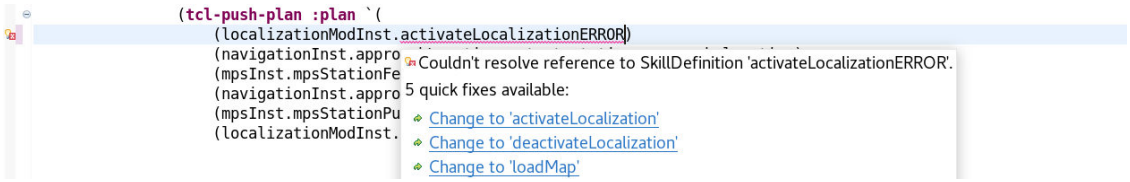

Figure 8.19.: The toolchain supports the role with checks and proposals of how to fix errors. In this example, an undefined skill *activateLocalizationERROR* is being used. The definition of the used coordination module, *LocalizationModule*, does not define such a skill. The tooling highlights the error and proposes fixes – in this case, the skills that are defined by the module type *LocalizationModule*.

When realizing a new robot behavior, the behavior developer can choose from the skills offered by the coordination modules, both defined in the domain models. In this example, the behavior developer chooses to use skills from the domain model repositories Basic, Localization, Navigation, and Robotino. The first few lines in the task model visible in Figure 8.17 instantiate the coordination modules from those repositories. The *NavigationModule* to make use of the skills for navigation of a mobile robot. The *KBModule* for the knowledge base for knowledge representation, mainly for behavior coordination. The *LocalizationModule* is used to coordinate the localization of the mobile robot. The *MobileBaseModule* is used for the mobile base itself and finally the *MPSModule* to interact with the Festo MPS. In this example, each module type is instantiated once and the names of the instances are defined by the behavior developer. The instances of the modules are used to define the execution context of the skills they provide.

The task *transportationTask* realizes the transportation of goods in small load containers between production stations. Overall, the task itself is rather simple. The core logic is a straightforward execution of the skills listed in the push-plan statement. The simplicity

of the task realization is made possible by the concept of the skills, which provide a high level of abstraction for robotics behavior development. Each skill is pushed to the plan using a prefix, namely the coordination module instance. Therefore, the skills are executed in the related context of this instance during runtime.

The example shows that the task is not directly dependent on any realization of the skills of the components. Thus, the task block captures the procedural knowledge of the domain or the application the task is developed for, while the realization of the skills with a concrete system could be done using different components and different approaches. For example, the skill to fetch a box from a MPS station can be realized using a docking approach based on laser scans and ir information, or using computer vision, or even realized by simply asking a human operator to load goods to the robot.

8.2.3. Composition of Behavior and System to Applications

This experiment evaluates and demonstrates the composition of robotic behaviors (tasks using skills) and sets of components. This primarily includes the mapping of the used abstract coordination module instances with those provided by the components of the system, and their coordination module realizations. The composition of robotics software components to systems without coordination is shown in [Sta+16]. The composition is performed by the system builder, thereby realizing a concrete application. This experiment picks up the threads of the last experiment and continues them.

The composition of the components results in a *component architecture model* with component instances, start-up configurations, and initial wiring of the connections between the components. The graphical representation of the model within the SmartMDSD Toolchain is visible in Figure 8.20. The system builder developed this model by instantiating and composing the components graphically.

System integration from a robotics behavior coordination perspective adds the most important the robotics behavior models on task level. The previously developed tasks are added to the system. The models are part of the dedicated behavior project type integrated into the SmartMDSD Toolchain, illustrated by the last experiment. Therefore, the project is added via a reference to the system project. With the imported behavior project, the abstract instances of the coordination modules used within the task models are known and open to be bound. The system builder needs to bind the abstract instances to the components realizing the matching coordination modules. The user is tool-supported in doing so, only modules with matching types can be chosen. The SmartMDSD Toolchain further proposes matching modules and highlights missing mappings. Figure 8.21 shows an excerpt of the textual component architecture model, realizing the described mapping.

With the coordination modules mapped, the runtime execution of the skills and tasks performed by the sequencer component is possible. The abstract instances and skills used within the task are bound to the realizations provided by the components. In this example, the abstract coordination modules instance *navigationInst* of the type *Comm-*

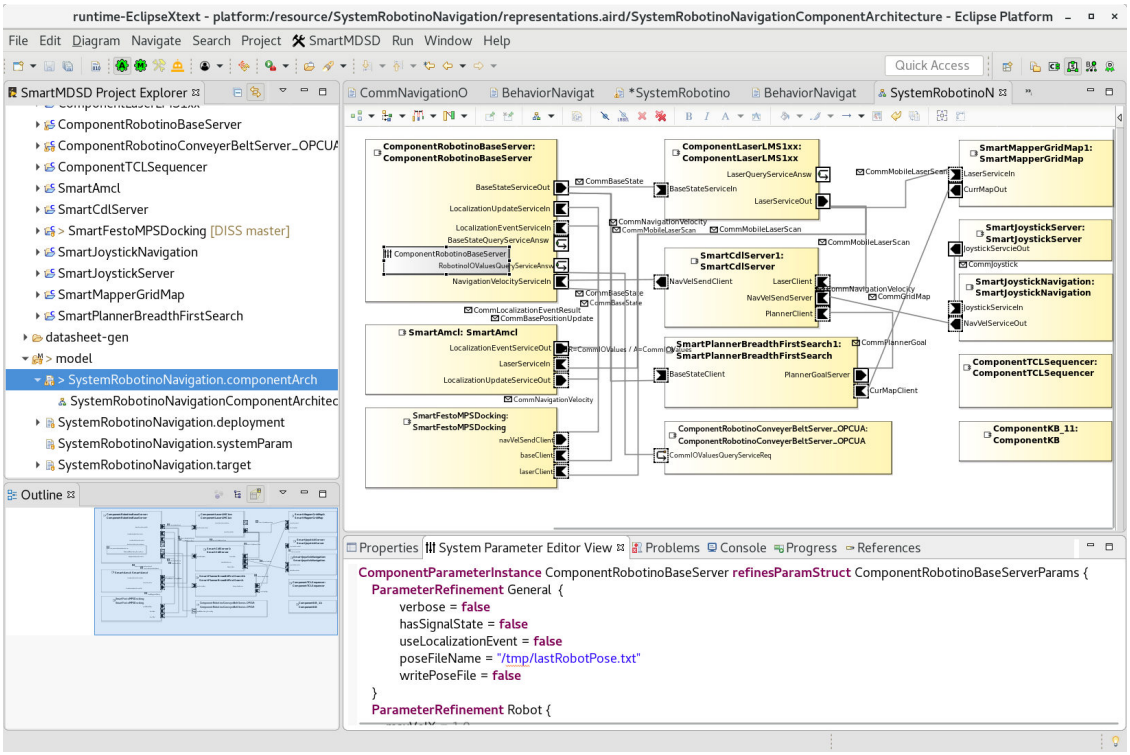

Figure 8.20.: Graphical representation of the component architecture model of the system project. The system builder is able to instantiate, connect, and configure the components graphically.

NavigationObjects.NavigationModule is bound to the realization *PlannedCdlNavigation* provided by the components SmartCdlServer, SmartMapperGridMap, SmartPlannerBreadthFirstSearch, and their instances within the system.

The bottom-up order of composition described in this example—composing a system of components to the behavior models—is only one possibility to realize an application. It is also easyl possible to start with the behavior model and the selected coordination modules, the system builder is able to select matching components from the ecosystem by composing the components to a system once selected. Whether to start with the system or with the behavior depends on many factors. The fact whether the application would be running on a known predefined robot's hardware system is a substantial one. With the hardware system defined, typically many design decisions are already taken or at least thought in advance.

The example validates and illustrates that the independently developed parts, including behavior coordination, can be composed to a working robotics system. The system builder is able to do so without further investigation of the parts that he needs to compose.

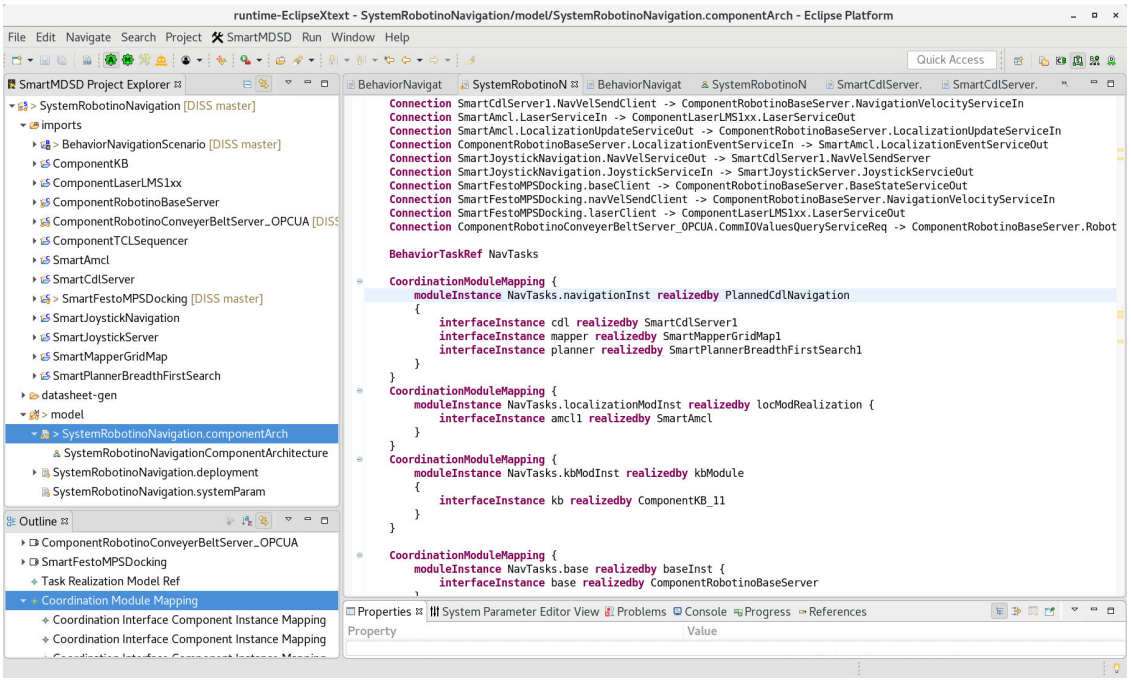

Figure 8.21.: Textual representation of the component architectural model. The excerpt shows the mapping of the abstract coordination module instances defined within the behavior project and the coordination module realizations contributed by the components and their instances.

8.2.4. Modification of Existing Robotic System - Composition of Building Blocks

This experiment demonstrates the replacement of a building block by composing a new component during system integration only. The ecosystem vision is fueled by the idea to compose systems out of readymade building blocks. The participants in the ecosystem are able to work and participate without knowing or meeting their partners. System composition needs to be possible without the need to know or change the internals of the building blocks. One of the most straightforward use cases is the exchange of individual building blocks in an already existing system. In this example, a mobile Robotino robot system using laser-based localization is altered to make use of a vision-based solution. The experiment is based on the application *Robotino Factory 4.0*.

Replacing system parts should be possible with minimal effort and without the need to change large portions of the system. In this example, the exchange requires swapping one software component only, see component architecture model for system composition in Figure 8.22. The component *SmartAmcl* is replaced by *ComponentAccerionTrition*, and the connections of the provided and required services are redone. The exchange of the localization approach reassembles a near-perfect example of a drop-in replacement. The

services for horizontal component communication are minimal and compatible, namely the services for the *baseState* and the *localization update*.

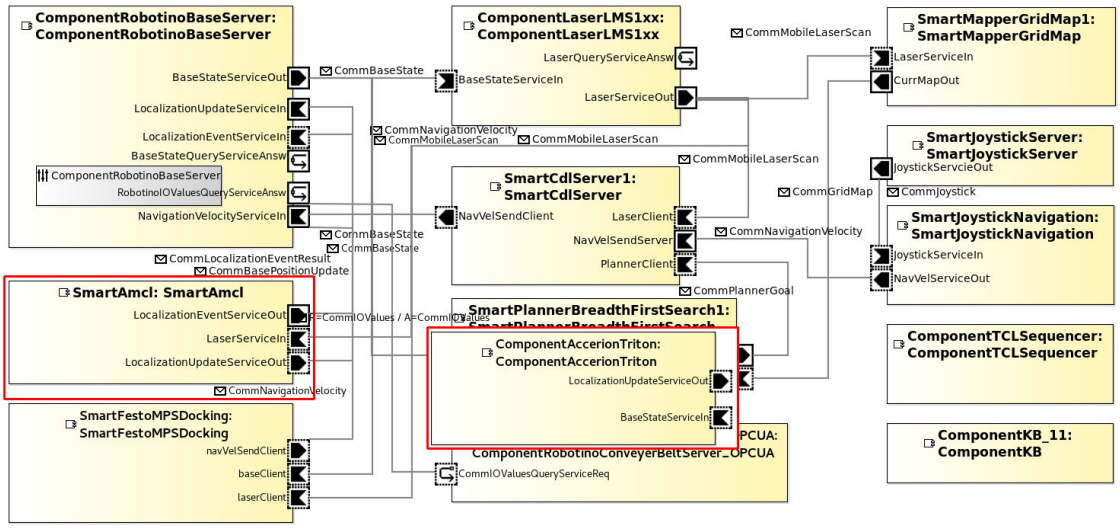

Figure 8.22.: Component architecture model of a system realizing a transportation task using a Robotino mobile robot. The red-marked components are those which are swapped, changing localization from Amcl to Triton.

For the composition of robotics behavior coordination, the only thing necessary is the adaption of coordination module realizations, as can be seen in the excerpt of the textual component system architecture model in Figure 8.23. Both components realize the same coordination module *CommLocalizationObjects.LocalizationModule*; therefore, it is only necessary to change the mapping of the realization name – the component chosen from *SmartAmcl* to *ComponentAccerionTrition* as well as the corresponding component instance names. The modeled task *transportationTask* developed by the separated role behavior developer stays unaffected. The task only relies on an instance of the coordination module type and the skills provided by it. Figure 8.24 shows the relevant part of the task model, with the highlighted lines showing the usage of the localization module.

The experiment demonstrates the composition of the building blocks, both functional as well as those for coordination. The building blocks of the example have been developed independent of each other. The system composition was done without altering any building blocks, components, skills or tasks. The involved roles were able to contribute and to receive elements from or to the ecosystem.

8.2.5. Adding Tasks to Existing Robotic Systems - Composition of Building Blocks

This experiment demonstrates the extension of an existing system with new or existing task models. The experiment uses the system described within the experiment “Com-

```
    CoordinationModuleMapping {
        moduleInstance NavTasks.localizationModInst realizedby locModRealization {
            interfaceInstance amcl1 realizedby SmartAmcl
        }
    }
```

```
    CoordinationModuleMapping {
        moduleInstance NavTasks.localizationModInst realizedby tritonlocalizationModule {
            interfaceInstance triton realizedby ComponentAccerionTriton
        }
    }
```

Figure 8.23.: Excerpt of the systems component architecture model, textual representation, illustrating the changes for coordination necessary to swap the localization approach. Only the mapping of the realization of the abstract coordination module instance introduced and used by the robotics behavior tasks need to be changed. The upper part of the figure shows the usage of the Amcl localization, the lower part the Triton one.

```
TaskRealizationModelTCL {
    AbstractCoordinationModuleInstance navigationInst coordModuleDef CommNavigationObjects.NavigationModule
    AbstractCoordinationModuleInstance kbModInst coordModuleDef CommBasicObjects.KBModule
    AbstractCoordinationModuleInstance localizationModInst coordModuleDef CommLocalizationObjects.LocalizationModule
    AbstractCoordinationModuleInstance base coordModuleDef CommNavigationObjects.MobileBaseModule
    AbstractCoordinationModuleInstance mpsInst coordModuleDef CommRobotinoObjects.MPSModule

    (define-task-block (transportationTask ?startStationID ?startBeltId ?goalStationID ?goalBeltId)
        (taskDefinition CommNavigationObjects.transportationTask)
        (rules (ruleFetchFromMpsFailed rulePushToMpsFailed ruleRobotBlocked))
        (action ( (format t "============== >>> transportationTask from:~s to:~s ~%" ?startStationID ?goalStationID)
            (let*   ((start-station (tcl-kb-query :kb-key '(is-a id) :kb-value `((is-a station)(id ?startStationID))))
                     (start-station-approach-location (get-value start-station 'approach-location))
                     (goal-station (tcl-kb-query :kb-key '(is-a id) :kb-value `((is-a station)(id ?goalStationID))))
                     (goal-station-approach-location (get-value goal-station 'approach-location)))
                (tcl-push-plan :plan `(
                    (localizationModInst.activateLocalization)
                    (navigationInst.approachLocation ,start-station-approach-location)
                    (mpsInst.mpsStationFetchFrom ?startStationID ?startBeltId)
                    (navigationInst.approachLocation ,goal-station-approach-location)
                    (mpsInst.mpsStationPushTo ?goalStationID ?goalBeltId)
                    (localizationModInst.deactivateLocalization)))))))
```

Figure 8.24.: Excerpt of the task model used within the application. The highlighted lines show the usage of the localization coordination module. The model ins unaffected by the change of the localization approach, as the model is dependent on the module definition only.

position of Building Blocks" and extends it toward the application *Collaborative Order Picking*. The system builder selects the new task models contained in a SmartMDSD Toolchain behavior project and composes them to the existing system. Based on the information provided by the model, the tooling checks the compatibility between the new tasks and the capability-delivering building blocks of the existing system.

The application uses a mobile robot, Robotino3, set up with the skills to navigate, dock, load and localize the robot. In short, it is able to autonomously transport containers between MPS stations. The newly added tasks extends the system to be used in a human-robot collaborative order-picking application, see application *Collaborative Order*

Picking 8.1.2 for further details. Therefore, the new task models require skills for a Graphical User Interface (GUI) to interact with a user as well as person following skills.

Figure 8.25 shows an expert of the new task model. In this example, the required coordination modules are not present. Thus, the skills used by the task would be undefined; the coordinating component (sequencer) would not be able to execute the skills; nor are there any components present to provide the required functionality. The red-marked lines in the figure are those introducing the incompatibility with the existing system. The line has been added by the author to highlight the interesting parts of the model. They are not present in the tooling-this is especially important to note as there cannot be errors in the task models introduced during composition. This would otherwise break the idea of the ecosystem composition since the system builder would need to change the task models to fix them. If the task is not compatible, it simply cannot be composed. Changing the task model is possible in this case, but would require the involvement of the behavior developer role (e.g., intellectual property of the role).

Once the new task is linked to the existing system, the tooling evaluates the existence of the required resources. In this case, the tooling correctly detects the missing coordination modules. The required mapping of the abstract instances defined by the task model, with the realizations provided by the components are missing, see Figure 8.26. The model editor offers a fix that adds the missing modules as well as their mapping.

The selection of matching components is currently done by the user. Further automation is possible as the interface information is modeled and present. Figure 8.27 shows the import dialog of the new components that provide the missing coordination module realization and the missing skills.

The new components are instantiated, configured, and connected to the existing system, see the graphical component architecture model Figure 8.28. In this example, the component *ComponentRealSenseV2Server*, representing an Intel realsense camera, requires a *baseState*, essentially the location of the robot. The *ComponentRealSensePersonTracker* makes use of rgbd data from the camera and provides a *CommTrackingGoal* for the navigation components to follow a person. The third component added *ComponentLogisticsWebInterface* has no horizontal connection to other components. It communicates with the coordination via the coordination interface only.

With the component instances added to the system, the missing coordination module realizations and skills are present. The system builder is able to realize the missing coordination module mappings, see Figure 8.29. In this case, there are two missing ones, one for the tracking and one for the GUI, as is directly required by the task model, visible in Figure 8.25. The third mapping is an instance required by the realization of the tracking, a camera coordination module.

The experiment demonstrates the ability of the approach to extend existing systems with new tasks and to manage the dependency required by the different building blocks. The toolchain supports the system builder in composing the tasks to the existing system. A missing building block could be composed to the system to fulfill the requirements of the new task. No changes to any building block – components, skills or tasks – were

```
TaskRealizationModelTCL {
    AbstractCoordinationModuleInstance fleetNavigation coordModuleDef CommNavigationObjects.NavigationModule
    AbstractCoordinationModuleInstance kbModInst coordModuleDef CommBasicObjects.KBModule
    AbstractCoordinationModuleInstance localizationModInst coordModuleDef CommLocalizationObjects.LocalizationModule
    AbstractCoordinationModuleInstance base coordModuleDef CommNavigationObjects.MobileBaseModule
    AbstractCoordinationModuleInstance mpsInst coordModuleDef CommRobotinoObjects.MPSModule

    AbstractCoordinationModuleInstance logisticsGui coordModuleDef DomainHMI.WebGuiModule
    AbstractCoordinationModuleInstance personFollowing coordModuleDef CommTrackingObjects.PersonTrackingModule

    (define-task-block (orderPickingJob ?dropOffStation ?dropOffBelt
                                ?emptyBoxStations ?emptyBoxBelt
                                ?parkingLocation)
        (taskDefinition CommNavigationObjects.orderPickingJob)
        (rules (ruleAbortJobErrorAckRestart))
        (action ( (format t "============== >>> orderPickingJob~%")
            (let*   ((empty-box-station (tcl-kb-query :kb-key '(is-a id) :kb-value `((is-a station)(id ?emptyBoxStations))))
                     (empty-box-station-approach-location (get-value empty-box-station 'approach-location))
                     (drop-off-station (tcl-kb-query :kb-key '(is-a id) :kb-value `((is-a station)(id ?dropOffStation))))
                     (drop-off-station-approach-location (get-value drop-off-station 'approach-location)))
                (tcl-push-plan :plan `(
                    ;; fetch empty box
                    (localizationModInst.activateLocalization)
                    (fleetNavigation.approachLocation ,empty-box-station-approach-location)
                    (mpsInst.mpsStationFetchFrom ?emptyBoxStations ?emptyBoxBelt)
                    ;; park robot
                    (fleetNavigation.approachLocation ?parkingLocation)
                    ;; wait for jobs from user
                    (logisticsGui.getUserInput => ?inputFormUser)
                    ;; evaluate user input and execute job
                    (executeNextStep ?inputFormUser)
                    ;; finalize jobs
                    (fleetNavigation.approachLocation ,drop-off-station-approach-location)
                    (mpsInst.mpsStationPushTo ?dropOffStation ?dropOffBelt)
                    (localizationModInst.deactivateLocalization)))))))

    (define-task-block (executeNextStep ?inputFormUser)
        (action ( (format t "============== >>> executeNextStep~%")
            (cond
              ((equal ?inputFormUser "pick-item")
                (tcl-push-plan :plan '(
                    (parallel (
                        (personFollowing.followPerson)
                        (logisticsGui.handOverObject))))))
              ((equal ?inputFormUser "end-job")
```

Figure 8.25.: Excerpt of the new task model composed to the system. Red-marked are those parts which make use of coordination modules and skills absent in the existing system. The red lines have been added by the author, they are not to be misinterpreted for the model errors highlighted in the toolchain. Those are present to support the user and are the result of the application of model checks. The shown task model itself is consistent, although the newly added parts to the task model will cause model errors in the system architecture model, shown in Figure 8.26, as coordination modules and skill realizations are missing.

required to compose them to the existing system. Thus, the development of the used building blocks by separated roles, as envisioned in the ecosystem, is possible.

8.2.6. Transferability of Tasks to other Robotic Systems - Composition of Behavior and System

This experiment demonstrates the transferability of task models between different robotic systems. These task models are independent of any concrete realization or functionality provided by components accessible via skills. Given the correct resources, expressed by

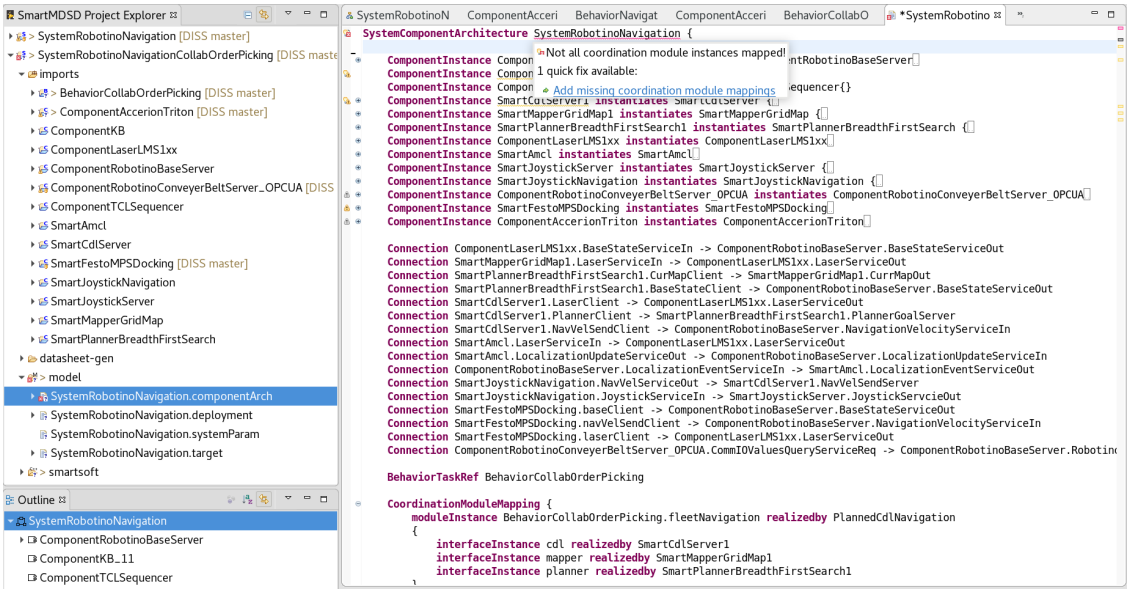

Figure 8.26.: The SmartMDSD Toolchain supports the system builder by checking the required and provided resources (coordination modules) of the system. In this example, the newly added tasks require coordination modules and skills so far not provided by the components in the system. The first line of the model shows the error as a result of the model check, due to the missing modules.

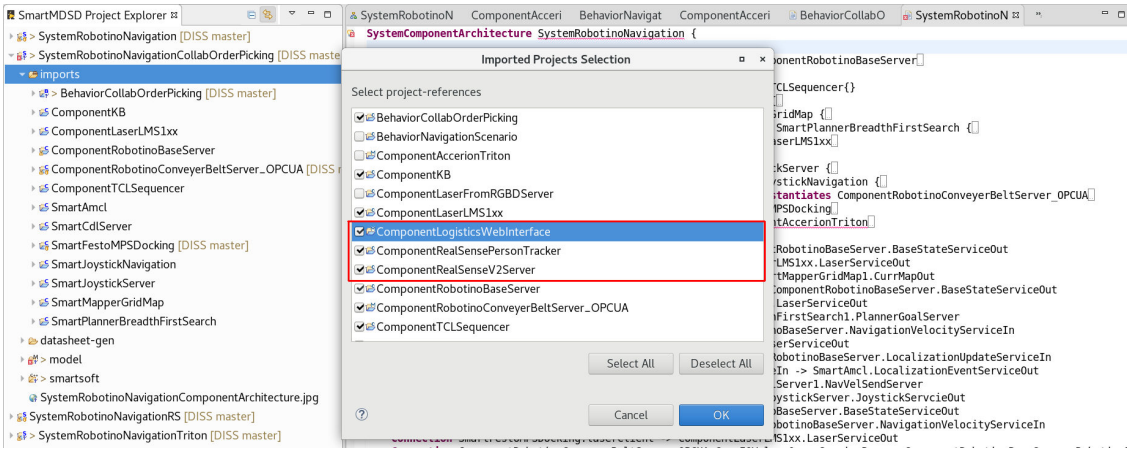

Figure 8.27.: SmartMDSD Toolchain, component import dialog, to add the missing components. The three red-marked components are those newly added to the system project.

the coordination module realizations and the skills provided by them, the task models are transferable between different robots. This is possible without the need to change the

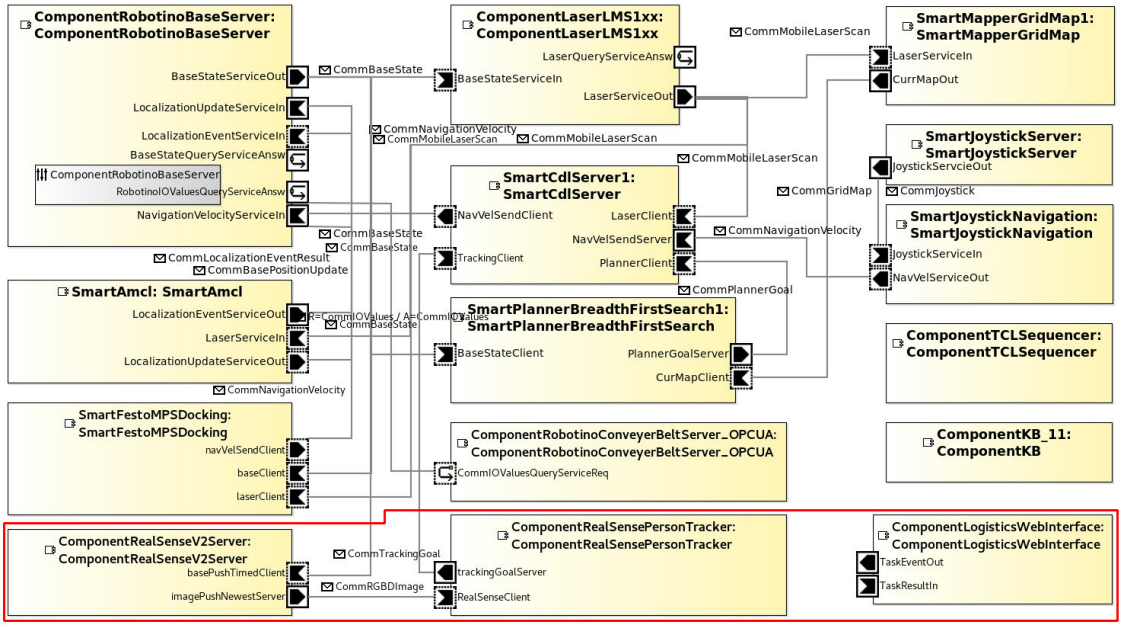

Figure 8.28.: Component architecture model of the system; the newly added components which provide the missing functionality and skills have been composed to the system (marked red).

```
BehaviorTaskRef BehaviorCollabOrderPicking

CoordinationModuleMapping {
    moduleInstance BehaviorCollabOrderPicking.logisticsGui realizedby LogistisWebInterface {
        interfaceInstance gui realizedby ComponentLogisticsWebInterface
    }
}

CoordinationModuleMapping {
    moduleInstance BehaviorCollabOrderPicking.personFollowing realizedby RealsensePersonTrackingModule {
        interfaceInstance tracker realizedby ComponentRealSensePersonTracker
    }
}

ForeignCoordinationModuleMapping {
    moduleInstance camera realizedby RealsenseCameraModule {
        interfaceInstance realsense realizedby ComponentRealSenseV2Server
    }
}

CoordinationModuleMapping {
```

Figure 8.29.: Excerpt of the textual representation of the component architecture model, showing the so far missing coordination module mappings. The missing modules, required by the new task models, are mapped to those provided by the three new components.

task models. Modifying them might not be possible as they are the intellectual property of the behavior developer, as also described in the previous experiment.

In contrast to the previous experiments, this one is not directly based on the application introduced in the previous application section. To avoid adding large component architecture models multiple times, the application will be sketched briefly in the following. The application the experiment is based on has been developed in the context of the research project ZAFH Servicerobotik (Collaborative Center for Applied Research on Service Robotics) [ZAFH]. The application realizes a robot butler scenario where the two mobile service robots Kate and Larry act as butlers and serve different beverages. Figure 8.30 shows a picture of both robots within the application, and videos of the application have been published [Heg+12] and [Lut+13].

Figure 8.30.: Service robots Kate and Larry operating as butlers in a domestic home environment.

While the previous experiments already showed some tasks being used by multiple different systems, this example exploits a wider range of capabilities and an overall level of increased complexity. The experiment focuses on the coffee-making and serving part of the application. Originally, the robot Kate is able to operate the coffee machine to fetch a cup of coffee from the kitchen. The robot fetches a cup, places it into the coffee machine, starts the brewing by pressing a button and finally, serves the coffee. An excerpt of the task model is visible in Figures 8.31 and 8.32. The model shows the used coordination module instances required to execute the task models in the first few lines.

Besides the header with the coordination module instances, the figure shows the task model realization of the task *prepareAndDeliverCoffee* and the task *prepareCoffee*. The realization uses skills to approach the kitchen and to set up the manipulation planning solution and object recognition components. It triggers the recognition, selects the cup from the detected objects, grasps the cup, and retracts the manipulator with the grasped cup. The robot approaches the place where the coffee machine is located in the kitchen. Next, the task uses the two other tasks *prepareCoffee* and *deliverOrder*. They are separated to allow for a better composition of the tasks, as well as better contingency handling (divide and conquer, not solving everything in one huge task block). Figure

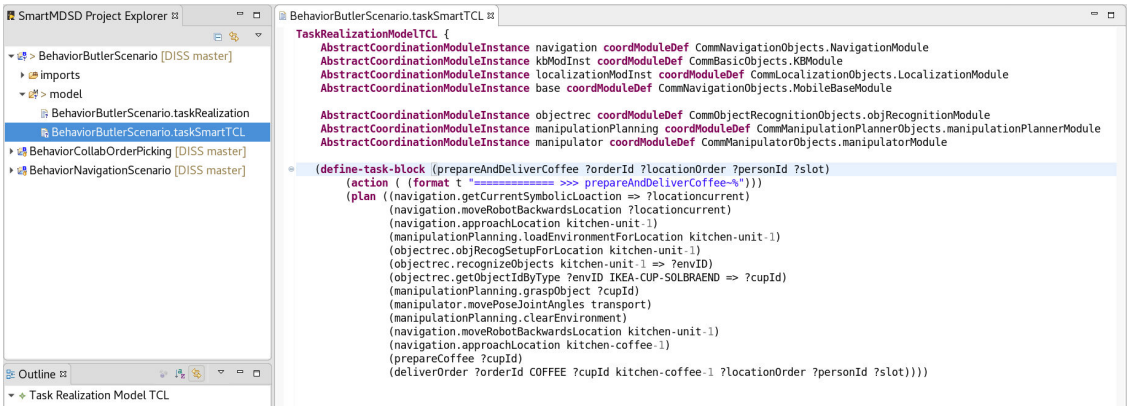

Figure 8.31.: Task model realization within the SmartMDSD Toolchain. The used coordination module instances are defined at the top. The task *prepareAndDeliverCoffee* makes use of skills and other tasks.

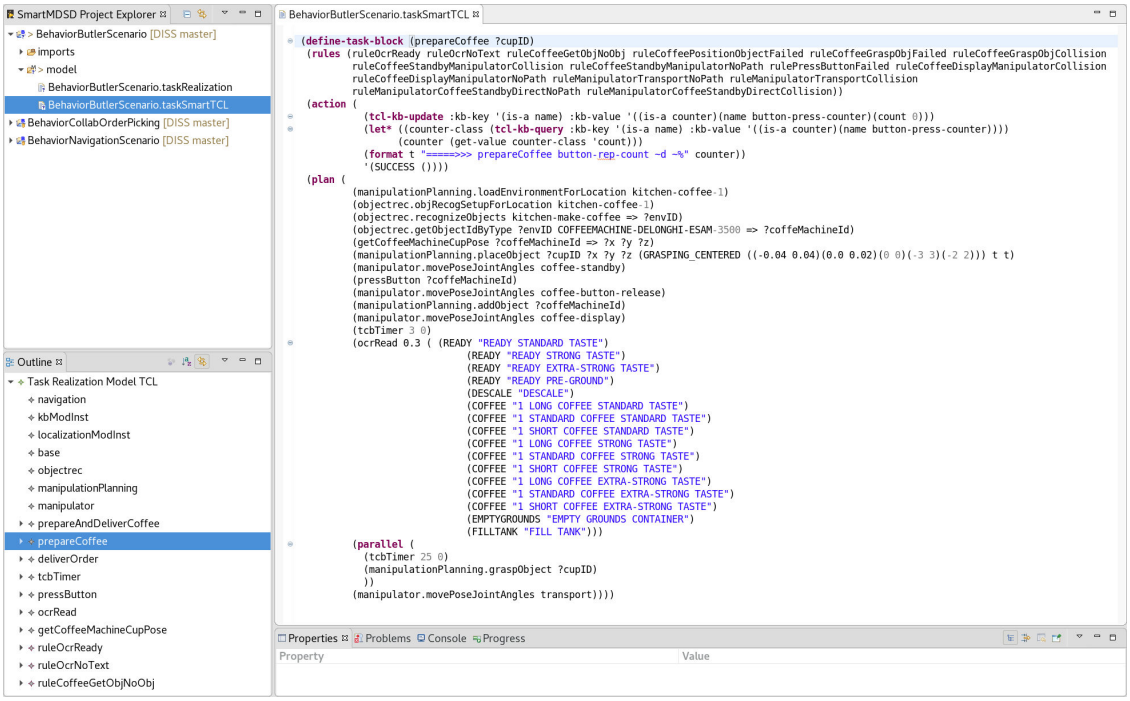

Figure 8.32.: The task model showing the realization of *prepareCoffee* task.

A.41 (appendix) shows the component architecture model used with the robot Kate. The system builder checks for compatibility of the new task with two other existing robotic systems modeled within the SmartMDSD Toolchain as system projects. The first system the check is applied to is the service robot Larry. As the robot is also equipped with mobile manipulation hardware and capabilities, the new tasks are compatible.

The component architecture model for Larry is shown in Figure A.39 (appendix). The different hardware is reflected by exchanged components, and apart from those the systems are rather similar. The system builder needs to map the coordination modules required by the task model to the realizations provided by the components of the system. With those mappings defined and some configuration data of the components configured (recognition models e.g. for the coffee machine, cups, etc.), and world knowledge defined (KB entries to know where stuff is located, etc.), the new tasks can be executed by Larry as well. The existing system parts, components, skills, and the task model are executed without the need for modification. The next system the task is checked against is the previously used Robotino-based transportation system. Figure A.40 (appendix) shows the component architecture model of the system. Obviously, the system does not provide any manipulation capabilities. Therefore, none of the coordination modules from this domain is instantiated. The check for compatibility indicates the incompatibility due to the missing coordination modules, see Figure 8.33. Therefore, the new task cannot be composed to the existing system. To make the task and the system composable, the system builder needs to add mobile manipulation components, realizing the missing coordination modules and skills or other coordination modules providing the required skills – e.g., humans handing over the freshly brewed coffee.

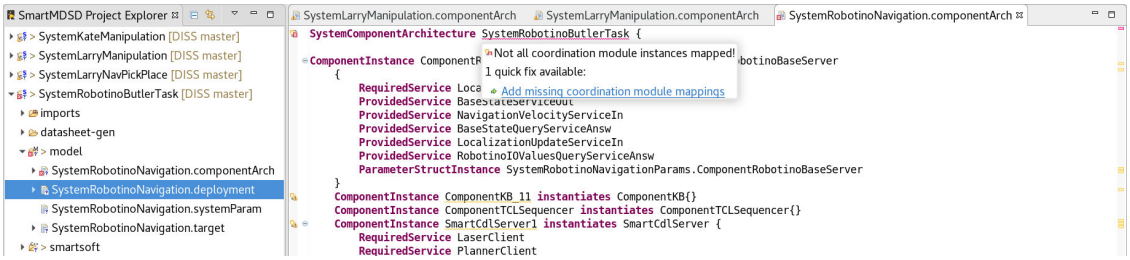

Figure 8.33.: Textual representation of the component architecture model of the Robotino-based system. The model checks detect the missing coordination module realization of the system. The new task is, therefore, not composable to the existing system.

The experiment demonstrates the reuse and transferability of task models to different robot systems. It further demonstrates compatibility-check of the task based on the coordination modules provided by the component instances of the systems. Overall, the experiment shows that the system parts developed by the different roles participating in the ecosystem can be composed without the need to know or change their internals.

8.3. Discussion

This section discusses the approach evaluated within this chapter in light of overall research goals and questions as outlined in Section 1.2:

> *How to develop robotic software systems, considering the tasking and coordination of robotic systems comprising many parts, developed by different roles in the context of a software ecosystem?*

The proposed approach contributes structures to enable the systematic development of robotics behavior coordination and robotic applications. The approach closes the gap between robotics behavior development and component-based robotics software development. It connects robotics behavior development with the development of robotic software systems in the context of a robotics business ecosystem. It introduces structures that enable the separated development of composable and coordinatable components and behavior building blocks, considering the constraints necessary to enable a robotics business ecosystem. The structures help to develop robotic applications by composing skills and tasks to robotics behaviors and composing behaviors to software components.

The structures make use of and extend those structures defined within the service-oriented, component-based SmartSoft framework. The approach presented here is not limited to SmartSoft, benefits, however, from the well-defined component model, especially from the strong encapsulation of the components. The limited sphere of influence provided by the SmartSoft components is beneficial to the presented approach as it also limits the sphere of the influence of coordination actions. This helps to enable the separated development of building blocks by different roles.

The experiments demonstrated that using the proposed approach, the development of complex robotic systems and applications, including robotics behavior coordination, is possible. The applications could be composed out of existing ecosystem building blocks developed by different roles and separated in time and space. The robotics behavior coordination can be developed relying on the defined interfaces and reusing existing skill- and task building blocks. The coordination is decoupled and no longer interwoven from the realizing functionality. This enables the systematic reuse of existing, matured robotics behavior coordination building blocks. Valuable domain knowledge is captured and made accessible to other participants of the ecosystem in this way. The introduced structures are an important contribution that enables an ecosystem of composable blocks, both components and behavior coordination (skill and task) ones.

> *1. How to enable the development of **composable** robotic behaviors separately from concrete functional building blocks?*

The approach proposes to split robotics behavior coordination into two separated parts, namely the skill level behavior models that lift the access of the functionalities to behavior level and the task level behavior models encoding the procedural knowledge,

not bound to any concrete realization. The type definitions of the skills (and the tasks) represent the interface, which is used to access functionalities to realize the task without being bound to any specific skill realization. The experiments demonstrated the realization of tasks decoupled from the functionalities provided by the components. Task level behavior models were composed to the skills of different realizations. The experiments showed the composition of tasks to different robotic systems, providing different realizations of same skill and coordination module types.

> *2. How to realize the **interface** between software parts providing functionality and software parts realizing the behavior of the robotic system in such a way that it supports the vision of composable coordination in a software ecosystem context?*

The introduced coordination interface with its typed instances, the coordination services, provides coordinating access to the components. It helps to separate the coordination logic from functional realizations within the components. With the harmonized and semantically defined coordination interface, the coordinating access to the components is realized independent of the chosen coordination approach.

The experiments shown in this section demonstrate the use of the coordination interface. The functionality wrapped within many software components was successfully coordinated to realize changing and challenging applications, which require the coordination of the system's components to achieve a working robotic system. The proposed interface and structures behind that explicate best practices out of many years of experience of how to coordinate software components in services robotics of many years. While it is impossible to foresee all eventualities, the proposed interface covers the fundamental operations required for coordination. Of course, there is still plenty of opportunities to extend the presented approach. However, care has to be taken not to dilute the semantics of the already provided elements. The proposed interface is not limited to a specific programming paradigm and does not limit the developers in realizing their components in their own way.

> *3. How to organize the development of the building blocks, functional and behavior, in a **development workflow**, supporting the separation and the collaboration of the different roles in a software ecosystem?*

The proposed workflow is based on the interfaces between the parts developed by different roles. It manages the handover of the parts. The central elements the workflow rests on are the domain model structures for coordination. The skill definition provides the interface to realize tasks not bound to specific realizations. Keeping this interface lightweight is reasonable to allow for a wide and technology-independent adoption. It does, however, capture the pivotal parts to make use of a skill. Further extensions, such as capturing non-functional aspects, are possible and can extend the presented structures. The experiments illustrate the separation of the building blocks developed by separated (time and space) roles. The applications described in this chapter have

been developed bottom-up as well as top-down, applying the relations proposed by the workflow. The presented tooling supports the involved roles to contribute parts and fetch parts from others in the ecosystem.

> *4. How to design integrated **tools** that supports the users in developing composable robotics behaviors?*

This thesis proposes patterns and structures, as well as concrete meta-models based on which integrated tools can be designed, to support the development of robotic applications including robotics behavior coordination. Further, the thesis provides a working realization of tools based on the proposed meta-models and structures. The tools have been integrated into the SmartMDSD Toolchain, available under an open-source license. The toolchain has been used to develop several robotics applications from research prototypes to market-accessible industry products. While not presenting too many implementation details regarding the tool realization, this thesis illustrates the tools being applied by the different roles. The implementation of the tools depends on the infrastructure used to realize, in this example, primarily the Eclipse modeling framework. As the framework will be subject to change, the half-life time of the tool implementation is expected to be rather small. Therefore, the implementation details are best found in the open-source project to which this thesis contributed its implementations, namely the SmartMDSD Toolchain [Foub].

8.4. Summary

This chapter presented several real-world systems and applications developed using the proposed approach. The chapter further evaluated and discussed several of the key properties the proposed approach achieved, mainly driven by the ecosystem idea this work is motivated by. The evaluations are based on experiments mostly using excerpts of the presented applications, highlighting the relevant aspects, without the need to understand and oversee the full application. The presented applications have been developed in several research and industry projects together with multiple partners. The contributing roles where able to work separately without to need to fully understand the contributions of the others. The proposed approach is further used to develop robotic systems and applications being sold as products by the industry which underpins the maturity ad achievable benefits of the approach. The applications and systems shown cover a wide range of complexity from plain and simple navigation, mixed type robotic fleets, robot-robot interaction, to mobile manipulation in open-ended environments.

Besides the evaluation, this chapter provided insights into the realized tools, namely the SmartMDSD Toolchain, and how the user benefits from it. The toolchain makes the approach usable without reading this thesis, understanding the concepts or reading hundred pages of manual. This chapter showed concrete examples of models and code that the users in their different roles encounter and develop.

The next chapter concludes this thesis and provides insights for possible future work.

9. Conclusion and Future Work

Robotics behavior development is currently most often done by handcrafting each application from scratch. Robotic applications are developed with a centralized integration phase towards the end of the development. The functional parts are fitted together, examining their interfaces and adapting them to their counterparts for a specific application. Once the functional parts of the system are realized and connected, the robotics behavior coordination is developed from scratch. In some cases reusing existing behaviors via copy and paste, adapting them to the needs of the current application. Thus little or no systematic reuse of developed robotics behaviors is realized.

This thesis closes the gap between robotics behavior development and component-based robotics software development. It showed how to connected robotics behavior development with the development of robotic software systems in the context of a robotics business ecosystem. The approach proposed within this thesis provides a step-change in the way robotic behaviors are developed. It enables the development of robotics behaviors and applications by reusing composable building blocks developed separately from the applications in which they are used. The structures the approach introduced, implemented, and successfully demonstrated enable the contribution of knowledge and solutions captured in building blocks by separated roles. Other participants in a robotics business ecosystem can use those building blocks without the need to get in touch with the contributors or to fully understand the building blocks' internals. This enables the presented ecosystem vision, including the development of applications that use robotics behavior coordination.

This thesis proposed the introduction of skills and tasks as two separated abstraction levels of robotics behaviors. This allows to separate the roles involved in behavior development. The technology provider, aka. the component builder is able to contribute skills, lifting the level of their contribution to directly applicable robotics behavior building blocks. The definition of interface types for both tasks and skills, as domain structures of the ecosystem, decouples the realization of independent task level behaviors from any realizing component. Thereby, the reuse and composition of behaviors to new systems is now possible. The expertise contributed by the behavior developer, e.g., procedural domain knowledge, can now be captured such that other ecosystem participants can make use of it. This drastically speeds up the development of robotic behaviors and applications, especially for existing systems used for new applications, which results in lower efforts, lower costs, and shorter time to market for robotic applications.

One abstraction level deeper, this thesis proposes an essential prerequisite for the

above-mentioned skills: a harmonized and well-defined component coordination interface. Without a solid foundation for how to connect skills with components and the functionalities implemented within the components, the composition of the skills with components would not be possible. In consequence, the skills themselves would not be possible, as each skill would need to be realized specialized to the realization of the component. This would break the envisioned ecosystem idea, as the "as-is" composition of coordinatable building blocks would not be possible.

The presented work rests on the stable foundation of CBSE and SOA, which are the baseline for the functional building blocks. The realization of the presented approach is based on SmartSoft and contributes to the SmartSoft world. The approach utilizes model-driven techniques to realize the step-change in robotics behavior development, from handcrafted and build from scratch to systematically composed. MDSD provides the means to introduce structures in meta-models and the tools to enable the user to realize their contributions following those structures, with little effort.

Without the proper tool support, the structures and concepts proposed would be very hard to use. Therefore, the approach has been realized within the SmartMDSD Toolchain. Several DSLs have been developed to support the users with implementing the models and the derived artifacts (e.g., code generation). Based on the harmonized and semantically defined coordination interface and the connection to the skills, coordination approaches can use and coordinate the functionalities realized within the components. The implemented tools support the user with the full power of existing MDSD tools, such as model checks, auto-completion, or syntax highlights.

The core structures of this thesis have been contributed to the RobMoSys (EU Horizon 2020) body of knowledge, see for example [Lut+18b]. This European-level project underlines the relevance of a model-driven and composition-oriented approach also addressed by this thesis. The possibility to make use of and to combine different robotics behavior coordination approaches (e.g., behavior tree, task net, state machines), based on the structures introduced by this thesis, has been demonstrated in the context of RobMoSys (c.f. [Pro19b]).

The applicability of the approach has been demonstrated in several real-world scenarios and applications. The complexity of the scenarios covers a range from simple navigation in open-ended environments, the coordination of heterogeneous robot fleets to mobile manipulation in kitchens. The "maturity level" of the scenarios and applications ranges from academic prototypes developed with partners and demonstrated in many research projects to industry-sold products.

A possible direction of future work could be the development of more easy-to-use coordination approaches for "technical end-users". They could make use of GUIs to adopt the tasking of a robotic system themselves. While this is already possible for closed systems, e.g., UR manipulators, for a system composed of parts from different manufacturers, there is still work to do. Many current robotics behavior coordination approaches are either very simple but tailored to a single product or complex and challenging to use. The approaches could either utilize the skills or the coordination

interface of the components proposed in this thesis to connect to the functionalities implemented within the components. The reuse of existing knowledge and realizations should simplify their connection to existing functional building blocks.

Another possible direction could be the extension of software engineering for robotics behaviors. A further extension of the ability to manage resources of robots and their parts would increase the level of composability of the building blocks for both the components as well as the behavior blocks. The proposed coordination modules provide the means to express required and provided resources on robotic behavior level. Apart from a first example, namely the communications services used by coordination modules, this work does, however, not provide further contributions to the management of resources on the robotics behavior level. The role of the robotics behavior developer has to take care of conflicting use of resources when using skills provided by the technology provider. Contributions to this topic could help to improve the composability of the building blocks, and possibly further reduce the effort required to realize robotic applications. The expression of the skill and coordination module interfaces as domain knowledge could serve as a basis to annotate further resources for behavior coordination. The use of MDSD to express those critical common structures should facilitate the connection of future contributions in this area.

Another interesting direction of further work, especially in the context of ecosystems, could be a run-time resolution of required and provided building blocks and skills. With the proposed approach the development of robotic behavior decoupled from any functional building block is possible. The proposed approach realizes the resolution of the skills required by the behavior tasks and the skill realizations provided by the components, in a manual step during the development of the system. While this is the most straightforward approach, a run-time resolution of this binding would be valuable. Especially pure software building blocks and skills (e.g. recognition or planning approaches) could be selected with valuable run-time context knowledge. Different ecosystem marketplaces could provide alternative competing building blocks for run-time deployment. The proposed approach does by no means limit this binding to be static or bound during the "development time" of the system. On the contrary, the realization of the approach already provides a run-time robot self-model with explicated skill requirements generated from the behavior task models, accessible via a knowledgebase component. This could serve as a good base even if no MDSD approach is utilized.

List of Acronyms

A.I.	Artificial Intelligence
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
KB	Knowledge Base
MDSD	Model-Driven Software Development
OCL	Object Constraint Language
OMG	Object Management Group
OPC-UA	OPC Unified Architecture
REST	Representational State Transfer
ROS	Robot Operating System
SLAM	Simultaneous Localization and Mapping
SME	Small and Medium-Sized Enterprise
SOA	Service-Oriented Architecture
SoaML	Service Oriented Architecture Modeling Language
SOAP	Simple Object Access Protocol
SPL	Software Product Line

SysML Systems Modeling Language

UML Unified Modeling Language

A. Appendix

A.1. Skills - Integration and Run-Time

This section introduces some important integration- and runtime-relevant aspects to the concepts of skills and coordination modules. The following details two aspects, both concerned with the execution of skills and the executing component. For the tasking of robotic systems, the modeled skills need to be executed. To enable the skills to make use of the services contained and explicated within the component coordination interface, the skills need to be executed within a software component. This could either be a single component (e.g. the sequencer) or multiple distributed components. Both approaches need to offer an interface to the task-level robotics behavior models to make use of the skills for the tasking of the robots. This interface can also be used to separate different realization approaches of skill modeling and task-behavior modeling. The skill realizations can, for example, make use of simpler state charts approaches, while on the task level more advanced and tailored approaches such as behavior trees (overview by Colledanchise and Ögren in [CÖ18]) can be used.

Besides the interface to make use of the skills at the task level, the skill execution components needs to make use of the concrete component coordination interfaces, following the different types of coordination services. The use of concrete instances of coordination interfaces by the component executing the skills binds these components to those coordination interface instances and types. Keeping the skill execution components independent of the component coordination interfaces is, however, mandatory in any scenario reaching beyond lab prototypes. Those components would otherwise be under constant change and need to be able to support any kind of the coordination interface type, including multiple instances of the interfaces. A concept of how to deal with this is clearly required and is presented in the second part of this subsection.

A.1.1. Skill - Run-Time Interface

Besides the design-time view on the skills, used to model skills and their interface, skills are used during runtime. During runtime different task level behavior coordination approaches can make use of existing skills. The components executing the skill model blocks can, therefore, offer a runtime skill interface. This enables the use of existing skills by different coordination approaches without the need to realize skills (not directly accessing the components, lifting the level of abstraction) over and over again to match a single used coordination approach. To do so, the skill definition, as a formal definition

of the interface of a skill, needs to be made accessible at runtime. A protocol for communication with the runtime skill interface needs to be established. From a bigger perspective, this allows for the composition of different robotics behavior coordination approaches at the level of individual skills. The other possible approach is to use one behavior coordination approach to realize both tasks and skills, with no runtime interface being required in this case.

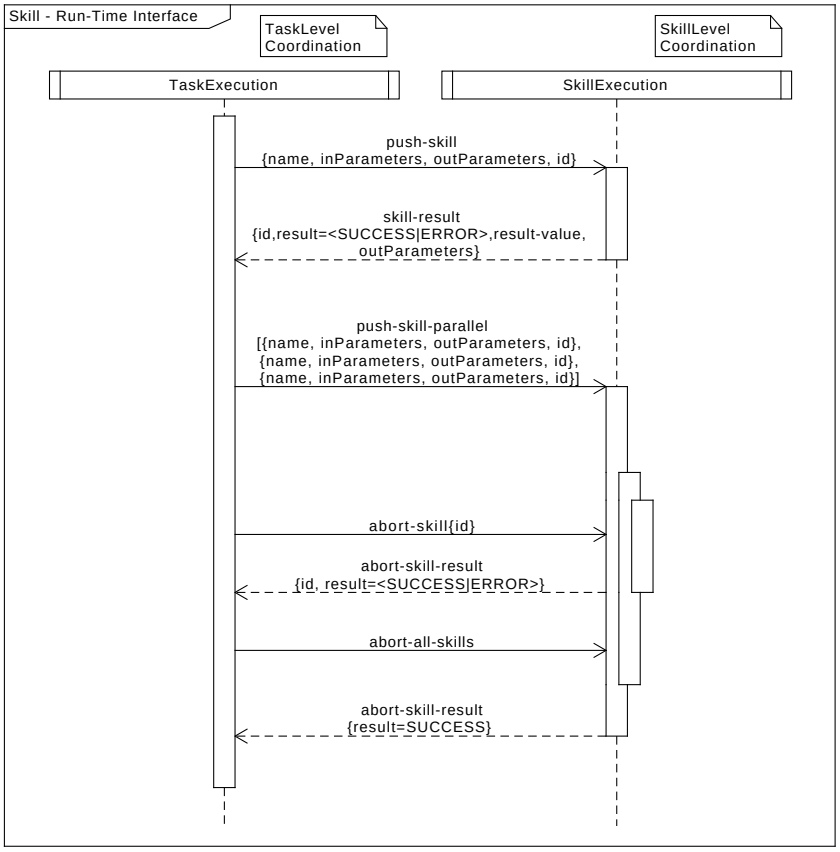

Figure A.1.: Run-Time skill execution interface to make use of skills from different task coordination approaches.

The interface for calling the skill is mainly used to execute skills and receive the result as modeled by the skill definitions. The interface, therefore, features foremost two calls, *push-skill* and *abort-skill*, with Figure A.1 detailing and illustrating the communication semantics.

The *push-skill* call is used to push a skill for execution. Following the information modeled by the skill definition, it must contain the parameters in and out of the skill definition. The out parameters define a variable name to bind a value after the execution of the skill. This variable can be used by subsequent skills as input parameter. Therefore,

the input parameters need a flag to indicate whether the input parameter is a variable to be bound at runtime or a value. In addition to the skill definition, *push-skill* contains an id to allow for the mapping of the response to the pushed skills. Once the execution of the skill is finished, the skill executor returns a *skill-result* message. The message contains the result values as they are defined by the skill definition and the id of the push skill to map the result. The results could either be “SUCCESS" or “ERROR" accompanied by a further detailed result string. The out parameters of the skill definition are also returned by the *skill-result* message. In case multiple skills are pushed, the skills are executed in the order in which they are pushed. The *push-skill-parallel* call can be used to push a list of skills to be executed in parallel, in contrast to the default push skill call execution in sequence. The *skill-result* messages are returned per executed skill, similar to the push-skill. The user interface is detailed in Table A.1.

push-skill	push a skill to be executed
name	name of the skill, as defined within the skill definition model
inParameters	inParameters, either values or variables to be bound during execution
outParameters	outParameters, variables which are bound past the execution of the skill
id	id defined by the task execution side
push-skill-parallel	push a list of skills to be executed parallel to each other
<list> skills	list of skills to be executed in parallel, each skill as in push-skill
skill-result	response send once a skill has been executed
id	id of the finished skill
result	Boolean result of the skill execution either SUCCESS or ERROR
result-value	detailed result value, as defined by the skill definition
outParameter	the out parameters of the finished skill block

Table A.1.: The push-skill calls and resulted messages, used to push skills to be executed and to receive the results of a skill execution.

The *abort-skill* call is used to abort a specific skill either running or queued. The skill to abort is identified by the id handed over when pushing the skill. The call returns an *abort-skill-result* containing the result of the action – either SUCCESS or ERROR – as well as the id of the skill to abort. The abort call might return ERROR in case the skill to abort is not running or not queued to run, or for other implementation-specific reasons. The *abort-all-skills* call is used to abort all currently running or queued skills. The call returns an *abort-skill-result* message with a result value of SUCCESS in any case. The implementation has to abort all blocks, and a failure to do so would result in an

undefined state at the task level side, as it would be unclear which skills are aborted and which are not. The user interface is detailed in Table A.2.

abort-skill	abort a specific skill
id	id of the skill to be aborted, as defined in the push-skill calls
abort-all-skills	abort all pushed skills
abort-skill-result	response send when a skill has been aborted
result	Boolean result of the skill abortion either SUCCESS or ERROR

Table A.2.: The abort-skill calls and result message, used to abort already pushed skills.

As simple as this interface is, it enables the combination of different robotics behavior coordination approaches. It allows for the reuse of the skills contributed by technology developers. To demonstrate the feasibility of the runtime skill behavior interface and the composition of different robotics behavior task coordination approaches with the skills, the interface has been implemented using a SmartSoft component offering different robotics framework independent communication mechanisms such as a message passing library ZeroMQ [ZMQ] as interface. The protocol for calling the skills has been implemented using JSON [Cro] and has been published within the RobMoSys Wiki [Lut+18b]. This implementation has been used by the Integrated Technical Project (ITP) MOOD2Be of EU H2020 RobMoSys project to illustrate task-level composition using behavior trees [Faca] together with the graphical editor Groot [Facb] using the run-time skill interface. The skills themselves have been implemented using an extended version of the robotics behavior coordination approach SmartTCL [SS10]. The approach has been demonstrated in a small "Intralogistics Industry 4.0" pilot, published as video [Fac+18], also presented in the experiments Chapter A.4.1.

A.1.2. Coordination Modules and Coordination Interfaces - Plug-ins

The second runtime relevant aspect is centered around the component or components executing the skills, the coordinating component. Following the idea of composable building blocks and the vision of a robotics business ecosystem, where different roles collaborate while being separated in space and time, the components executing the skill blocks needs to deal with the emerging challenges. Using the concept of coordination interfaces, forming a unified and explicated access for coordination to the components, the coordinating components needs to know and make use of specific instances of a coordination interface (typed by the coordination service definition). This is true for approaches using typed interfaces as well.
The one challenge that emerges immediately is the dependency between the skill executing components and the used coordination interfaces toward the components. The coordination interface between the coordinated component and the coordinating components is used to coordinate software components by executing the skills. The

content of a concrete coordination interface is dependent on the component to coordinate and the skills; it reflects the consolidated domain knowledge, for details see the Chapter 6 on coordination interface. Concrete coordination interface instances, therefore, differ from each other – e.g., the one for coordinating a navigation planner component from the one for an object-recognition component. The coordination interface instances, typed by the coordination service, use different services on the side of the coordinated components. Therefore, the coordinating components needs to be able to use those instances as well. As the set of possible interfaces is unlimited, the coordinating components cannot be realized in a static manner containing all possible coordination interface types, this is only possible in a contained environment where the coordinating components is extended every time a new coordination interface type is defined. Multiple instances of the same coordination interface type add to the problem. Figure A.2 illustrates the problem drastically: a coordinating component containing interfaces to many services for coordination. Opposed to the component shown in Figure A.4 featuring only the generic services of a coordinating software component. The structure behind this problem is illustrated in Figure A.5 with a coordination master interface contained within the coordinating component. The coordinating component is design time bound to the typed coordination interfaces, see Figure A.5.

To overcome this problem, the coordinating components needs to be independent (design time) of the of the used coordination services. The dependency between them is required at runtime but not at design time. The coordination module as a container for both skill level behavior models and the coordination services should be fully decoupled from the coordinating components during design time. In general, the interface between the information provided by the coordinated components and the coordinating components needs to be dealt with. One can, however, shift where and by whom the interface is realized. The following presents the two most important possible approaches separated by the roles and also the place where the interface could be realized.

The first possible approach is to make the interface generic toward the coordinated components, avoiding the use of specific services. This approach makes the coordinating components design and runtime independent of specific interfaces or services. This approach shifts the interface problem to the role of the component developer. This role has to realize the transformation of the specific information into the generic format used by the coordinating components. This is what the proposed coordination interface in Chapter 6 does for the configuration and activation of the component; it does, however, not provided unified use of the communication services used between coordinated components. Applying the unification to those services also introduces the drawback of the duplicated services provided by the component to other components. For example, a localization component offers its location result, namely the location of the robot, to other components such as a path planning. If the same information is required for coordination, the developer of the localization component would need to develop a conversion to provide the information to the generic coordination interface.

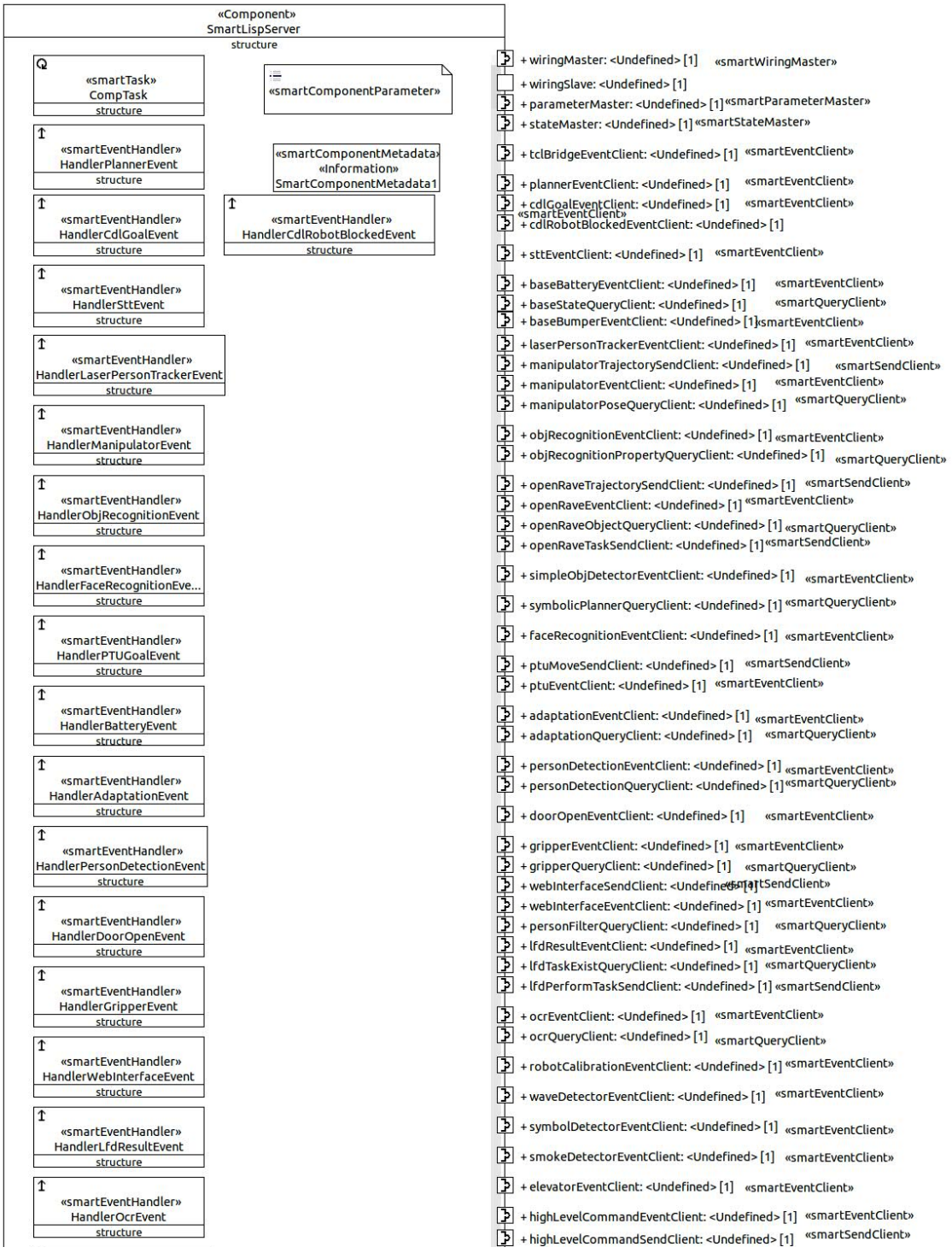

Figure A.2.: Part1/2: Coordinating component illustrating the problem of the unlimited number of different coordination services, as opposed to a generic set of services.

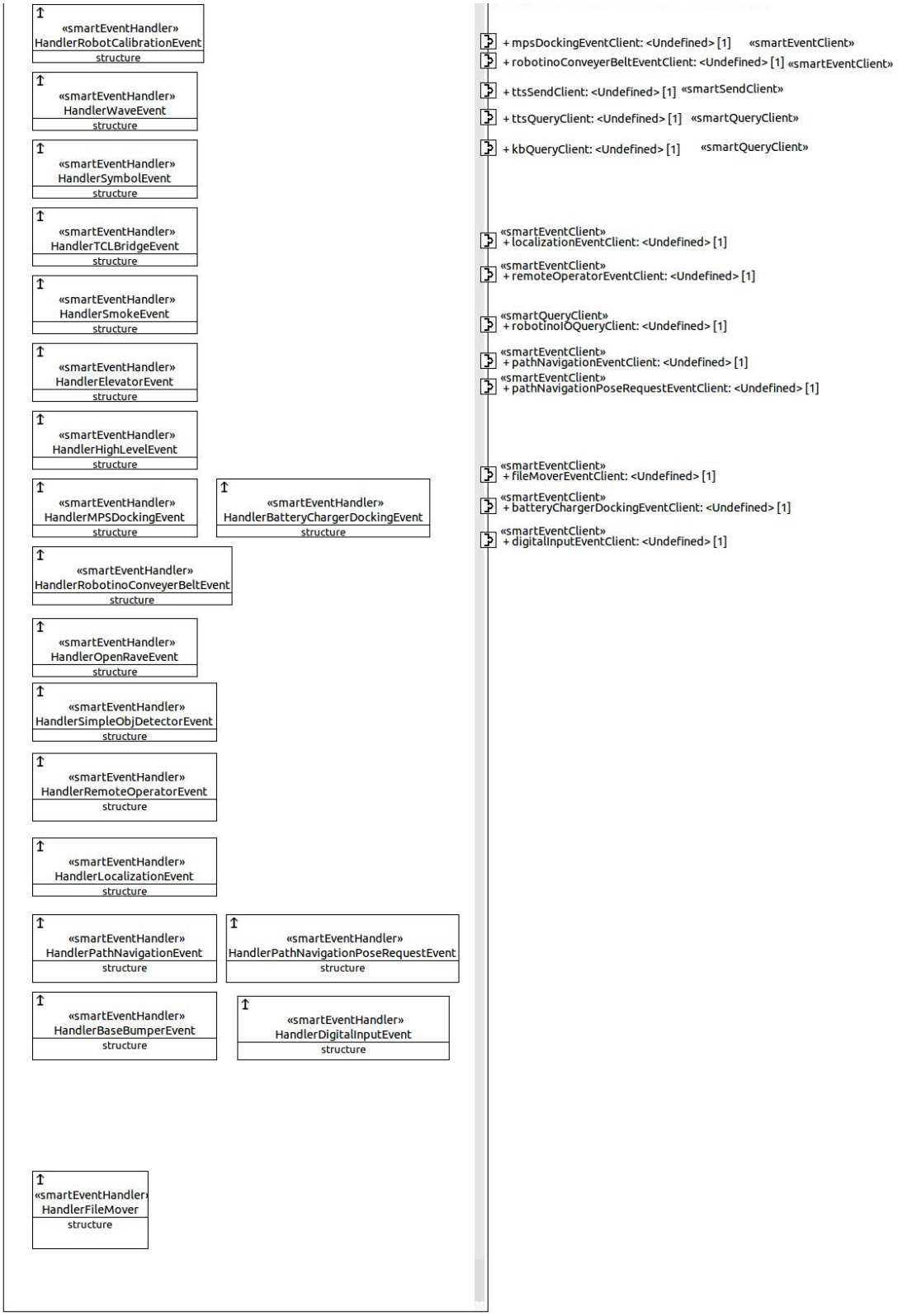

Figure A.3.: Part2/2: Coordinating component illustrating the problem of the unlimited number of different coordination services, as opposed to a generic set of services.

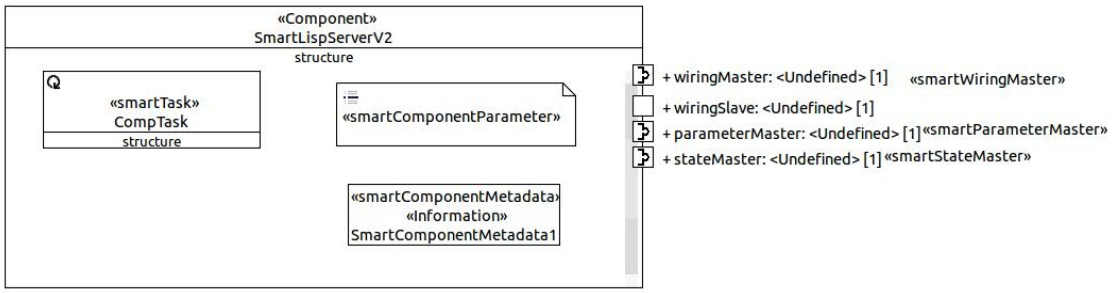

Figure A.4.: Coordinating component with a generic coordination set of services, at design time, opposed to Figure A.2.

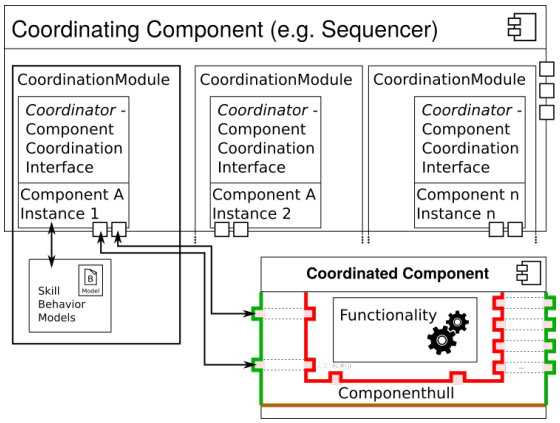

Figure A.5.: Problem, design time bound coordination interfaces (typed) to a coordinating component.

This conversion needs to be done by the role of the technology developer for every component.

The second possible approach is to shift the dependency to runtime, using a plug-in concept. Same as the other approach mentioned above, this one enables a coordinating component to be design-time generic. With this approach, the role and the place where and by whom the interface is located can be shifted. From a composition perspective, the realization of this interface close to the coordination interface definition is reasonable. In this way, the interface has to be realized once per coordination interface type only, as opposed to multiple times, if done by the technology developer within the component as mentioned above. Figure A.6 illustrates the structure of the proposed approach. The coordination interface together with the skills is fully separated from the coordinating components at design time. As a downside, this approach has to deal with the added complexity of the runtime plug-in mechanism.

Each coordination interface plug-in follows a uniform structure, as shown in the Unified Modeling Language (UML) class diagram in Figure A.7. In contrast to the other

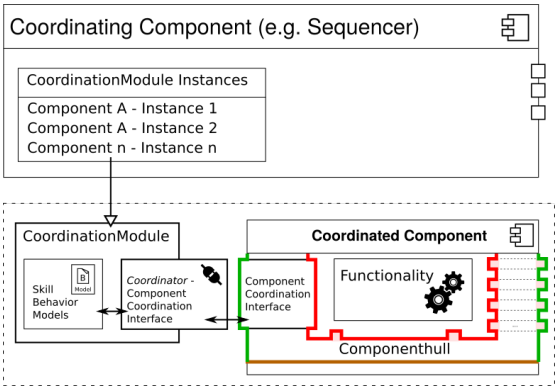

Figure A.6.: Coordination modules logically grouped with the coordinated component and runtime decoupled from the coordinating component.

elements, this structure is not presented as Ecore meta-model since the conceptional structures of the interface are already defined by the coordination module and the coordination interface meta-models. The additional structures concerning the plug-ins on the coordinator side are, therefore, presented in UML as those concepts are not role-related. It is possible in theory to fully automate the interface mappings. In practice, however, this interface might introduce technology mappings, combining different implementation technologies such as programming languages. As the effort here is spent once per coordination interface type (following a coordination service definition) only, full automation is not the most pressing need.

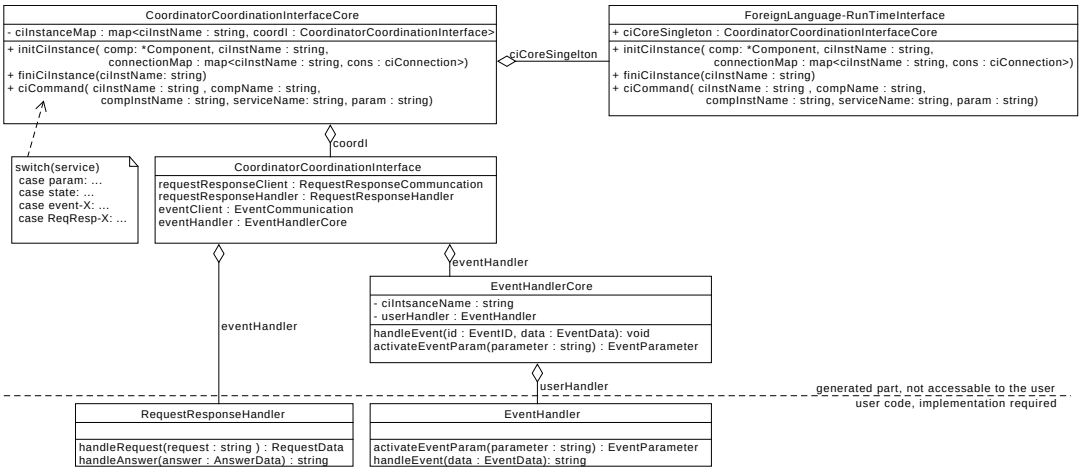

Figure A.7.: UML class diagram modeling the plug-in structure of the coordinator side of a coordination interface.

Each plug-in realizes the access to one specific coordination interface type (coordi-

nation service definition). If multiple instances of the same type are used, the plug-in needs to manage those instances. Most of the plug-in content will result in generated code not to be touched or seen by a user of any role if a MDE approach is applied. The role of the domain expert does, however, need to implement the interface for the reused communication services, which affects the pattern for request-response and event. In Figure A.7, the two bottom most classes – the *RequestResponseHandler* and the *EventHandler* – are being accessible to the user. The user has to implement the interface mapping toward a technology- independent mapping such as a simple XML or JSON structure. The *RequestResponseHandler* class, therefore, features two callback methods to perform the mapping, see Table A.3. The second class *EventHandler* is responsible for the mapping of the event communication and contains two callback methods to do so, see Table A.4.

RequestResponseHandler	user class to implement request response communication mapping
handleRequest(request : string) : RequestData	the method converts the coordinator message to the message type used by the communication service
handleAnswer(answer : AnswerData) : string	this method converts the communication service message into the format for the coordinator

Table A.3.: RequestResponseHandler class of the coordination interface, to be implemented by the user.

EventHandler	user class to implement event communication mapping
activateEventParam(parameter : string) : EventParameter	the method converts the coordinator message to the message type used by the communication service
handleEvent(data : EventeData) : string	this method converts the communication service message into the format for the coordinator

Table A.4.: EventHandler class of the coordination interface, to be implemented by the user.

Besides the user visible parts, the plug-in's most interesting part is the main class *CoordinatorCoordinationInterfaceCore*. The class features three most important methods:

The *iniCiInstance* method to create a new instance of the coordination module. This is especially important if the coordination interface type is used multiple times. All instances are stored within the *ciIntanceMap*. To create a new instance, the methods feature the following parameter: a reference to the component to attach the communication endpoints to, a name of the new instance, and the connection information about where

to connect the endpoints to.

The *finiCiInstance* method to destroy an existing coordination interface instance identified by the name of the instance.

The *ciCommand* method used to interact with the services contained within the coordination interface. This method uses the following parameters – the name of the coordination interface instance, the name of the component type to which the service is attached to, the instance name of the component, the service name to use, and the parameter that contains the actual data to communicate. The method needs to perform the switching to access the correct part of the coordination interface. The necessary information to do so is created during the instantiation and initialization of the coordination interface instance. All interface-relevant parts are stored inside the *CoordinatorCoordinationInterface* class. The class contains the clients and the handler classes. To enable easier access from other programming languages, the plug-in features a *ForeignLanguageRunTimeInterface*. It contains a singleton of the *CoordinatorCoordinationInterfaceCore* class and maps the access methods of the core class to a plain function. For many languages, this allows for an easier mapping, hiding class, and object structures to the interface. The interaction among a coordinating component, the runtime coordination module, and the coordinated component is illustrated in Figure A.8.

The infrastructure to use the plug-ins within the coordinating component side needs to manage the plug-ins to load and unload them. It needs to further keep track of the instances and map them to the coordination modules and contained skills. As this part is rather straightforward, it does not contribute important structures or role-relevant contributions and is to a large extent specific to the approach taken for skill realization.

A.2. Component Coordination Interface - Framework and Component API

This section details the component coordination interface, proposed in Chapter 6. The section contributes the component developer API of the coordination interface, as well as further realization remarks. The section follows the structure of the coordination interface, which is divided into six parts.

A.2.1. Configuration

In contrast to classical communication patterns, as for example, the SmartSoft communication patterns [Sch04], the here presented parameter pattern describes not only the communication part but also the usage for component and system coordination as well. It describes the communication semantics as well as the data intended to communicate and the structure this data is communicated within. Figure A.9 illustrate this communication semantics.

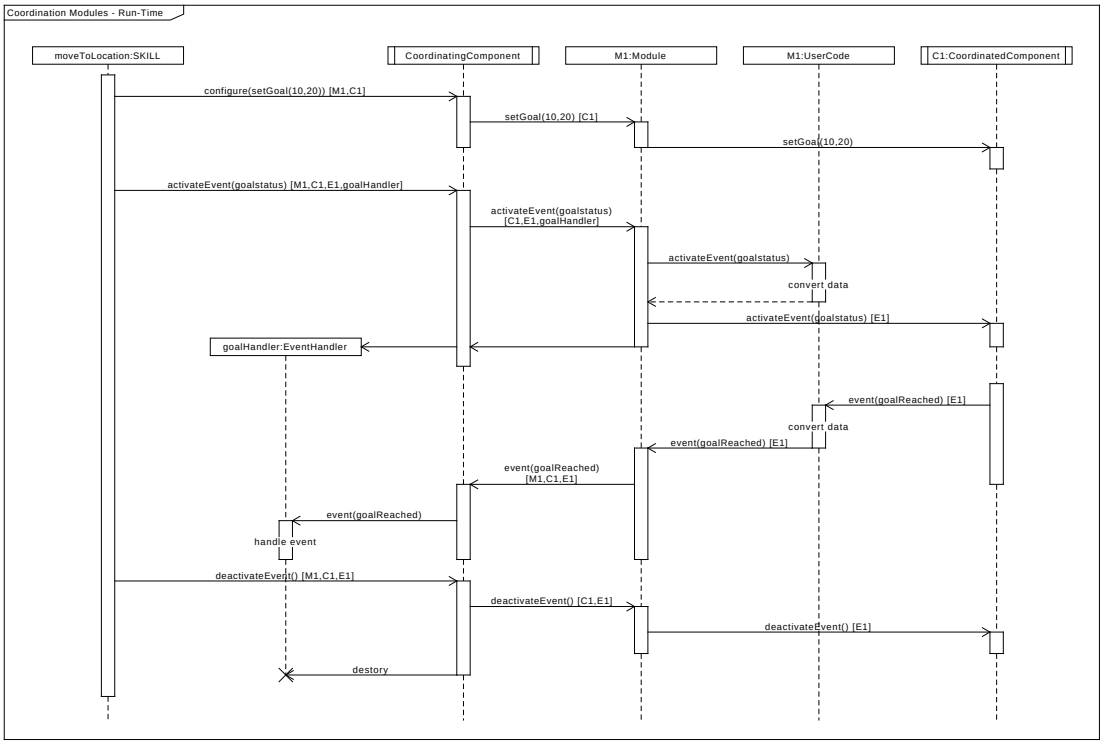

Figure A.8.: Sequence diagram showing the interaction among the coordinating component, the used runtime coordination module, and the coordinated component.

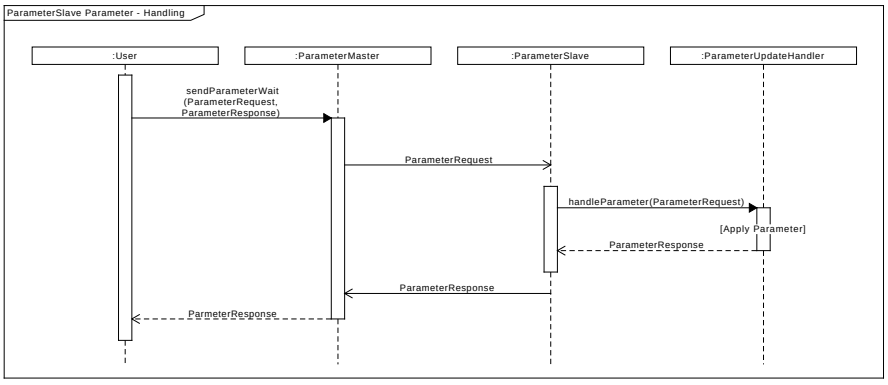

Figure A.9.: Sequence diagram showing the typical use of the pattern.

The run-time coordination part of the configuration pattern uses a list of key-value pairs to configure software components, as defined by the pattern's meta-models. A single parameter consists of a name (tag) and an optional list of name-value pairs that

also feature a name. The tag of a single parameter identifies and describes the semantics of a parameter. The optional list of name values pairs is the values a parameter can feature. From the communication semantics part, the parameter pattern implements a two-way communication, where the main communication data flow is from the *parameter master* to the *parameter slave*. The *parameter master* sends parameter requests, usable via a member method interface, to the *parameter slave* that features a handler-based interface. The *parameter slave* responds to the parameter request past the application of the sent parameter request by sending a parameter response back to the *parameter master*, one *parameter master* can communicate with many *parameter slaves*. The application of the parameters on the *parameter slave* side is synchronous and uses a handler upcall. The component cannot change its configuration by itself, as this would lead to an unknown configuration state by the coordination and thereby break the control architecture.

The *sendParameterWait* member method call is blocking, and no new parameter can be sent from the *prameter master* until the send parameter is applied on the *parameter slave* side. This is important to ensure that the send parameters are applied before the component (typically the sequencer) on the *parameter master* side continues the execution. An asynchronous non-blocking parameter application would, in some cases, require the usage of a second communication channel (e.g., events) to indicate the application of the components' parametrization to ensure the completed application of the component parameters. The *parameter slave* should not block the handler upcall for a longer period of time (timespan in context of coordinator execution) as it blocks the execution of the coordinating component. The underlying communication layer needs to ensure that the parameters are applied at the *parameter slave* (user handler upcalls) in the order the *parameter master* sends them. Depending on the implementation on the user side, the application of the parameters is user-dependent. In the context of an object recognition example, the configuration of a list of objects to recognize and the order of removing and adding new items to a list is crucial. The response sent back from the *parameter slave* to the *parameter master* is used to provide two different functionalities. First, it is used for synchronization to ensure the application of component parameters. Second, the response is used to approve the correctness of the send parameter in the sense that the master and the slave use the same parameters at run-time. The parameter pattern with its two-way communication can be implemented using a request-response communication mechanism, for SmartSoft [Sch04] it can be implemented using the SmartSoft *query* communication pattern, hiding some parts of the *query* communications options to enforce the here described semantics and user interface.

Master Side

The *parameter master* offers a simple and single method based interface for synchronous communication with the *parameter slave*. The interface of the master side is listed in Figure A.10. The main method is *sendParameterWait*, which takes a object of type CommParameterRequest and returns by reference a CommParameterResponse object.

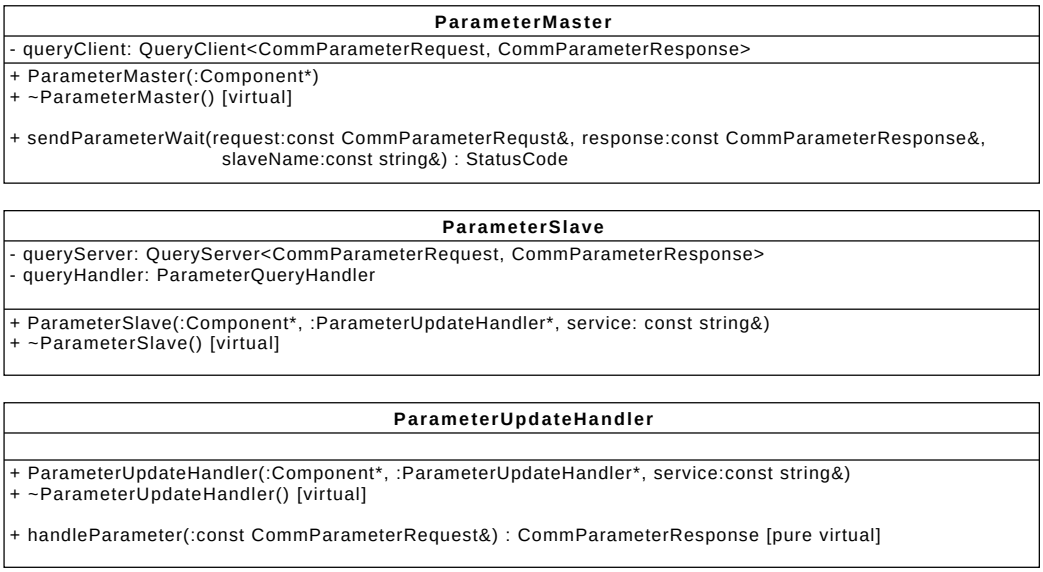

Figure A.10.: Parameter pattern, user interface classes.

The return value of the method states different run-time errors as listed in Table A.5.

sendParameterWait	send parameter to slave
ok	parameter send ok, the response object is valid
unknown component	component or service not reachable or compatible
communication error	communication error, response is not valid
error	Something went completely wrong, response is not valid

Table A.5.: The *sendParameterWait* member function, used to send a parameter from a *ParameterMaster* to a *ParameterSlave*.

Slave Side

The *parameter slave* offers a handler-based interface, shown in Figure A.10. Every CommParameterRequest is passed to the handler and results in an upcall providing the communicated response object. The handler is realized using an abstract class that enforces an implementation of the *handleParameter* method by the user. The communicated parameter is passed to the user via the method parameter. The return value of the method is of type *CommParameterResponse* and encourages the user to evaluate or apply the parameter within this handler upcall. To keep the communication with the master synchronous, the *parameter slave* does not offer an explicated method to answer a parameter request. The processing of the parameter in the *ParameterUpdateHandler*

should be limited to a short period of time, in the context of the *parameter master*, since the master is blocked meanwhile, no blocking or long taking calls should be used.

Parameter Communication Data

The data communicated between the *parameter master* and the *parameter slave* is the parameter itself, as defined by the parameter model. For component parametrization, there is no need to communicate large or complex data structures, as the parameter is at the interface between symbolic and subsymbolic communication. No complex data structures (e.g., sensor values such as images) are passed on to the symbolic sequencing layer and should therefore also not be used for component parametrization. If a component parametrization would need large or complex data structures, it would be a clear indication that the intended interface to the symbolic sequencing layer does not clearly separate the concerns. Configuring a component with a sensor value, e.g., a camera image does not automatically mean to introduce a camera image to the sequencing layer and using the parameter to configure the component directly with the image since this would violate the principle of separation of concerns. Nevertheless, configuring a component with a camera image could be solved by identifying it using a symbolic ID known to both sequencing and coordinated component. This way, the camera image would stick to the subsymbolic skill component layer, and the component to be configured would then be parametrized with the ID (know by sequencer and components) of the image and would fetch the image from another component or source (e.g., file) providing it.

Following the parameter definition model, the communicated data is rather simple and well structured. Applying the principle of freedom of choice, only basic data types can be usable. No composite data types consisting of primitive data types are allowed.

The data communicated for the parameterization is contained by a tailored, simple, and reusable data structure. The usage of a generic data structure for all parameters simplifies the usage of the component parameter pattern. No component-specific data structures need to be generated and implemented. Even if the pattern is used within the context of MDSD, where most or even all code could be generated, the definition of a specific data structure for each component parametrization would introduce more complexity than necessary. The interface on both sides is simpler if it uses the same data structure for all component parameters. Keeping the interface lightweight helps to enforce the clear separation of the architectural layers (skills and components).

To allow for a flexible and expendable data structure *name-value* representation is used. The parameters that are send to the skill components are send as a list of *name-value* pairs, Figure A.11 illustrates the data structure.

A parameter request contains a single parameter only, which consists of a list of *name-value* pairs. Each request contains a special *name-value* pair that identifies the parameter. The name is bound to “Slot” and the value is fixed to be of string type,

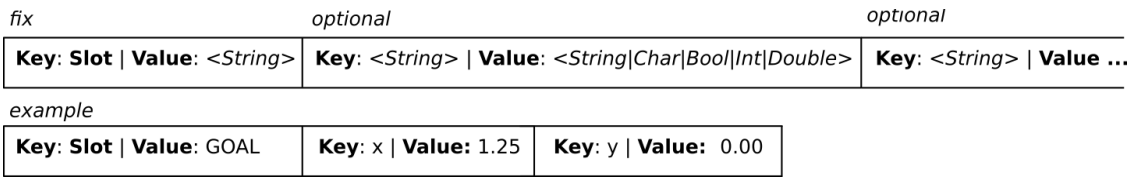

Figure A.11.: Parameter communication data structure, consisting of name (key) value pairs.

representing the name of the parameter, including optional namespaces that the user could define. Following the meta-model, the parameters are contained in namespaces. The value part of a component parameter is limited to primitive data types: integer numbers, floating-point numbers, characters, strings, enum, and booleans. During the communication of the parameter from the master to the slave, the communicated data is transformed to a unified transportation format and handed over to the underlying communication middleware. The parameter slave needs to know the data format of the received parameter to fetch the data from the received parameter in the correct format and data type. The identification of the parameter using its first name-value pair "slot" is therefore important.

The response data structure send back from the *parameter slave* to the *parameter master* is a simple enum and can not be extended by the user, nor can it be used to send back any user data. The response channel is meant for synchronization and for run-time validation only. The two different communication values are listed in Table A.6. The response could either be *ok* or *invalid*, the former means that the parameter is successfully applied on the *parameter slave* side. *Invalid* means that the *paramerter slave* was not able to deal with the received parameter at run-time. This could either result from an unknown parameter (name) or a conversion failure while extracting the optional name-value pairs.

Slave Side Parameter Handling - Parameter Extension

The so far described interface for configuration of the component provides the baseline for consistent communication and configuration using individual parameters of the components. The pattern can now be extended by mechanisms that support the component developer on the *parameter-slave* side to consistently use multiple component run-time parameters. To ensure persistent configuration states on the client-side, the *parameter patter*, therefore, is extended to uses a lightweight commit protocol on the user and data level. Once the configuration of a component is finished, a specially tagged *COMMIT* parameter is used to atomically apply the new configuration on the *parameter slave* side.

In robotics, many components execute algorithmic tasks cyclically in multiple threads, for example performing obstacle avoidance within a component every 100ms. The persistent handling of the components' parameter throughout the whole component is a nontrivial and repetitive task every component developer has to deal with. Extending

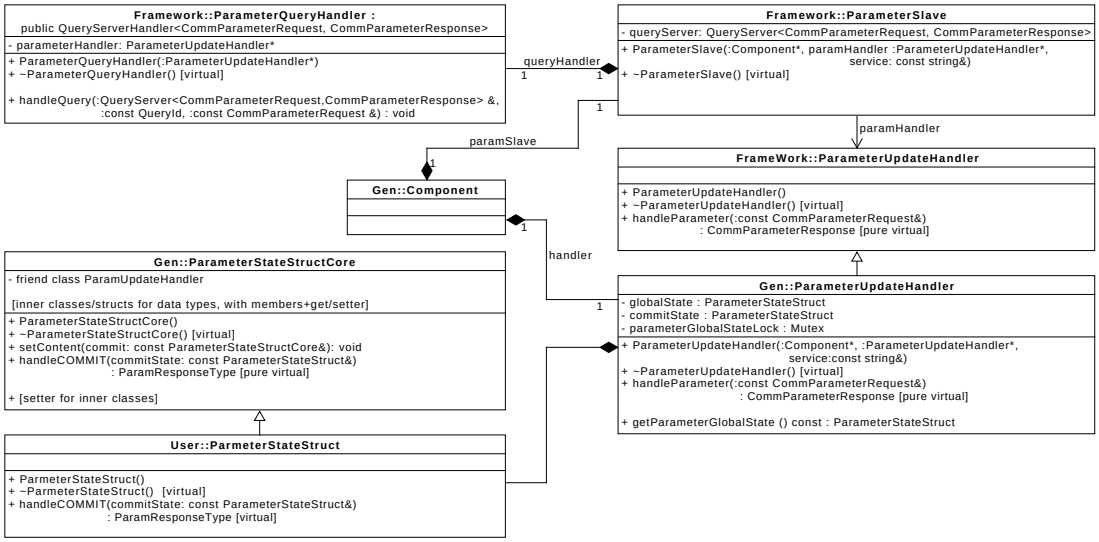

Figure A.12.: UML class diagram showing the extension structure on top of the so far presented pattern.

the parameter pattern to tackle this problem, in combination with the above-mentioned usage of multiple parameters, supports the component developer in implementing a consistent component coordination interface.

To allow for consistent usage of the components' parameter, the plain *ParameterUpdateHandler* is extended by a construct of classes that deal with the consistent handling of the parameters. Figure A.12 illustrates the extension to the plain parameter handler, described above. In the context of MDSD, the whole constructs can be generated based on the parameter model. Implementing the full pattern extension described below for simple components from scratch might be overkill for many components. However, for larger components featuring multiple threads and handlers, the consistent handling of components' parameter pays off quickly.

The most important problem to solve is the challenge of using a consistent set of parameters during transient configuration states. A set of parameters may depend on each other and only once all parameters are set the whole configuration is valid again. During reconfiguration of a component, transient states in the component configuration can occur. To overcome this problem, a lightweight commit protocol is established. The underlying communication middleware needs to assure the persistent communication of the data, thus the commit protocol is limited to the user data layer.

To enable easy access to the component configuration, the set of component parameters is encapsulated in simple wrapper classes. The set of all those classes, each representing the data transmitted via a parameter, represents the configuration of the component, including run-time and start-up parameters. The sequence of the component run-time parameter handling on the slave side is shown in Figure A.13. The *Parame-*

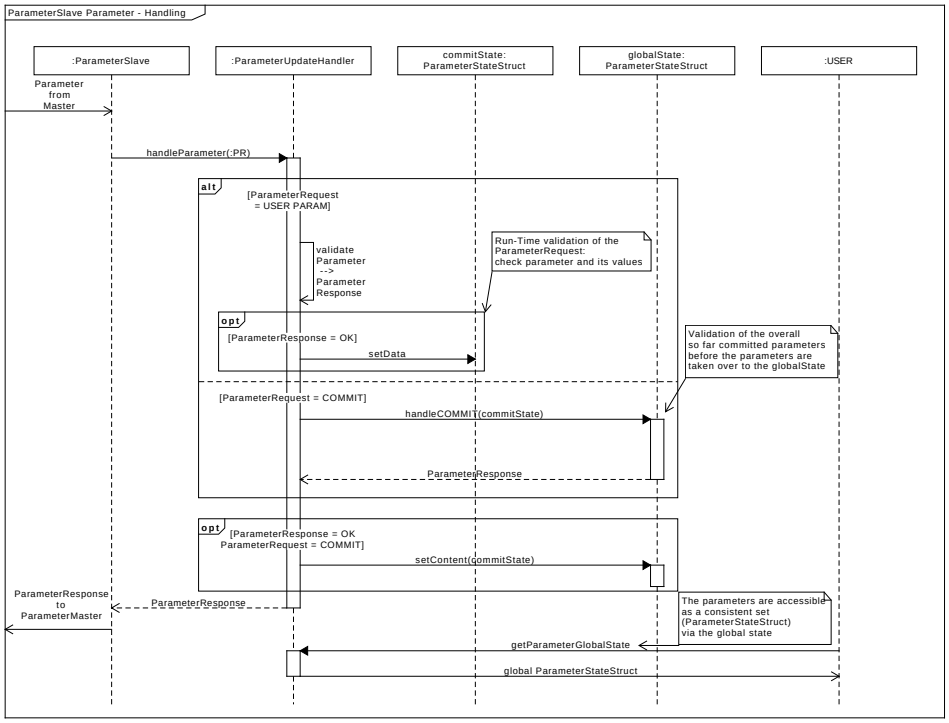

Figure A.13.: UML sequence diagram showing the interaction among the classes of the extended pattern.

terUpdateHandler holds two instances of this wrapper set, a so-called *commitState* and a *globalState*. The *commitState* is only accessible from within the *ParameterUpdateHandler* and holds the transient configuration state of the component during the configuration. All parameters sent to the *parameterSlave* will be evaluated, see the previous section, and in a successful case, they will be applied to *commitState*, containing the wrapper class with the corresponding data members.

The *globalState* is accessible throughout the whole component and holds the most recent valid component configuration. Once a configuration action from the master side is finished, and a consistent set of configurations has been communicated, the master sends a special "COMMIT" parameter to signal the slave the end of the configuration and to apply the parameters within the component, copying the *commitState* to the *globalState*. Before applying the parameters to the global state, a handler upcall *handlerCommit* is executed to validate the consistency of the overall configuration within the *commitState*. As individual parameters might depend on each other, the validation of the overall set enables to check them against each other before applying them to the *global state*. The handler decides if the *commitState* is applied to the *globalState* and therefore applying the component's configuration. The handler's return value is of type *CommParameterResponse* and is sent back to the master side.

CommParameterResponse	parameter response type
ok	parameter applied ok
invalid	invalid parameter received

Table A.6.: Communication data send as response to each parameter.

The second most important issue to deal with is the consistent usage of the components' configuration within cyclic tasks. The same problem as with the consistent set of parameters for the whole component holds true for an executed task. The individual parameter values may depend on each other. The maximum allowed translational velocity might depend on the maximum allowed rotational velocity. During the execution of an algorithm, the persistent configuration of such parameter values might be necessary. As threads in components run concurrently and the components' parameter uses an own upcall handler, the configuration of the component might be performed during the execution of the algorithms. No assumption about when a component configuration might occur should be made. To keep the set of configurations stable during the execution or at least during the critical phase of execution of a thread, the component developer can fetch a local copy of the most recent components' configuration. For cyclically executed tasks, this is typically done at the start of every cycle. However, the usage of this construct is up to the user and depends on the algorithmic details and the implementation of the component.

The pattern supports the component developer in copying a valid set of the component configuration using the same wrapper classes used for the *commitState* and the *globalState*. The *globalState*, accessible throughout the component, is applied to a user-defined fixed copy, e.g, named *localState*. During the critical phase of the execution, the access to the components' parameter can be done in the same way as it could be done using the global state, except that the configuration values are fixed. The sequence of actions and an example is shown in the sequence diagram in Figure A.14.

A.2.2. Activation

This section describes the realization of relevant parts of the pattern, including the user view on the usage of the pattern. The described structures are realized on the code layers (component and framework). UML models are used to define the structures on this level. The here presented structures follow those defined by the model and meta-model layer.

The realization is split into two parts, the activation of the one-shot and the one of the cyclic activities, due to their different interface on the coordinated component side. Both types feature the same communication direction, sending data from a coordinating component to the coordinated component. In contrast to classical communication patterns, for example, the SmartSoft communication patterns [Sch04], the here presented

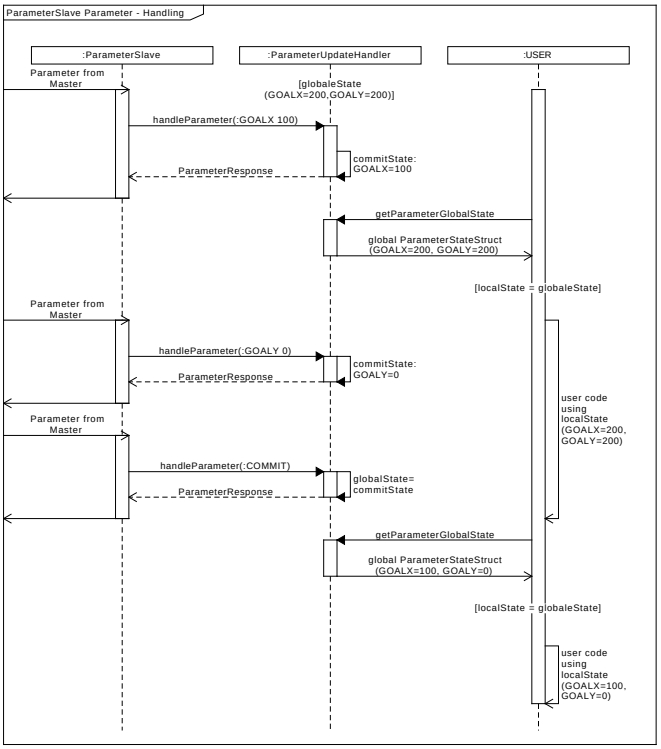

Figure A.14.: Sequence diagram showing an example how the pattern is used to configure a component, using multiple dependent parameters.

activation pattern not only describes the communication part but extends to the usage for component and system coordination. The continuous activation part of the pattern is based on the SmartSoft state pattern. The here presented pattern tailors the state pattern to the coordination use case.

The communication data for the one-shot trigger activation part follows the same type of meta-model as the one used for component run-time configuration and can therefore use the same generic communication data construct. Thereby, the same key value construct is used; this time, the special key “slot” is used to identify which trigger is addressed.

Master Side

The master side of the two activation types offers a simple and single method bases interface for synchronous communication with the slave side. The main methods are the *sendTriggerWait*, which takes an object of the type CommTriggerRequest and returns by reference a CommTriggerResponse data object. The *sendStateWait* method takes the mode to set as single argument. The return value for both cases which explicates different run-time errors is listed in Table A.7. The response messages send back from

the slave as a result of the trigger, and state activation call are listed in Table A.8 and A.9. The execution semantics defined in the model, following the meta-model, is encoded on the slave side (component level, code generated).

sendTriggerWait/sendStateWait	send trigger/state activation to slave
ok	trigger/state send ok, response object is valid
unknown component	component not reachable, response not valid
communication error	communication error, response not valid
error	Something went wrong, response not valid

Table A.7.: The sendTriggerWait and sendStateWait member function, used to send a trigger- or state-activation from a master to a slave.

CommTriggerResponse	trigger response type
ok	trigger applied ok
invalid	invalid trigger received
declined	trigger activation declined async trigger still running

Table A.8.: Communication data send as response to each trigger.

CommStateResponse	state response type
ok	state change ok
invalid	invalid state received

Table A.9.: Communication data send as response to each state change request.

Slave Side

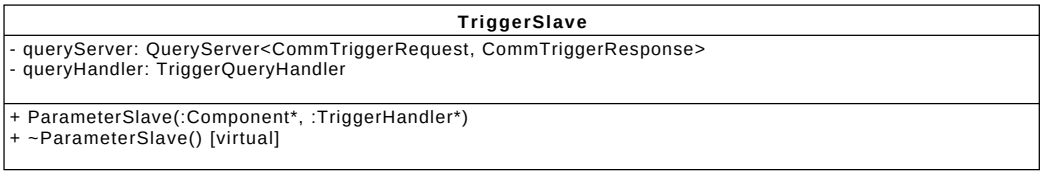

Figure A.15.: One-Shot Activation, trigger pattern user interface classes.

The slave side of the two activation types are rather different and motivate the separation in their realization. The slave side interface of the trigger coordination pattern on the framework level is illustrated in Figure A.15. The *trigger slave* offers a

handler-based interface, every CommTriggerRequest is passed to the handler and results in an up call giving access to the communicated data. The handler is realized using an abstract class that enforces an implementation of the *handleTrigger* method by the user. The communicated trigger, the type, and the optional additional attributes are passed to the user via the method parameter. The return value of the method encourages the user to evaluate the trigger call within this handler. The so far presented interface represents the framework side part. On the slave side, these interfaces could be used to realize components using the trigger pattern for one-shot activation. Therefore, the user would need to implement the selection of the different trigger types and the attributes on their own, which is possible but repetitive and error-prone implementation. Using the pattern with this simple interface, the execution semantics would be required to be fixed, as the semantics would otherwise depend on the user implementation and would therefore break the encapsulation. To avoid this, the pattern is extended using MDSD and a model to code transformation to generate a more easy-to-use interface. Figure A.16 shows the UML class diagram of the extended pattern. The *trigger slave* on the framework side in extension with a generated trigger handle class *TriggerHandlerCore* wraps the pattern. The generated user class *TriggerHandler* represents the only interface the user is presented with. This class features individual up calls for each trigger the component is using, as is defined by the component activation model. The selection of the individual triggers and their attributes is realized in the generated *TriggerHandlerCore* class. This class also realized the modeled execution semantics of the individual parameters. The class contains threads to actively call asynchronously executed triggers. A single thread for the realization of the pattern is not sufficient as it would not be possible to run multiple asynchronous triggers in parallel. The activation of an already running asynchronous trigger is not allowed and needs to be rejected and signaled by the *TriggerResponse* object. This ensures the execution order of the trigger even if sync and async triggers are mixed. It is, however, possible to implement this part of the pattern differently using multiple queues to ensure a consistent execution order. In the context the pattern is used, the coordination of the software components by the skill behavior models, in usage within the orchestration cycle, the case of multiple asynchronous activations (while running) is very unusual. When using MDE tools to model the skills, a model checker could be used to detect those cases as well. Weighting the overhead and the added complexity with the gained flexibility, the here presented realization is more reasonable. Figure A.17 shows a sequence diagram, illustrating the usage and realization of the pattern on the slave side, including the generated code. The difference between the two execution semantics of the one-shot activation is illustrated there. For asynchronous execution, the upcall from the trigger handler is forwarded to the related thread assigned to the trigger.

The realization of the activation of the continuous running activities can be done using the SmartSoft state pattern as described by Schlegel in [Sch04]. Not all features offered by SmartSotf state pattern Schlegel proposed are necessary to realize the coordination interface part for activation. The following introduces only those structures

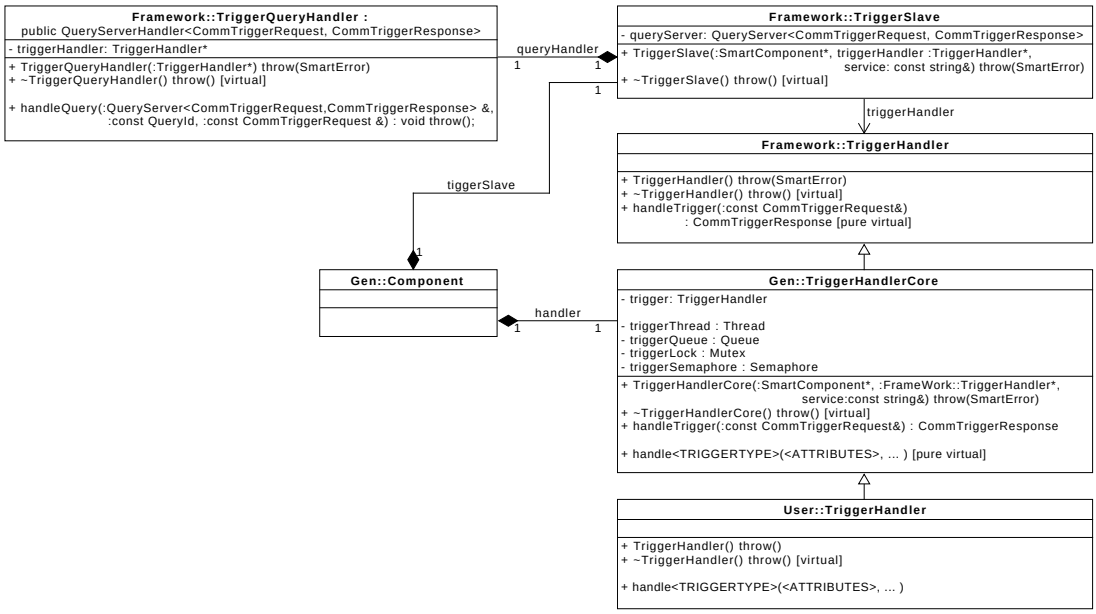

Figure A.16.: UML class diagram showing the full (to component extended) structure of the pattern.

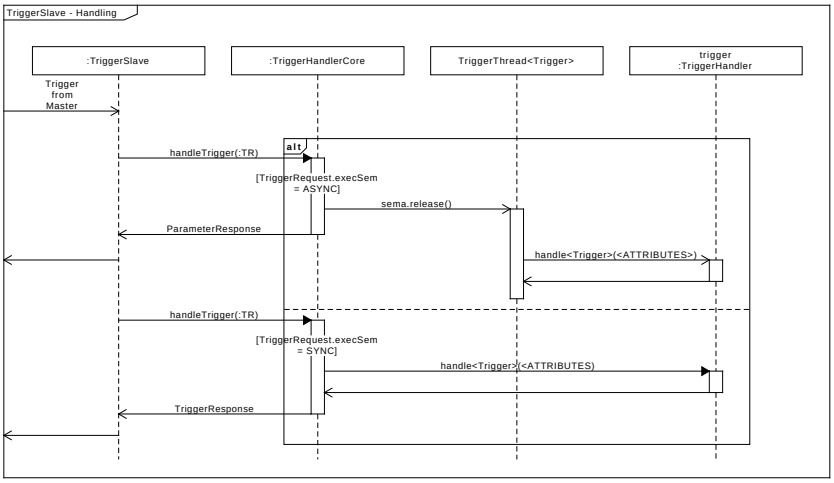

Figure A.17.: UML sequence diagram showing the interaction among the classes of the pattern.

and semantics of the pattern which are required. The additional parts can be used as well and are aligned with the here presented approach. Further information about the SmartSoft state pattern can be found in [SLS11].

In contrast to the trigger, the state pattern offers an acquire and release semantics

with no handler up call, Figure A.18. This enables flexible usage of the pattern with respect to the implementation of the user content. Dependent on the realization of the functionalities, the states could either be directly bound to the activities, or the states can be used directly from the user code. The sequence diagram A.19 illustrates both use cases of the patterns slave side.

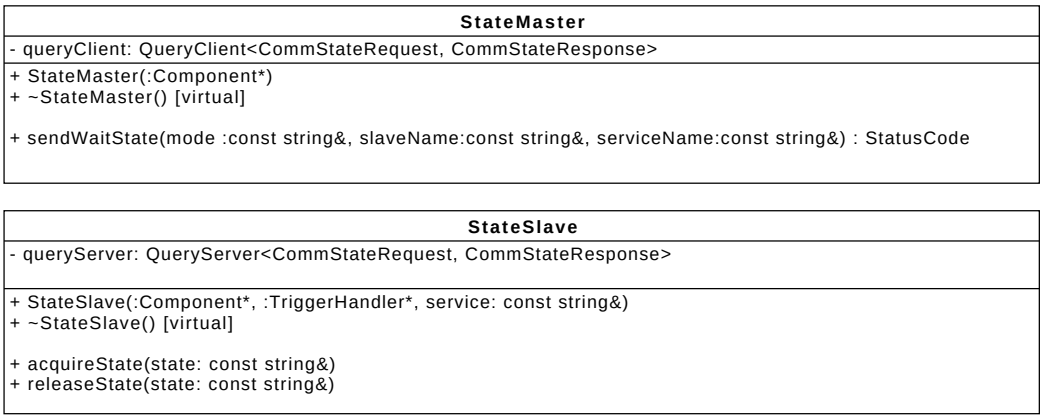

Figure A.18.: Continuous Activation, state pattern user interface classes.

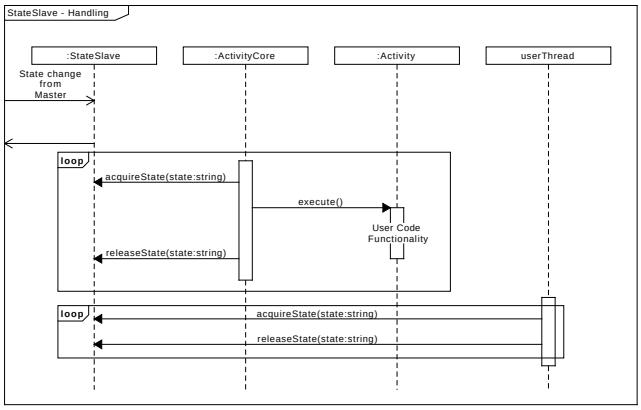

Figure A.19.: UML sequence diagram showing the interaction among the classes of the pattern.

A.2.3. Connection

The realization of the pattern follows the structures of the SmartSoft dynamic wiring pattern as described by Schlegel in [Sch04]. No further structures than those described

by Schlegel are required. The following shows only a short overview presenting the user interface.

Master Side

Regarding the connection pattern, the more interesting part is on the master side of the pattern. The user interface for those parts is illustrated in Figure A.20. The two main methods are *connect* and *disconnect*. Both methods required the name of the instantiated component service and component to perform the action on. The connect method additionally requires the name of the new component and service to connect the service too.

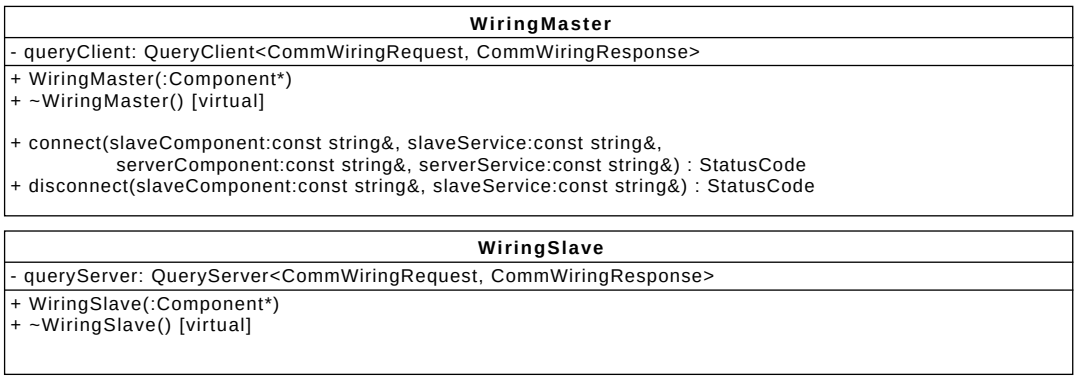

Figure A.20.: Wiring pattern, user interface classes.

Slave Side

On the slave side, the pattern does not feature any user API. When realizing the pattern, one needs to make sure that all component services are connected to the wiring slave service per default. This could either be done on the framework code level or using an MDSD approach by generating this per default.

A.2.4. Results (Event)

This section describes the realization of relevant parts of the pattern, including the user view on the pattern. Figure A.21 shows the typical usage of the pattern. UML models are used to define the structures on this level. The here presented structures follow those defined by the model and meta-model layer. The realization of the pattern follows the SmartSoft event pattern as introduced by Schlegel in [Sch04]. The communication pattern described by Schlegel offers further possibilities to make use of the pattern in a wider spectrum of use cases. The realization of the here presented event pattern tailors

the SmartSoft event to the specific use case described, thereby reducing some of the options to choose from. The reduction enables an easier usage, tooling integration and automation, when bridging the different abstraction levels, from services to skills. The following description is reduced to the coordination relevant aspects; further details such as the framework integration are not detailed and depend on the realization of the pattern. Further details on how the pattern can be implemented are described by Schlegel in [Sch04] as SmartSoft event.

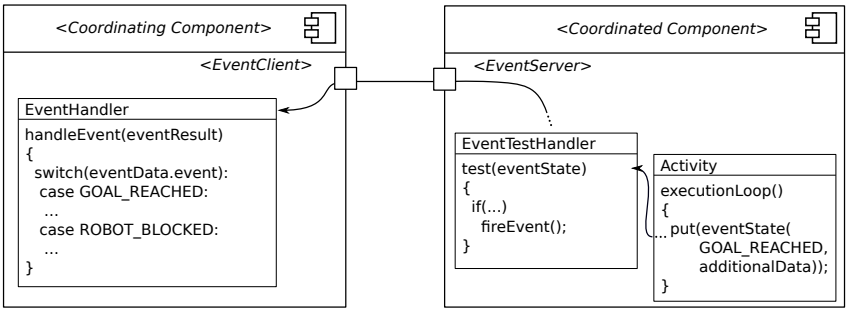

Figure A.21.: Event pattern, asynchronous results send back from the coordinated component.

In contrast to the other coordination patterns presented so far, the event pattern does not use a master-slave style connection but a server-client. While the other patterns use generic communication data, the event pattern can communicate domain-specific user-defined messages and features an inverted communication direction. Therefore, the coordinating component features one pattern slave endpoint per service. If a coordination service does contain multiple event services, the coordinating component must feature multiple event slaves. To decouple the non-generic endpoints from generic coordinating components, a plug-in mechanism is proposed, see section A.1. Figure A.22 show the user interface of the pattern accessible by the role of the component developer. The sequence diagram in Figure A.23 shows the typical usage of the pattern and interaction of client and server.

Server Side

The server side of the pattern is used by the coordinated component which is the originator of the communicated data. The server-side interface is split into two parts. The component developer accessing the frameworks API uses a single method to communicate the event close to the realization of the functionality. Typically this is part of the glue code which glues functionality into the component. The method *put* takes the current event state and hands it over from user code to the second part of the user interface the *EventTestHandler*. Within the *EventTestHandler* the user implements the logic to decide if the event has to be sent to a specific even activation. The *EventTestHandler* features a single method (testEvent) the component developer has to implement. The

method provides access to the activationData, the eventState handed over via the put method. The method is a callback that is called for each activation of the event. This can be done for different slaves or multiple activations from the same client. If the test handler validated to a successful result (return value of the handler equal to true) the user needs to fill the eventResult object which is then communicated to the client-side.

Client Side

The client-side of the pattern is used by the coordinated component. The client sides user interface consists primarily of a single handler. To manage the usage of the events from the client-side, the coordinating component activates and deactivates the events. During the activation of the client-side, the event activation object is communicated to the server. The server can use this object to decide if the event should be sent to a specific activation and client. The handler interface is called asynchronously once the event is activated on client-side and provides access to the eventResult object communicating the event itself.

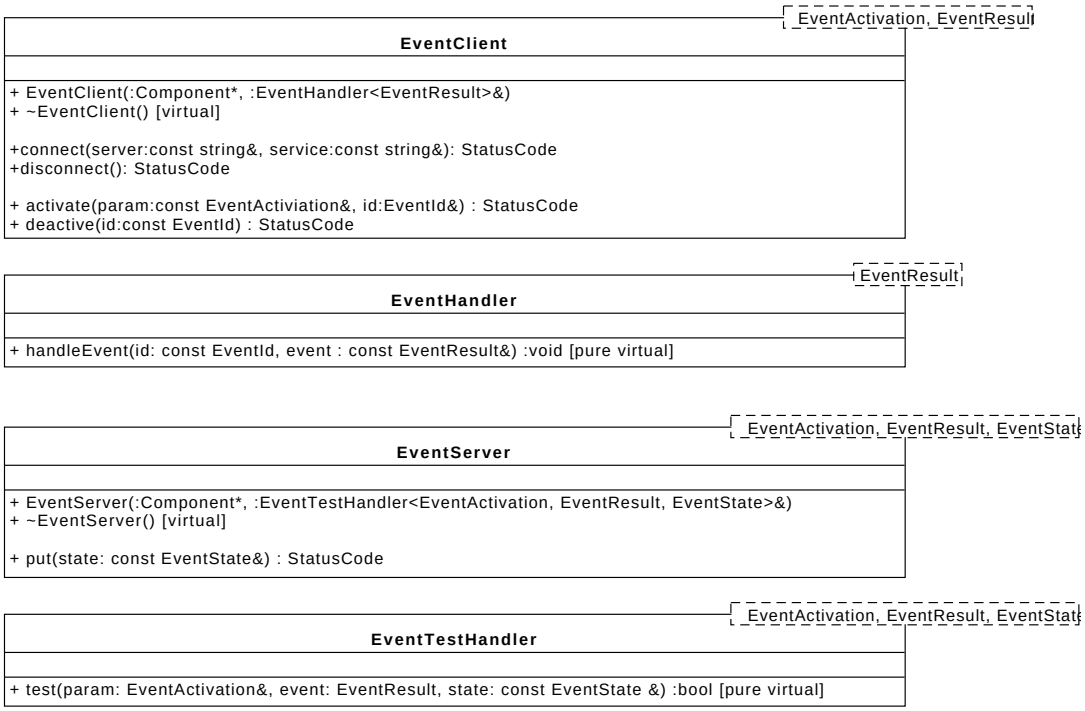

Figure A.22.: Event pattern, client and server user interface.

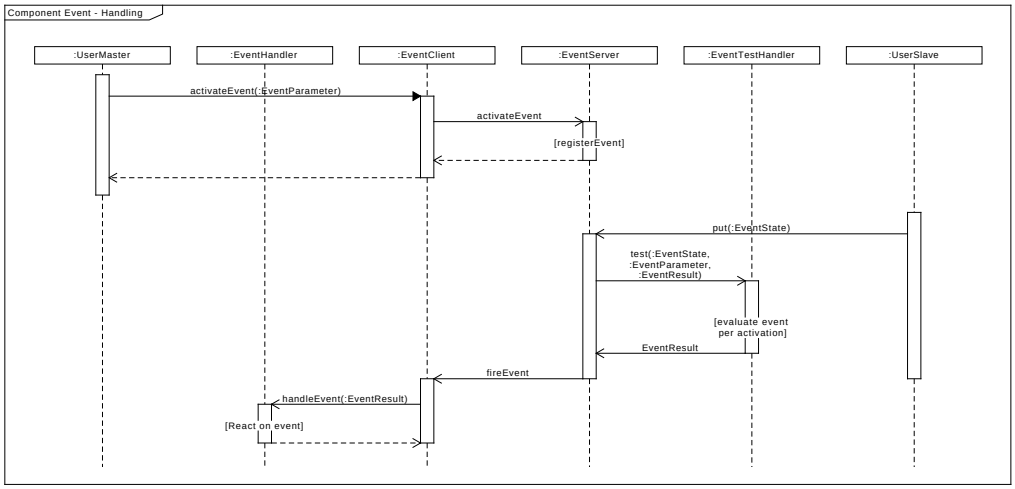

Figure A.23.: Event pattern, client/server interaction including user interface.

A.2.5. Information Query

This section describes the user view on the usage of the pattern. Figure A.24 shows the typical usage of the pattern. The described structures are realized on the code layers (component and framework). UML models are used to define the structures on this level. The here presented structures follow those defined by the meta-model layer. The realization of the pattern follows the structures and the communication semantics of the SmartSoft query pattern as described by Schlegel in [Sch04]. No further structures than those described by Schlegel are required. The following focuses on the visible user interface and the communication semantics, no framework realization is given, for further details see [Sch04]. The pattern is divided into two parts, a client and a server, according to the architecture of the coordination interface. The client part maps to the SmartSoft query client and the master to the SmartSoft query server. The sequence diagram A.26 illustrates the client-server interaction.

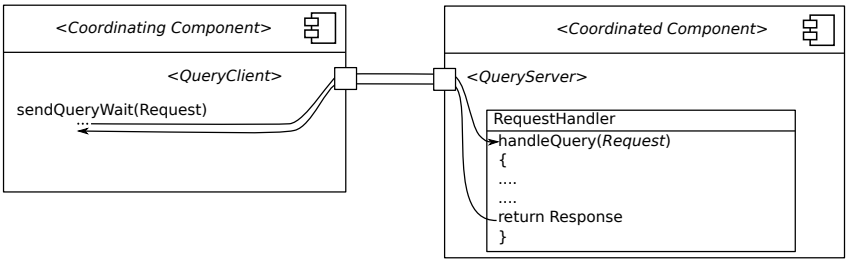

Figure A.24.: Query pattern, request for information used to coordinate software components.

The information query pattern introduces user-specific data. The pattern is realized

as client-server type with a one to N communication semantics with the server being able to deal with multiple clients requesting information. This is required to enable the reuse of communication services for coordination and to deal with possible multiple clients using the same server. The coordination component must therefore feature one endpoint per information query service. The realization can be done using a plug-in interface described in Chapter A.1. This decouples the coordinating component from concrete service types.

Client Side

The client-side of the pattern is used within the coordinating component. A simple one method interface realizes the client side interface. The method *sendQueryWait*, containing the request and response communication data as parameter. The return value of the call is only used to deal with error handling. The response user data is communicated via the second parameter. In contradiction to the fully-fledged SmartSoft query pattern, the pattern for information request for coordination limits the communication semantics to blocking synchronous. Therefore, the pattern does not offer any further user handler. Offering a non-blocking interface would introduce semantic overlap with the event result pattern and is therefore omitted. In the context of behavior coordination and for the use case the pattern is thought, a non-blocking interface is not strictly required. If the request takes a longer time to process, it should be modeled using an activation and the asynchronous event notification, as it is then arguably consuming a considerable amount of resources. Figure A.25 illustrates the client-side user interface. The client-side of the pattern is compatible with the corresponding SmartSoft query pattern as it only uses a subset of the interface but does not change the pattern's interface.

Server Side

The server side of the pattern is used by the coordinated component. The interface of the server-side is realized using an up-call handler. The handler features the request data as parameter and returns the response data. In contrast to the SmartSoft realization of the *queryServer*, the handler of this pattern is not offering to respond to the request from a different location than the handler itself. This helps to encourage a short response time as the pattern is used for synchronous requests only. When implementing the pattern based on SmartSoft, this detail can be neglected. The important part is to get the user not to implement long blocking parts within the patterns handler. Figure A.25 illustrates the slave side user interface.

A.2.6. Component Lifecycle

The following describes the user view and the usage of the component lifecycle pattern. Any implementation of the pattern depends on the robotics framework to implement

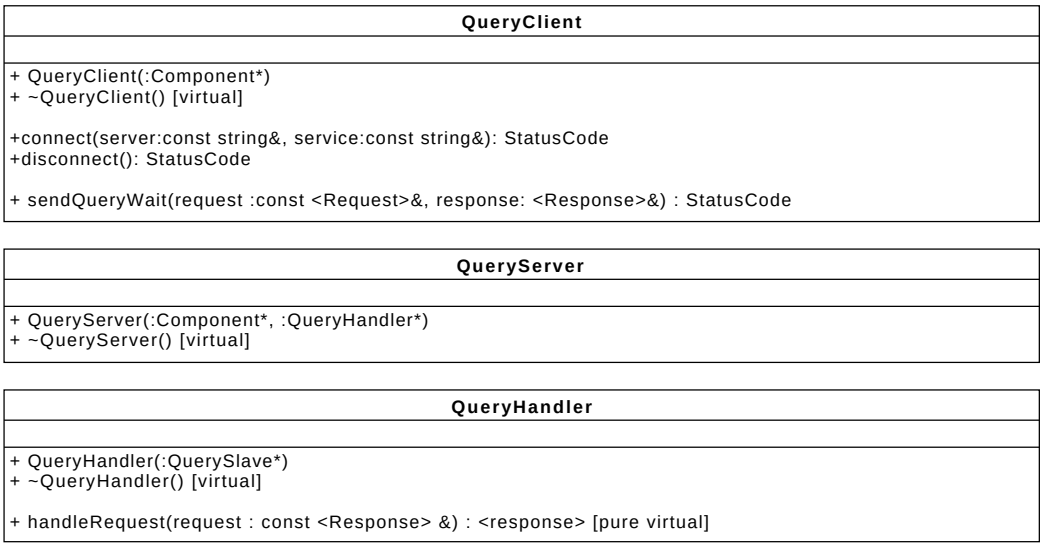

Figure A.25.: Query pattern, client and server user interface.

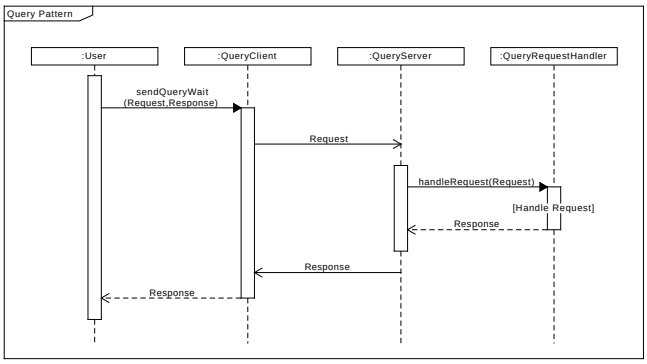

Figure A.26.: Query pattern, client/server interaction including user interface.

the pattern with or in and how the framework realizes the component model and the run-time lifecycle. The realization of the pattern uses the SmartSoft communication patterns and their semantics as described by Schlegel in [Sch04]. Any implementation of the pattern based on another framework needs to realize the same communication semantics.

The component lifecycle pattern combines two communication directions with different communication semantics. The coordination part controlling the coordinated component lifecycle from the outside follows the typical master-slave communication direction and semantics, see Figure 6.36. The communication follows a one-to-one connection that can be established each time or can be kept for each slave. The slave part

enables the coordination from one master only. Simultaneous connections of multiple masters are not possible and not reasonable from a control architecture perspective. The coordination master sends a state change request synchronously to the slave. On the slave side, a handler up-call realizes the user interface. The handler features the old and new state as parameters to enable the user to react to the state change. The handler is intended to be the entry handler for the new state. Dependent on the component model and the lifecycle states, the realization can dispatch specific handlers from this up-call. For the here used minimal lifecycle, the only specific handler would be the one for the component shutdown. The end of the components' lifecycle state change is postponed till the end of the user handler. The pattern's master port is blocked till the state change is completed. Therefore, the user should not perform time-consuming actions. To realize this communication semantics, the SmartSoft query pattern realizing a request-response communication can be used. The lifecycle coordination slave contains the query server being called by the master with a synchronous blocking query. The communication data used by the pattern consists of a simple message containing the new state and an empty response value. The result value is required for error handling only and does not contain any user data. The slave can not decline the state change request if the requested state change follows the lifecycle model.

The second part of the pattern consists of an asynchronous notification of the master in case of a coordinated component-induced state change. The user interface of the pattern on the slave side consists of a simple set method to trigger the state change. The slave then notifies the master. The pattern realizes a publish-subscribe communication semantics. The realization uses the SmartSoft push (newest) pattern, with an additional client-side handler. The mast side, located at the coordinating component, features the push client. On the slave side, the coordinated component features the push server. The pattern extends the master (push newest pattern side) with an up-call notification handler. The master needs to keep track of multiple connections as it is coordinating multiple components. Dependent on the coordination approach, the user can decide to dispatch the up-calls to separated handlers, e.g., one per coordinated component. The pattern further expands the slave side as well. A separated on-activation handler is added, which is used to force the slave side to publish the current live cycle state once the master is connected and subscribed. For component coordination purposes, the slave side needs to deal with one master connected to it only.

A.3. Robotics Behavior Coordination Models - SmartTCL Realization

This section introduces the meta-models for robotics behavior coordination, realized using the coordination approach SmartTCL [SS10]. The meta-models follow the generic structures presented in Chapter 5.

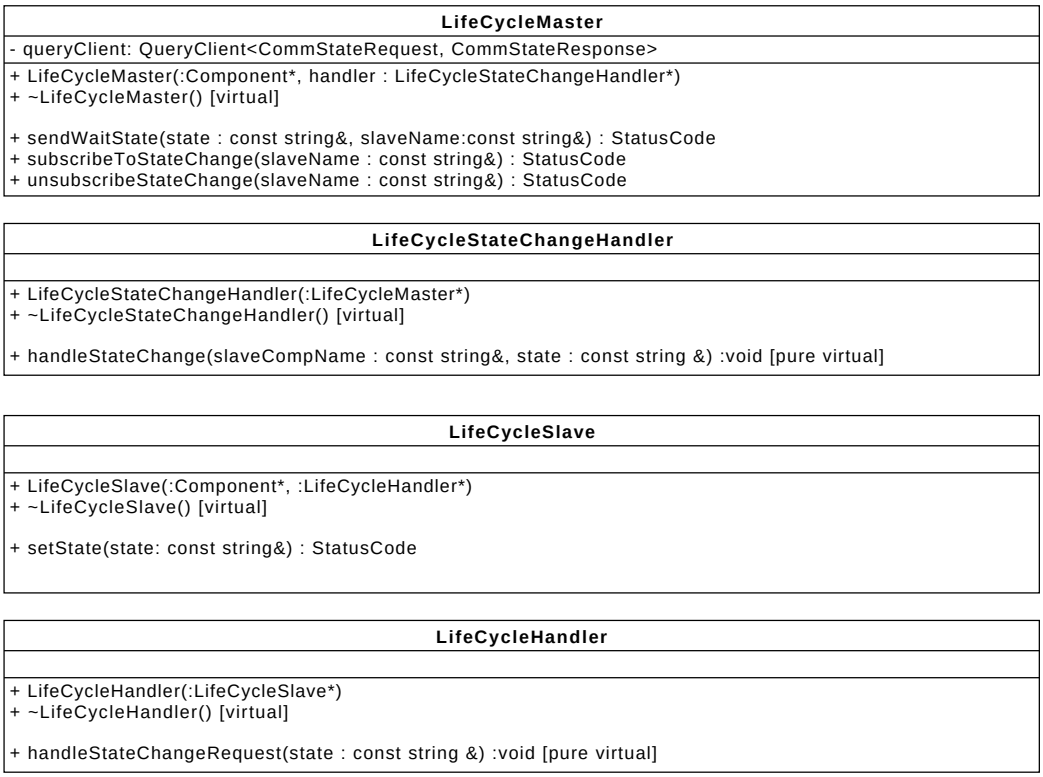

Figure A.27.: Component lifecycle pattern, master and slave user interface.

A.3.1. Task Models - SmartTCL Realization

this section introduces the meta-models of the robotics behavior coordination models on task abstraction level, realized using the coordination approach SmartTCL [SS10]. The meta-models presented are realized using the Eclipse modeling framework [Fouc]. The figures shown are the actual Ecore meta-models the SmartMDSD Toolchain makes use of. The experiments in Chapter 8 show examples of the task models realized within the SmartMDSD Toolchain that are based on these meta-models.

Figure A.28 shows the task realization meta-model. The model can be partitioned into two parts. First, the core structure of the model, which follows the generic structures presented in the previous section. This part is visible on the left side of the graphical representation of the meta-model (Figure A.28). The second part contributes the logic which is used to realize the tasks. SmartTCL is realized as Lisp internal DSL, therefore, the logic of the task is expressed as Lisp code. The meta-model reflects the internal DSL realization considering Lisp expressions to be used at several places. The abstract class *Expression* subsumes all Lisp expressions. This includes some functionality of the coordination approach SmartTCL, see for example, the class *AbstractTCLAction*.

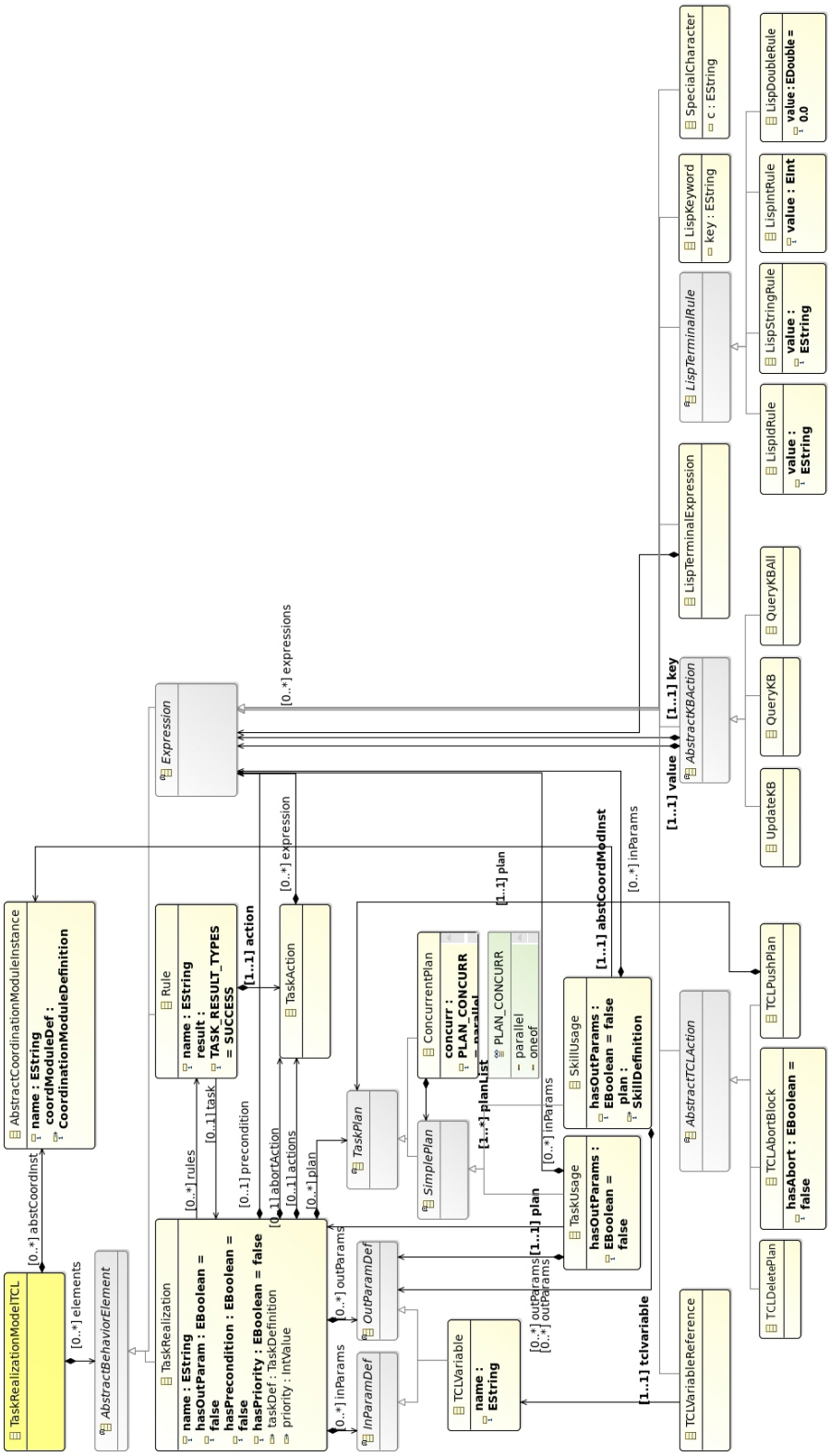

Figure A.28.: Task realization meta-model for SmartTCL.

The structural model part for the logic expressing Lisp code is kept simple and could be extended to enable further semantic validation. Some of the simplifications in the meta-model can and are, in fact, compensated by the realization of the DSL and the tooling (e.g., model check, code completion, etc.) based on the meta-model.

The classes matching the SmartTCL approach are located in the left core structures. The class *TaskRealization* features a list of further attributes such as a priority, for example. Some of the attributes are only present to ease the development of the DSL, such as *hasOutParam*, for example. The link to the definition of the task is achieved through the referenced attribute *taskDef*. The in and out Attributes of the task definition are mapped to SmartTCL variables using the class *TCLVariable*.

The class *Rule* as one of the three realizations of *AbstractBehaviorElement* is used to realize the rules for contingency handling within SmartTCL. The rules are named elements and are bound to specific tasks and result values. Same as the *TaskRealization*, the *Rules* contains an instance of the *TaskAction* class, to contain logic to be executed.

Besides the *TaskRealizations*, *Rules*, and *Expressions*, the model contains *AbstractCoordinationModuleInstaces* to instantiate the coordination module types. Thereby, the task realization can make use of the skills, via *SkillUsage*. The skills in connection with the correct module instance can be added to the plan of a task realization. The elements pushed to the plan can feature different execution semantics, parallel, one-of or sequentially (*SimplePlan*).

The experiments in Chapter 8 show the usage of the models realized as textual models, that are conform to these meta-models. Some of the semantically necessary relations are not realized in the meta-models but in the DSL and are therefore not visible. An example for this is the access of valid skills matching the selected coordination module. This relation is not expressed in the meta-model. The meta-model does, however, contain the connections to the classes to derive the required information, modeled by the class *SkillUsage*.

Task Models - SmartTCL Example

This subsection provides an example to illustrate the usage of the task models realized with SmartTCL. More comprehensive examples are shown in Chapter 8. The example task model shown in Listing A.1 realizes a transportation task. The task makes use of skills provided by the components of the coordination module instances, defined within the first lines of the model. The *AbstractCoordinationModuleInstance navigationInst coordModuleDef CommNavigationObjects.NavigationModule*, for example, provides the skill *approachLocation*, that is used to realize the task. With SmartTCL, the tasks are refined within a tree expanded during run-time. The *transportationTask* is expanded by sequentially executing the used skills, see *tcl-push-plan* in the listing.

```
1 TaskRealizationModelTCL {
    AbstractCoordinationModuleInstance navigationInst
3     coordModuleDef CommNavigationObjects.NavigationModule
    AbstractCoordinationModuleInstance kbModInst
```

```
    coordModuleDef CommBasicObjects.KBModule
  AbstractCoordinationModuleInstance localizationModInst
    coordModuleDef CommLocalizationObjects.LocalizationModule
  AbstractCoordinationModuleInstance base
    coordModuleDef CommNavigationObjects.MobileBaseModule
  AbstractCoordinationModuleInstance mpsInst
    coordModuleDef CommRobotinoObjects.MPSModule


(define-task-block (transportationTask ?startStationID ?startBeltId ?goalStationID
     ?goalBeltId)
  (taskDefinition CommNavigationObjects.transportationTask)
  (rules (ruleFetchFromMpsFailed rulePushToMpsFailed ruleRobotBlocked))
  (action ( (format t "=> transportationTask from:~s to:~s ~%" ?startStationID ?
    goalStationID)
    (let*   ((start-station (tcl-kb-query :kb-key '(is-a id)
             :kb-value `((is-a station)(id ?startStationID))))
         (start-station-approach-location (get-value start-station '
    approach-location))
         (goal-station (tcl-kb-query :kb-key '(is-a id)
           :kb-value `((is-a station)(id ?goalStationID))))
         (goal-station-approach-location (get-value goal-station '
    approach-location)))
             (tcl-push-plan :plan `(
               (localizationModInst.activateLocalization)
               (navigationInst.approachLocation ,start-station-approach-location)
               (mpsInst.mpsStationFetchFrom ?startStationID ?startBeltId)
               (navigationInst.approachLocation ,goal-station-approach-location)
               (mpsInst.mpsStationPushTo ?goalStationID ?goalBeltId)
               (localizationModInst.deactivateLocalization)))))))
         ...
```

Listing A.1: Task block realizing a transportation task of goods between modular production stations.

A.3.2. Skill Models - SmartTCL Realization

This section presents the skill realization meta-model for the SmartTCL based approach. As with the task meta-model, this model follows the structures introduced in the Chapter 5, applied to the specifics of SmartTCL. These meta-models are realized using the Eclipse modeling framework Ecore. Since both the skills and the tasks use the same coordination approach, the meta-model for the skills is somewhat similar. The skill meta-model is presented second, as it basically uses the task meta-model and extends it with access to the component coordination interface. Figure A.29 shows the skill realization meta-model, which is used to realize the tooling to support the technology developer within the SmartMDSD Toolchain. The experiments in Chapter 8 show examples of the skill models realized within the SmartMDSD Toolchain, based on the presented meta-model.

This meta-model can be partitioned into three parts, the first two parts are similar to those of the task realization meta-model tasks. First, on the left side (Figure A.29), the classes represent the core structures of SmartTCL. Second, on the bottom and the center, the classes represent the logic to realize the skills, again based on the Lisp internal DSL

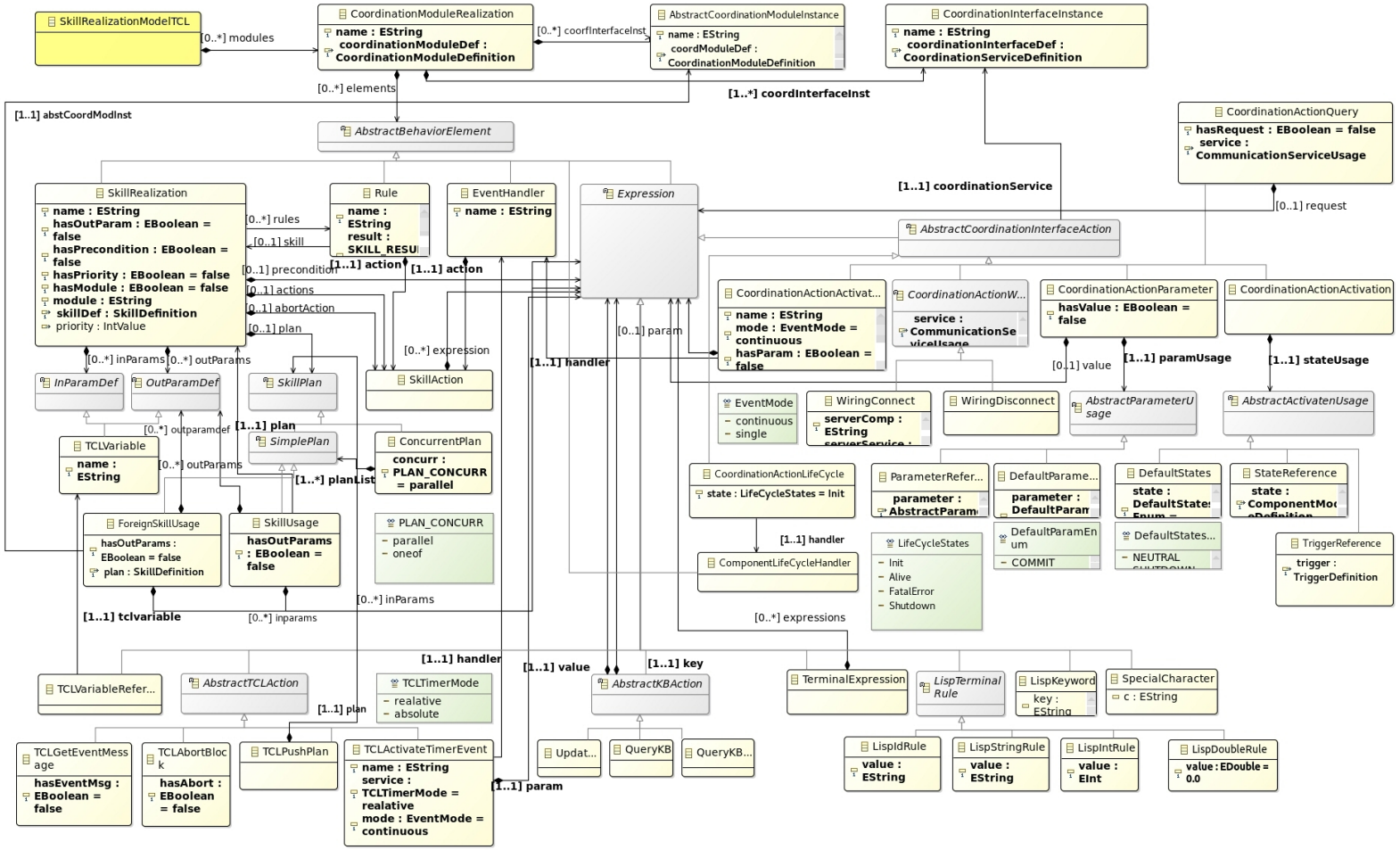

Figure A.29.: Skill Realization meta-model for SmartTCL. The meta-model is rather extensive, some of the class or attribute names are cut to make the model fit onto a sheet.

for SmartTCL. The third part, on the right side, represents the structures to access the component coordination interface, to perform the coordination and interaction with the components.

The access to the coordination interface is realized with all derived classes from *AbstractCoordinationInterfaceAction*. This includes activation, parameter, connection handling, information query, asynchronous results (events), and the component lifecycle. This part of the meta-models follows the realization of the independent structures presented in the coordination interface (Chapter 6), further details can be found there. The typed access to the coordination interface is provided by the class *CoordinationInterfaceInstance*, which contains a reference to the coordination service definition. To make use of the asynchronous results, the model contains a handler represented by the class *EventHandler*. The same holds true for asynchronous notifications of component life cycle changes, the class *ComponentLifeCycleHandler* represents the handler for the up-call notification.

The skills represented by the class *SkillRealization* are contained in a coordination module. The class *CoordinationModuleRealization*, therefore, contains all instances of the model. To enable the usage of skills not realized within the own coordination module, instances of coordination modules can be defined, class *AbstractCoordinationModuleInstance*. Skills from other coordination modules are used by being pushed to the plan of a SkillRealization. The class *ForeignSkillUsage* represents the usage of skills from other modules. Therefore, it contains a reference to the *SkillDefinition*. The other parts are similar to the task realization model.

Skill Models - SmartTCL Example

This subsection provides an example to illustrate the usage of the skill models realized with SmartTCL. More comprehensive examples are shown in Chapter 8. The example skill model shown in Listing A.2 realizes the loading of a good from a modular production station to a mobile robot. The skill makes use of coordination services to connect to the components. The services used and the coordination module the skill is contained by are modeled in the first lines of the model. Within the action clause of the skill the run-time parameters are set to configure the station-id to communicate with. The skill uses the state to activate the cyclic activity *load* and the event *loadevent* to receive the results of the activation. The last lines of the model show the realization of the event handler *handlerMpsLoading* that handles the results, in this case mainly returning the skills' result value.

```
1 SkillRealizationModel {
    CoordinationModuleRealization MPS coordModuleDef CommRobotinoObjects.MPSModule
      uses {
3   CommRobotinoObjects.FestoMPSDockingCoordinationService instName mpsdocking
    CommRobotinoObjects.ConveyerBeltCoordinationService instName belt
5 }
  {
7 (define-skill-block (mpsStationLoad ?stationid)
```

```
(skillDefinition mpsStationLoad)
(module "MPSModule")
(abort-action ( (format t "=>>> ABORT ACTION mpsStationLoad ~%")
                (tcl-ci-state :server belt :state Neutral)))
(action (
          (format t "=>>> mpsStationLoad stationid: ~a~%" ?stationid)
          (format t "DO LOADING ~%")
          (tcl-ci-activate-event :name evtBeltLoading
                                 :handler handlerMpsLoading
                                 :server belt
                                 :service loadevent
                                 :mode continuous)
          (tcl-ci-param :server belt :param CommRobotinoObjects.
  RobotinoConveyerParameter.SetStationID :paramvalue ?stationid)
          (tcl-ci-param :server belt :param COMMIT)
          (tcl-ci-state :server belt :state load))))


  (define-event-handler (handlerMpsLoading) (action (
   (format t "=>> HANDLER MPS LOADING: ~s ~%~%" (tcl-event-message))
          (cond
            ;;LOAD
            ((equal (tcl-event-message) "(load not done)")
              (format t "==========>>> load START ~%" ))

            ((equal (tcl-event-message) "(load done)")
              (format t "==========>>> load DONE SUCCESS ~%" )
              (tcl-ci-state :server belt :state Neutral)
              (tcl-abort)
              '(SUCCESS ()))

            ((equal (tcl-event-message) "(load error no box)")
              (format t "==========>>> load DONE ERROR ~%" )
              (tcl-ci-state :server belt :state Neutral)
              (tcl-abort)
              '(ERROR (LOAD LOADING)))
      ...
```

Listing A.2: Skill block realizing the load of a good from a modular production station to a mobile robot.

A.4. Detailed Technical Experiments

A.4.1. RobMoSys Mood2Be - Composition of Task Level Behavior Blocks

The following experiment focuses on the use of a task-level robotics behavior coordination approach. The experiment has been developed in the context of the RobMoSys open-call as Integrated Technical Project MOOD2Be by Davide Faconti, Eurecat [Pro19b]. The project developed the robotics behavior coordination approach BehaviorTree.CPP [Faca] using the concept of behavior trees (overview by Colledanchise and Ögren in [CÖ18]), as well as the matching graphical designer Groot [Facb]. A video of the application demonstrating the integration of the behavior coordination approach with the Smart-MDSD toolchain has been published by the project [Fac+18]. Figure A.30 shows the

application running in a lab environment (at THU) as well as the runtime visualization of the behavior tree in Groot. Within this application, the behavior tree approach orchestrates a Robotino robot to transport small load carriers between two stations using the components and the skills of the system.

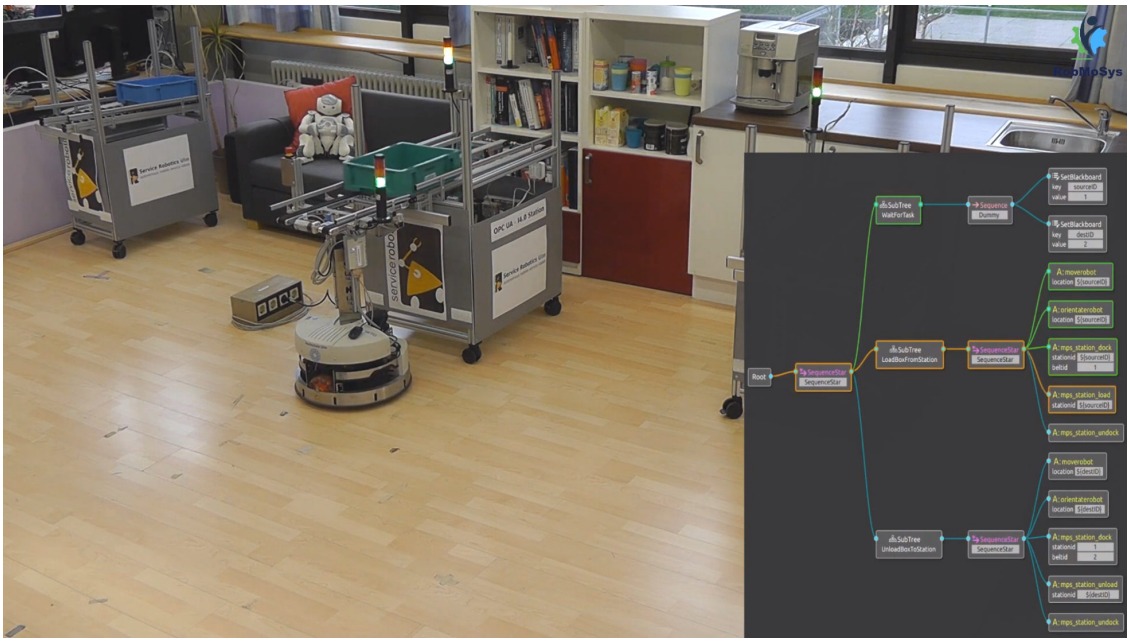

Figure A.30.: The behavior tree realizes a simple transportation task, with a Robotino robot and two stations to fetch and deliver small load carriers. The right overlay shows the runtime visualization of the behavior tree in Groot. Picture from video published by RobMoSys [Fac+18].

This experiment is one of the examples where a third party developed a behavior coordination approach, which makes use of the skills provided by the composed components, is used to realize an application. Figure A.31 illustrates the interaction of the system parts that deals with the tasking and coordination of the system. Table A.10 shows the used skills and coordination modules of the application. Basically, the graphical editor Groot is able to fetch a list of all available skills of the robotic system. This is done via a runtime JSON based interface but could also be done directly via the system datasheet or within the SmartMDSD Toolchain accessing the Ecore meta-models. Using those skills, the behavior developer is able to realize the tasking of the application. An example behavior tree in Groot is visible in Figure A.32. The BT executor later takes the developed behavior tree and executes it, invoking the used skills. This order assumes that the system is already composed, which was reasonable in the project but is not a necessity. The order of development can be changed in such a way that the behavior developer is using the domain model-defined skills and coordination modules. When composing the components to a system and adding the developed behavior, the system

builder needs to ensure that all the used skills (and coordination module instances) are present. This check can be automated as all the information is available from the different models (is realized in the SmartMDSD Toolchain for SmartTCL). In this way, the workflow proposed in Chapter 7 can be realized using the presented behavior approach and tooling of this work.

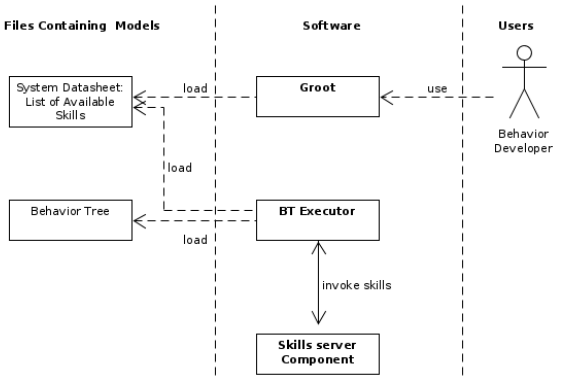

Figure A.31.: Interaction among system parts. The behavior developer is using Groot to model a behavior tree. Groot provides the skills to the user which are offered by the system model datasheet (JSON interface). During runtime the BT executor runs the developed behavior tree invoking the execution of the modeled skills. Figure from RobMoSys Wiki [Pro19e].

A.4.2. Skill Realization Dependencies - Utilized Coordination Modules

This experiment demonstrates the usage of other skills provided by foreign coordination modules for the purpose of skill realization. The realization of skills requires the coordination of one or many components. The encapsulation introduced with the presented concept of skills and coordination modules prevents access to components not belonging to its own coordination module. In some cases, it is, however, necessary to make use of the functionality realized within other components. The realization of navigation skills, for example, requires the usage of sensor data and therefore the sensors need to be coordinated e.g. activated. To keep the encapsulation and the independence of the skill realization from other specific components, the skills are able to make use of skills not defined within their own coordination module. To do so, skill realizations can define abstract coordination module instances. The instances are used the same as those being defined for task realization.

Figure A.33 shows an example use case of the described approach. The excerpt of the skill-realization model shows the header with the instantiated coordination modules, as well as the usage of skills available via those module instances. In this experiment, the

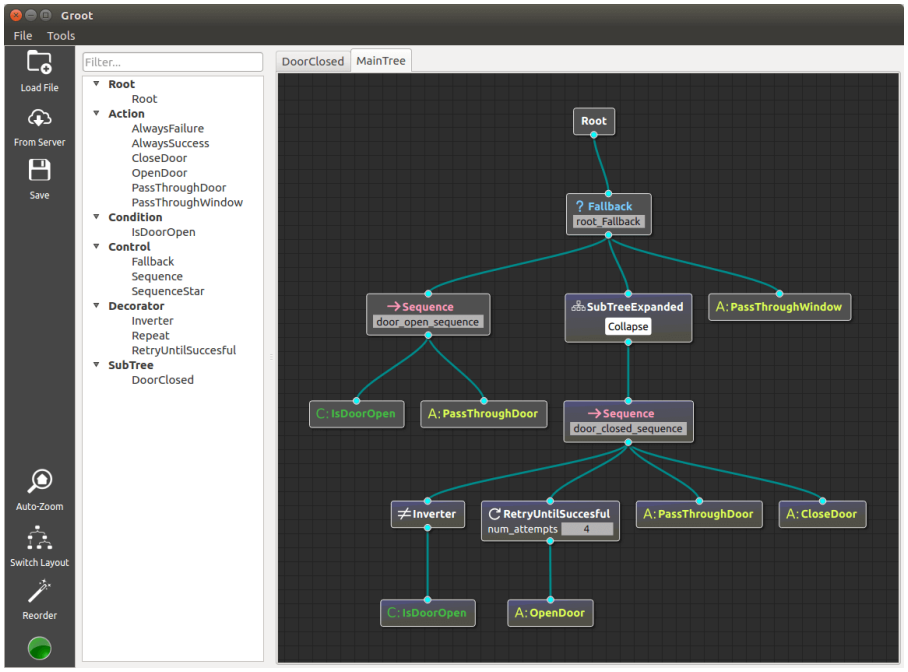

Figure A.32.: The graphical designer Groot for the behavior trees, developed within the ITP Mood2Be that is part of RobMoSys, figure from RobMoSys Wiki [Pro19e].

skill-realization model for the navigation coordination module uses skills from other modules. Skills from the coordination modules *LaserModule* and *MobileBaseModule* are used. Instances of those types are created and used during the realization of the skills. The skill realization *approachLocation* uses the skills of the *laserModule* instance to activate and deactivate the generation of laser measurements.

During system composition, performed by the system builder, the abstract instances introduced by the skill realizations need to be bound in the same way as those defined for task realization. Figure A.34 shows the mapping of the instances. The tooling supports the system builder to bind all necessary abstract instances. During runtime, care has to be taken to avoid duplicated instantiation of all required modules. Abstract instances from skill realizations need to be mapped to the same possibly existing instances already used to map those present from the task realizations.

To demonstrate the approach of how to use the abstract coordination modules instances within the skill realization, the previous experiment is extended. The existing robots system hardware and software is altered, while the laser ranger sensor is replaced by an Intel realsense rgbd camera, see Figure A.35. The video [SL19] illustrates the exchange of the sensors and how the software system using the SmartMDSD Toolchain has to be changed. The system builder fetches two components from the ecosystem – the *ComponentRealSenseV2Server* (device driver) and the *ComponentLaserFromRGBDServer*

Skill	Parameter In/Out	Purpose
NavigationModule		
initNavigation		setup navigation
moveRobot	location	move robot to location, simple
moveRobotOrientate	location	rotate robot as defined by location
MPSModule		
mpsStationDock	stationId beltId	dock to mps station
mpsStationUndock		undock from mps station
mpsStationLoad	stationId	load a container from station to robot
mpsStationUnload	stationId	unload a container from robot to station
manualLoad		load a container on robot by human worker, handover via button
manualUnLoad		unload a container from robot by human worker, handover via button

Table A.10.: Skills provided by the coordination modules of the robot used by Groot and BehaviorTree.CPP to realize tasks and the application.

```
SkillRealizationModel {
    CoordinationModuleRealization PlannedCdlNavigation coordModuleDef CommNavigationObjects.NavigationModule
    uses {
        CommNavigationObjects.CdlCoordinationService instName cdl
        CommNavigationObjects.MapperCoordinationService instName mapper
        CommNavigationObjects.PlannerCoordinationService instName planner
    }
    usedModules {
        AbstractCoordinationModuleInstance laserModule coordModuleDef CommBasicObjects.LaserModule
        AbstractCoordinationModuleInstance baseModule coordModuleDef CommNavigationObjects.MobileBaseModule
    }
{

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; approach-location

(define-skill-block (approachLocation ?location)
  (skillDefinition approachLocation)
  (module "NavigationModule")
  (precondition  (equal '(region) (get-value (tcl-kb-query :kb-key '(is-a name) :kb-value '( (is-a location) (name ?location))) 'approach-type)))
  (rules (ruleUnknownLocation ruleLocationSuccess ruleRobotBlocked ruleAbortJobErrorAck))
  (abort-action (
                (tcl-ci-state :server cdl :state Neutral)
                (tcl-ci-state :server mapper :state Neutral)
                (tcl-ci-state :server planner :state Neutral)
                (tcl-push-plan :plan '((laserModule.setNeutral)))))
  (plan (
          (laser.setActive)
          (moveRobotLocation 'approach-region ?location)
          (moveRobotLocation 'orientate-region ?location)
          (laser.setNeutral))))
```

Figure A.33.: Excerpt of the skill-realization model for the navigation coordination module of the example. The modules use skills from the coordination modules *LaserModule* and *MobileBaseModule*. Instances of those types are created and used during the realization of the skills. The skill realization of *approachLocation*, uses skills of the *laserModule* to activate and deactivate the generation of laser measurements.

(converting rgbd images to laser scans) – and composes them to the system, replacing the *ComponentLaserLMS1xx* for the sick laser ranger. The required changes only affect the component architecture model. Both, the task models as well as the skill realizations

```
ForeignCoordinationModuleMapping {
    moduleInstance laserModule realizedby laser {
        interfaceInstance LASER realizedby ComponentLaserLMS1xx
    }
}

ForeignCoordinationModuleMapping {
    moduleInstance baseModule realizedby baseInst {
        interfaceInstance base realizedby ComponentRobotinoBaseServer
    }
}
```

Figure A.34.: System integration, component architecture model showing the mapping of abstract coordination module instances defined by skill realizations and the components realizing them.

provided by the components are unaffected.

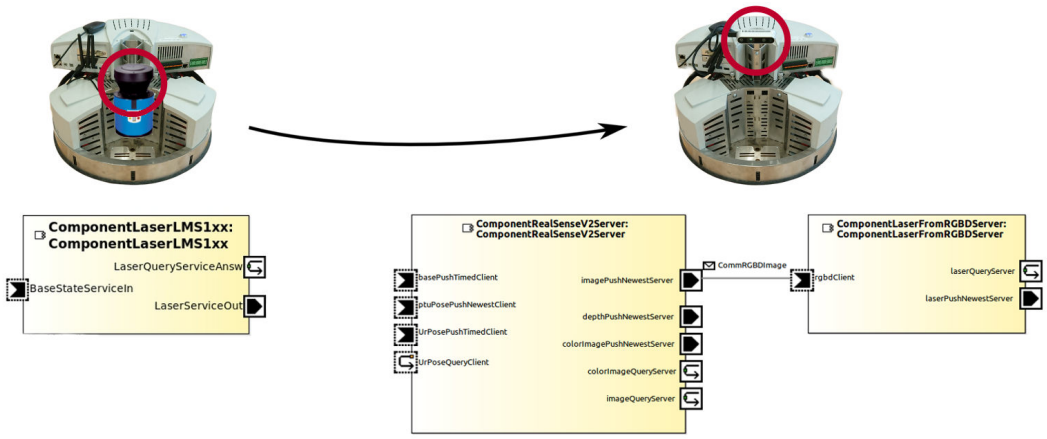

Figure A.35.: Replacing laser with rgb camera on Robotino3.

Figure A.36 illustrates the changes to the component architecture model. Apart from the two new components replacing the laser component, the coordination module mapping in the component architecture model (textual representation) needs to be changed, see Figure A.37. The module realization for the laser coordination module is now realized by the component *ComponentLaserFromRGBDServer*. The coordination module realization *LaserFromRGBDModule* further adds an abstract coordination module instance *cameraModule*. The instance also needs to be mapped; it is provided by the component *ComponentRealSenseV2Server*. The new module is required to coordinate the camera component to activate the capture of rgbd images, for example. The skill *setActive*, now realized within the *LaserFromRGBDModule*, therefore makes use of the skill *activateCameraPush*, see A.38.

This experiment demonstrates how the approach deals with dependencies between the skills and other coordination modules. The experiment shows that the skills can be used decoupled from concrete components, even when using functionalities from other

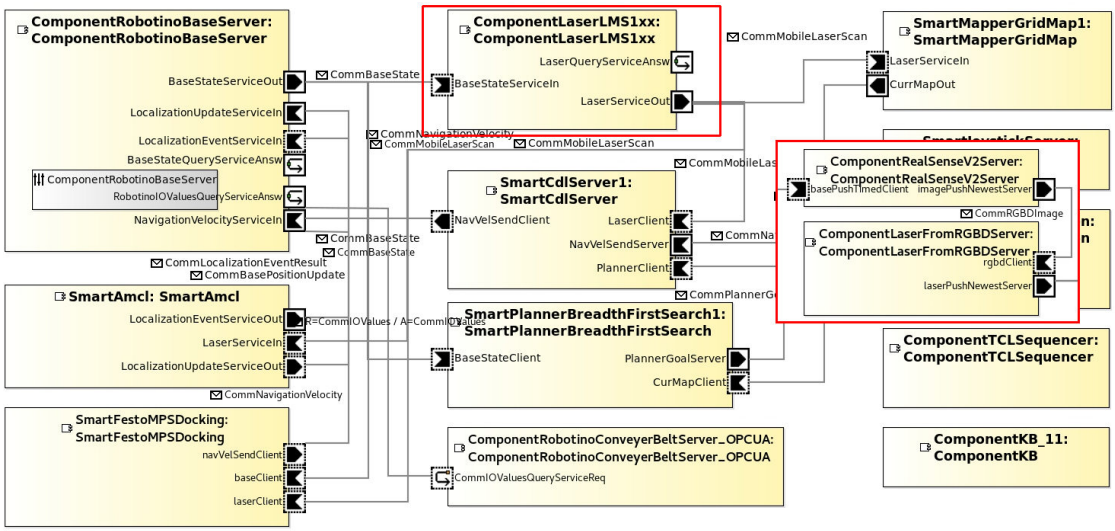

Figure A.36.: SmartMDSD Toolchain, component architecture model showing the changes required to realize the software side of replacing the laser ranger against an rgbd camera. The red-marked components are replaced.

```
ForeignCoordinationModuleMapping {
    moduleInstance laserModule realizedby LaserFromRGBDModule {
        interfaceInstance laserfromrgbd realizedby ComponentLaserFromRGBDServer
    }
}

ForeignCoordinationModuleMapping {
    moduleInstance baseModule realizedby baseInst {
        interfaceInstance base realizedby ComponentRobotinoBaseServer
    }
}

ForeignCoordinationModuleMapping {
    moduleInstance cameraModule realizedby RealsenseCameraModule {
        interfaceInstance realsense realizedby ComponentRealSenseV2Server
    }
}
```

Figure A.37.: Excerpt of the textual component architecture model which shows the mapping of the abstract coordination module instances introduced due to the swapped components and coordination instances. Regarding robotics behavior coordination, these are the only changes necessary for system integration. All other models, skill and task stay unchanged.

coordination modules. This is an important prerequisite for the ecosystem, as it enables the composition of building blocks developed by different roles that can be separated in space and time. The experiment further demonstrates that different realizations of functionality and coordination modules are possible. The realization of independent tasks is not affected by swapping realizations and skills. This is again important for the ecosystem vision the composition and reuse of building blocks, including coordination.

```
SkillRealizationModel {
    CoordinationModuleRealization LaserFromRGBDModule coordModuleDef CommBasicObjects.LaserModule uses{
        CommBasicObjects.LaserFromRGBDCoordinationService instName laserfromrgbd
    }
    usedModules {
        AbstractCoordinationModuleInstance cameraModule coordModuleDef DomainVision.CameraModule
    }
    {

    (define-skill-block (setActive)
        (skillDefinition setActive)
        (module "LaserModule")
        (action (
            (format t "====>>> setActive ~%")
            (tcl-ci-state :server laserfromrgbd :state GenerateLaser)
            '(SUCCESS ()))
        (plan ((cameraModule.activateCameraPush))))
```

Figure A.38.: Skill realization model, realizing the coordination module type *CommBasicObjects.LaserModule* as is provided by the component *ComponentLaserFromRGBDServer*. The module introduces dependencies to a coordination module of type *DomainVision.CameraModule*, as it is using skills from the module *cameraModule*.

A.4.3. Experiments - Component Architecture Models

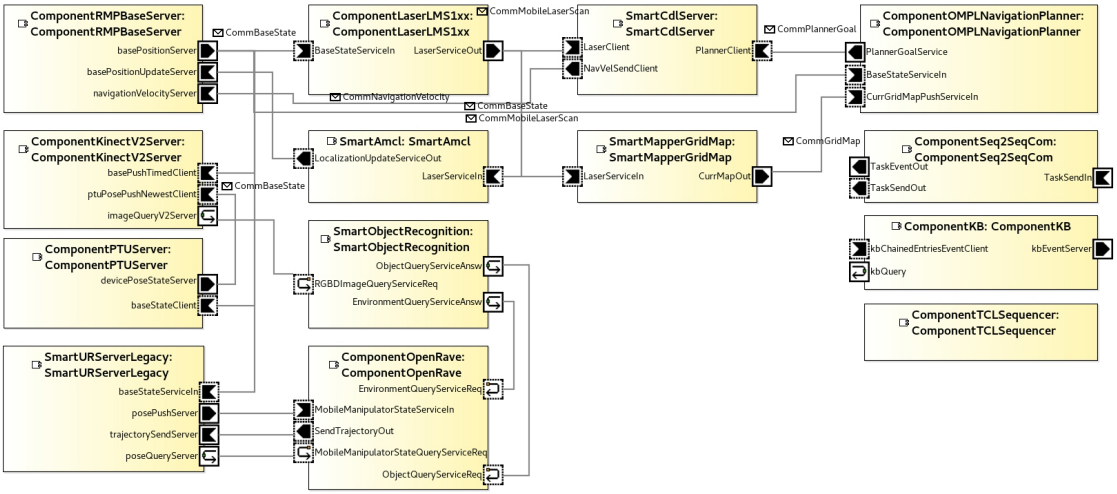

Figure A.39.: Transferability of Tasks to other Robotic Systems - Graphical representation of the component architecture model of the mobile service robot Larry.

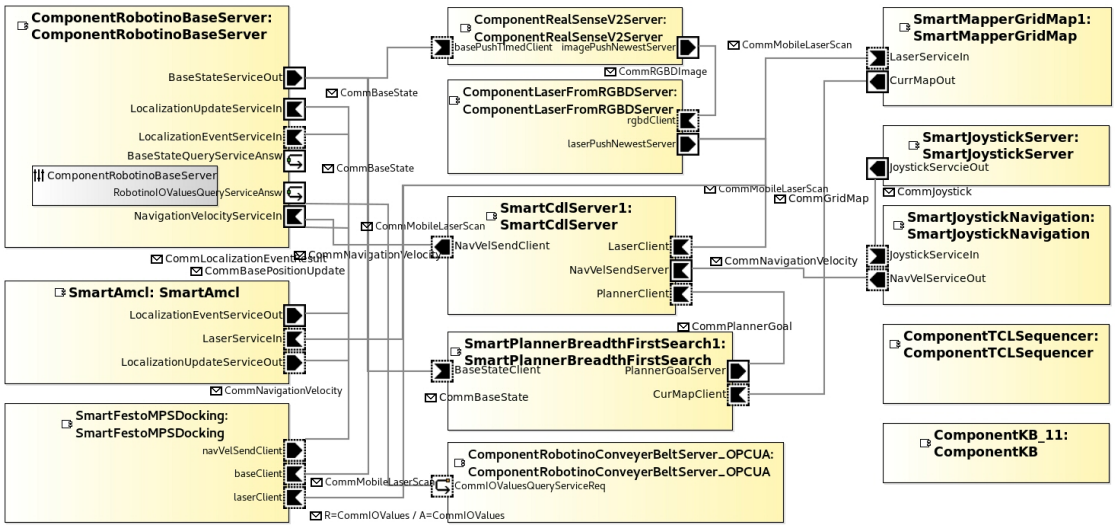

Figure A.40.: Transferability of Tasks to other Robotic Systems - Graphical representation of the component architecture model of the mobile service robot Robotino.

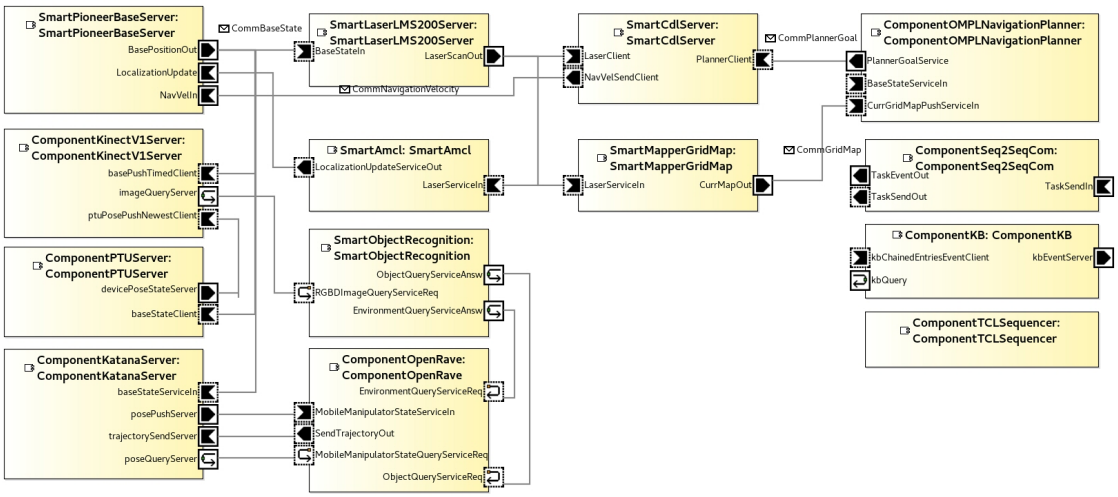

Figure A.41.: Transferability of Tasks to other Robotic Systems - Graphical representation of the component architecture model of the mobile service robot Kate.

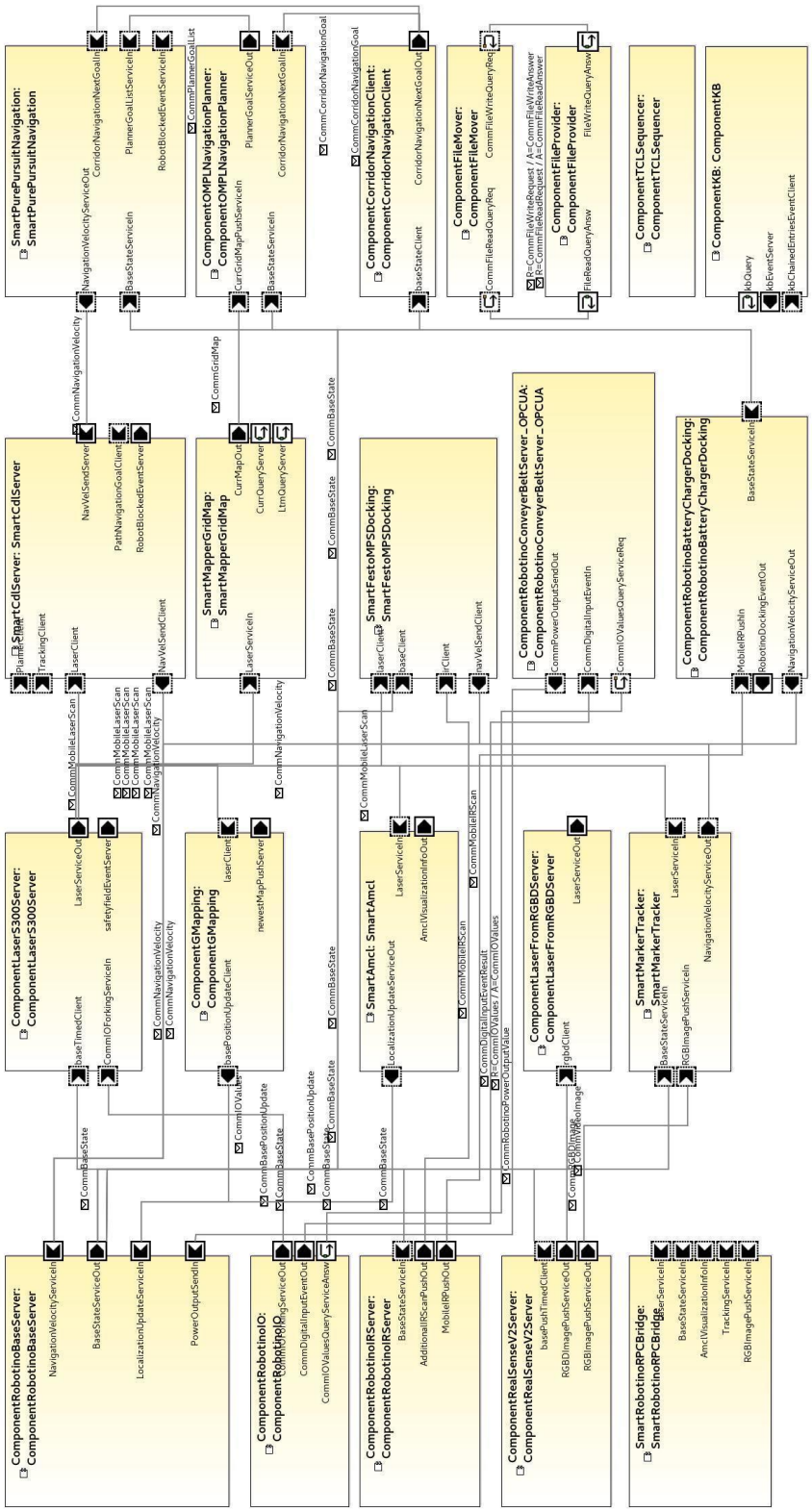

Figure A.42.: Robotino Factory 4.0 - Component architecture model of a single Robotino robot within a fleet. The model shows the components of the system with the most important connections in the initial wiring. The connections among the components are rewired during runtime depending on the context (environment, job to perform etc).

A.4.4. Autonomous Order Picking - Lists

Component	Purpose	Robotino	Larry	FleetMgmt.
ComponentTCLSequencer	Coordination	x	x	
ComponentKB	Coordination	x	x	x
ComponentSeq2SeqCom	Coordination	x	x	
ComponentFileProvider	Utility	x	x	x
ComponentFileMover	Utility	x	x	
ComponentRMPBaseServer	Navigation		x	
ComponentRobotinoBaseServer	Navigation	x		
ComponentLaserLMS1xx	Navigation	x	x	
ComponentLaserS300Server	Navigation	x		
ComponentRobotinoIRServer	Navigation	x		
ComponentRobotinoIO	Utility	x		
SmartRobotinoRPCBridge	Utility	x		
SmartCdlServer	Navigation	x	x	
SmartMapperGridMap	Navigation	x	x	x
SmartPurePursuitNavigation	Navigation	x	x	
ComponentOMPL- NavigationPlanner	Navigation	x	x	
ComponentCorridor- NavigationClient	Fleet Nav.	x	x	
SmartAmcl	Localization	x	x	
ComponentGMapping	Mapping	x		
SmartMarkerTracker	Vision	x		
ComponentLaser- FromRGBDServer	Navigation	x		
SmartFestoMPSDocking	Docking	x		
ComponentRobotino- BatteryChargerDocking	Docking	x		
ComponentRobotino- ConveyerBeltServerOPCUA	Handling	x		
ComponentKinectV2Server	Recognition		x	
ComponentRealSenseV2Server	Recognition	x		
ComponentPTUServer	Manipulation		x	
ComponentKinectBoxDetection	Recognition		x	
ComponentRackDetection	Recognition		x	
ComponentOpenRave	Manipulation		x	
SmartURServerLegacy	Manipulation		x	
SmartJobDispatcher	Coordination			x
ComponentCorridor- NavigationServer	Fleet Nav.			x
ComponentSymbolicPlanner	Coordination			x
SmartFestoFleetCom	Coordination			x
SmartRobotinoMasterRPCBridge	Utility			x

Table A.11.: Components used by the robots and the fleet management.

Skill	Parameter In/Out	Purpose
ManipulationPlannerModule		
graspObject	obj-id	grasp an object
movePose	pose	move manipulator to pose Cartesian
movePoseJointAngles	angles	move manipulator to config angular
loadEnvironmentForLocation	envId	load recognition results to planning env
ManipulatorModule		
moveLinear	goal speed acc	manipulator linear movement
moveCirular	via goal speed acc	manipulator circular movement
executeProgram	name	execute manipulator saved program
ObjRecognitionModule		
recognize	envId	trigger recognition
recognizeObjects	locationId envId	trigger recognition
objRecogSetup	location	setup recognition for location
MPSModule		
setDockingNeutral		neutral state
mpsStationDock	stationid beltid	dock to mps station
mpsStationUndock		undock from mps station
mpsStationLoad	stationid	load a container from station to robot
mpsStationUnload	stationid	unload a container from robot to station
manualLoad		load a container to robot by human worker, handover via button
manualUnLoad		unload a container from robot by human worker, handover via button
MobileBaseModule		
getBasePose	x y yaw	get current pose of robot
monitorBaseBumper		handles base bumper events
NavigationModule		
setNavigationNeutral		neutral state
initNavigation		setup navigation
loadNavigationMapFromFile	map	load grid map for navigation
approachLocation	location	move robot to location, policy by location
moveRobotPlain	x y radius	move robot to position x y
moveRobot	location	move robot to location, simple
moveRobotOrientate	location	rotate robot as defined by location
CorridorFleetNavigationModule		
setCorridorNavigationNeutral		neutral state
approachLocation	location	move robot to location
enterNetworkAtClosestPathnavNode	nodeID	enter the navigation network
pathNavInitRegisterLocation	location	acquire location resource
LocalizationModule		
activateLocalization		start localization
deactivateLocalization		stop localization
loadLocalizationMap	map	load grid map for localization
triggerGlobalLocalization		start global localization
localizationSetRobotPose	x y yaw	initialize localization with pose
SlamModule		
startMapping	x y a	start the mapping at pose
stopMapping	mapName mapDirName	stop map and save to file
saveCurrentMap	mapName mapDirName	save map without stopping
FleetManager		
waitForFleetCommands		receive messages from fleet management
addRobotToFleetRequest		add robot to fleet
addRobotToFleet		add robot to fleet
PtuModule		
moveptuAbsolute	pan tilt	move ptu
moveptu	location nmbr	move ptu as defined by location
Robot2RobotDockingModule		
setRobotToRobotDockingNeutral		neutral state
dockToRobot		start docking to robot
undockFromRobot		start undocking from robot
PersonFollowingModule		
detectPersons	personIds	detect persons in range
startFollowPerson	personId	start following person with ID
stopFollowPerson		stop person following

Table A.12.: Excerpts of the skills provided by some of coordination modules of both robots.

References

[Ada+16] Kai Adam, Arvid Butting, Robert Heim, Oliver Kautz, Bernhard Rumpe, and Andreas Wortmann. "Model-Driven Seperation of Concerns for Service Robotics". In: *Proceedings of the International Workshop on Domain-Specific Modeling*. 2016. ISBN: 978-1-4503-4894-2.
DOI: 10.1145/3023147.3023151.
URL: http://doi.acm.org/10.1145/3023147.3023151.

[Ada+17] Kai Adam, Katrin Hölldobler, Bernhard Rumpe, and Andreas Wortmann. "Modeling Robotics Software Architectures with Modular Model Transformations". In: *Journal of Software Engineering for Robotics (JOSER)* 8.1 (2017), pp. 3–16.

[And+05a] Noriaki Ando, Takashi Suehiro, Kosei Kitagaki, Tetsuo Kotoku, and Woo-Keun Yoon. "RT-Component Object Model in RT-Middleware - Distributed Component Middleware for RT (Robot Technology)". In: *Proceedings of the 2005 International Symposium on Computational Intelligence in Robotics and Automation*. Espoo, Finland: IEEE, June 2005, pp. 457–462.
DOI: 10.1109/CIRA.2005.1554319.

[And+05b] Noriaki Ando, Takashi Suehiro, Kosei Kitagaki, Tetsuo Kotoku, and Woo-Keun Yoon. "RT-Middleware: Distributed Component Middleware for RT (Robot Technology)". In: *Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems 2005 (IROS'05)*. Edmonton, Alberta, Canada, Aug. 2005, pp. 3933–3938.
DOI: 10.1109/IROS.2005.1545521.

[Awa+16] Ramez Awad, Georg Heppner, Arne Roennau, and Mirko Bordignon. "ROS Engineering Workbench based on semantically enriched App Models for improved Reusability". In: *Proceedings of the 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.

[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.

[BC11] Jonathan Bohren and Steve Cousins. "The SMACH high-level executive". In: *Robotics & Automation Magazine, IEEE* 17 (Jan. 2011), pp. 18–20. DOI: 10.1109/MRA.2010.938836.

[BCW12] Marco Brambilla, Jordi Cabot, and Manuel Wimmer. *Model-Driven Software Engineering in Practice*. Synthesis Lectures on Software Engineering. Morgan & Claypool Publishers, 2012. ISBN: 978-1-60845-882-0. DOI: 10.2200/S00441ED1V01Y201208SWE001.

[Big+13] Geoffrey Biggs, Radu Rusu, Toby Collett, Brian Gerkey, and Richard Vaughan. "All the Robots Merely Players: History of Player and Stage Software". In: *Robotics & Automation Magazine, IEEE* 20 (Sept. 2013), pp. 82–90. DOI: 10.1109/MRA.2012.2201593.

[Bil] Bundesministerium für Bildung und Forschung. *Projekt LogiRob: Multi-Robot-Transportsystem im mit Menschen geteilten Arbeitsraum.* URL: http://www.bmbf-softwareforschung.de/media/content/Infoblatt_LogiRob.pdf (visited: Mar. 1, 2021).

[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. URL: https://doi.org/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. San Francisco, California, USA: Carnegie Mellon University, 2009, pp. 111–119. DOI: 10.1145/1753235.1753251.

[Bos19] Jan Bosch. *Digital Transformation: A holistic perspective for business leaders*. 2019. ISBN: 978-91-519-2465-6.

[Boz+21] Darko Bozhinoski, Esther Aguado, Carlos Hernández Corbato, Mario Garzon Oviedo, Ricardo Sanz, and Andrzej Wąsowski. "A Modeling Tool for Reconfigurable Skills in ROS". In: *Proceedings of the 3rd International Workshop on Robotics Software Engineering (RoSE'21) Co-located with the 43rd International Conference on Software Engineering (ICSE 2021)*. Virtual, June 2021.

[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+17] Sebastian Brunner, Franz Steinmetz, Rico Belder, and Andreas Dömel. "RAFCON: a Graphical Tool for Task Programming and Mission Control". In: *RoboCup 2016: Robot World Cup XX*. Springer International Publishing, May 2017. ISBN: 978-3-319-68792-6.

[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.

[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).

[Bus+96] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stal. *Pattern-Oriented Software Architecture - Volume 1: A System of Patterns*. Wiley Publishing, 1996. ISBN: 978-0-471-95869-7.

[CamDict] *Cambridge Online Dictionary*.
URL: https://dictionary.cambridge.org/dictionary/english/ (visited: Mar. 1, 2021).

[Che76] Peter Pin-Shan Chen. "The Entity-Relationship Model - toward a Unified View of Data". In: *ACM Trans. Database Syst.* 1.1 (Mar. 1976), pp. 9–36. ISSN: 0362-5915.
DOI: 10.1145/320434.320440.
URL: https://doi.org/10.1145/320434.320440.

[CÖ17] M. Colledanchise and P. Ögren. "How Behavior Trees Modularize Hybrid Control Systems and Generalize Sequential Behavior Compositions, the Subsumption Architecture, and Decision Trees". In: *IEEE Transactions on Robotics* 33.2 (Apr. 2017), pp. 372–389. ISSN: 1552-3098.
DOI: 10.1109/TRO.2016.2633567.

[CÖ18] Michele Colledanchise and Petter Ögren. *Behavior Trees in Robotics and AI: An Introduction*. Vol. abs/1709.00084. CRC Press, July 2018. ISBN: 9781138593732.
DOI: 10.1201/9780429489105. arXiv: 1709.00084.
URL: http://arxiv.org/abs/1709.00084.

[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.

[Cro] Douglas Crockford. *JSON - JavaScript Object Notation*.
URL: http://www.json.org (visited: Mar. 1, 2021).

[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.

[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: *Third International Conference on Simulation, Modeling, and Programming for Autonomous Robots (SIMPAR'12)*. Ed. by I. Noda, N. Ando, D. Brugali, and J. J. Kuffner. Lecture Notes in Computer Science. Tsukuba, Japan: Springer Berlin Heidelberg, Nov. 2012. ISBN: 978-3-642-34326-1.

[Emf] William Emfinger. *Hierarchical Finite State Machine (HFSM) Design Studio*.
URL: https://cps-vo.org/group/hfsm (visited: Mar. 1, 2021).

[Erl05] Thomas Erl. *Service-Oriented Architecture: Concepts, Technology, and Design*. Upper Saddle River, NJ, USA: Prentice Hall PTR, 2005. ISBN: 978-0-13-185858-9.

[Erl07] Thomas Erl. *SOA Principles of Service Design*. Upper Saddle River, NJ, USA: Prentice Hall PTR, 2007. ISBN: 0132344823.

[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.

[ES95] Gregor Engels and Andy Schürr. “Encapsulated Hierarchical Graphs, Graph Types, and Meta Types”. In: *Electronic Notes in Theoretical Computer Science* 2 (1995). SEGRAGRA 1995, Joint COMPUGRAPH/SEMAGRAPH Workshop on Graph Rewriting and Computation, pp. 101–109. ISSN: 1571-0661.
DOI: https://doi.org/10.1016/S1571-0661(05)80186-0.

[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.

[Faca] Davide Faconti. *BehaviorTree.CPP*.
URL: https://www.behaviortree.dev (visited: Mar. 1, 2021).

[Facb] Davide Faconti. *Groot*.
URL: https://github.com/BehaviorTree/Groot (visited: Mar. 1, 2021).

[Fac+18] Davide Faconti, Dennis Stampfer, **Matthias Lutz**, Alex Lotz, and Christian Schlegel. *Robotic Behavior in RobMoSys using Behavior Trees and the SmartMDSD Toolchain (MOOD2be ITP)*. Video. Dec. 2018.
URL: https://www.youtube.com/watch?v=_EwNZG5Xo1k (visited: Apr. 1, 2021).

[FHC97] Sara Fleury, Matthieu Herrb, and Raja Chatila. “GenoM: A Tool for the Specification and the Implementation of Operating Modules in a Distributed Robot Architecture”. In: *Proceedings of the 2007 IEEE/RSJ International Conference on Intelligent Robots and Systems*. Vol. 2. San Diego, CA, USA, July 1997, pp. 842–848.
DOI: 10.1016/S0920-5489(99)90856-5.

[Fie00] Roy Thomas Fielding. “REST: Architectural Styles and the Design of Network-based Software Architectures”. Doctoral dissertation. University of California, Irvine, 2000.
URL: http://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm.

[Fir89] Robert James Firby. “Adaptative Execution in Complex Dynamic Worlds”. PhD thesis. New Haven, CT, USA: Yale University, Jan. 1989.

[Foua] Eclipse Foundation. *Eclipse Modeling Framework (EMF)*.
URL: https://www.eclipse.org/modeling/emf/ (visited: Mar. 1, 2021).

[Foub] Eclipse Foundation. *Eclipse Project: Eclipse SmartMDSD*.
URL: https://projects.eclipse.org/projects/modeling.smartmdsd/ (visited: Mar. 1, 2021).

[Fouc] Eclipse Foundation. *EcoreTools – Graphical Modeling for Ecore*. (visited: Nov. 11, 2019).
URL: http://www.eclipse.org/ecoretools/ (visited: Mar. 1, 2021).

[Foud] Eclipse Foundation. *Xtext Website*.
URL: http://www.eclipse.org/Xtext/ (visited: Mar. 1, 2021).

[FOW14] Martin Fowler. *Microservices*. 2014.
URL: http://martinfowler.com/articles/microservices.html (visited: Mar. 1, 2021).

[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.

[GB11] Luca Gherardi and Davide Brugali. "An eclipse-based Feature Models toolchain". In: *Proceedings 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: *Proceedings of the 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).

[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.

[Ham+19a] N. Hammoudeh Garcia, L. Deval, M. Lüdtke, A. Santos, B. Kahl, and M. Bordignon. "Bootstrapping MDE Development from ROS Manual Code - Part 2: Model Generation". In: *Proceedings of the 2019 ACM/IEEE 22nd International Conference on Model Driven Engineering Languages and Systems (MODELS)*. Munich, Germany: IEEE, Sept. 2019, pp. 95–105.
DOI: 10.1109/MODELS.2019.00-11.

[Ham+19b] N. Hammoudeh Garcia, M. Lüdtke, S. Kortik, B. Kahl, and M. Bordignon. "Bootstrapping MDE Development from ROS Manual Code - Part 1: Metamodeling". In: *Proceedings of the 2019 Third IEEE International Conference on Robotic Computing (IRC)*. Naples, Italy: IEEE, Feb. 2019, pp. 329–336. DOI: 10.1109/IRC.2019.00060.

[Har87] David Harel. "Statecharts: A visual formalism for complex systems". In: *Science of computer programming* 8.3 (1987), pp. 231–274. ISSN: 0167-6423. DOI: 10.1016/0167-6423(87)90035-9.

[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: Mar. 8, 2021).

[Hen96] Thomas. A. Henzinger. "The Theory of Hybrid Automata". In: *Proceedings of the 11th Annual IEEE Symposium on Logic in Computer Science*. New Brunswick, NJ, USA, July 1996, pp. 278–292.
DOI: 10.1109/LICS.1996.561342.

[HN96] David Harel and Amnon Naamad. "The STATEMATE Semantics of Statecharts". In: *ACM Trans. Softw. Eng. Methodol.* 5.4 (Oct. 1996), pp. 293–333. ISSN: 1049-331X.
DOI: 10.1145/235321.235322.
URL: https://doi.org/10.1145/235321.235322.

[IDC96] Roberto Ierusalimschy, Luiz Henrique De Figueiredo, and Waldemar Celes Filho. "Lua - An Extensible Extension Language". In: *Softw. Pract. Exper.* 26.6 (June 1996), pp. 635–652. ISSN: 0038-0644.
DOI: 10.1002/(SICI)1097-024X(199606)26:6<635::AID-SPE26>3.0.CO;2-P.
URL: http://dx.doi.org/10.1002/(SICI)1097-024X(199606)26:6%3C635::AID-SPE26%3E3.0.CO;2-P.

[IL04] Marco Iansiti and Roy Levien. "Strategy as Ecology". In: *Harvard Business Review* 82.3 (Mar. 2004). Reprint R0403E, pp. 68–81.

[Iov+20] Matteo Iovino, Edvards Scukins, Jonathan Styrud, Petter Ogren, and Christian Smith. "A Survey of Behavior Trees in Robotics and AI". Unpublished - Preprint. May 2020.

URL: https://www.researchgate.net/publication/341341939_A_Survey_of_Behavior_Trees_in_Robotics_and_AI.

[Isl05] Damian Isla. *Handling Complexity in the Halo 2 AI*. 2005.
URL: https://www.gamasutra.com/view/feature/130663/gdc_2005_proceeding_handling_.php?print=1 (visited: Apr. 1, 2021).

[JA11] Sylvain Joyeux and Jan Albiez. "Robot development: from components to systems". In: *Proceedings of the 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. Part1 URL: http://www.it-radar.org/serendipity/archives/101-Software-OEkosysteme-Teil-1.html, Internet Archive URL: http://web.archive.org/web/20160915205922/http://www.it-radar.org/serendipity/archives/101-Software-OEkosysteme-Teil-1.html, Part2: http://www.it-radar.org/serendipity/archives/102-Software-OEkosysteme-Teil-2.html, Part3: http://www.it-radar.org/serendipity/archives/103-Software-OEkosysteme-Teil-3.html, Part4: http://www.it-radar.org/serendipity/archives/104-Software-OEkosysteme-Teil-4.html, (visited: Mai. 11, 2021). 2012.

[Jet16] JetBrains s.r.o. *JetBrains Meta Programming System (MPS) Website*. 2016.
URL: https://www.jetbrains.com/mps/ (visited: Apr. 1, 2021).

[JFB09] Slinger Jansen, Anthony Finkelstein, and Sjaak Brinkkemper. "A Sense of Community: A Research Agenda for Software Ecosystems". In: *Proceedings of the 31st International Conference on Software Engineering (ICSE)*. Vancouver, Canada, May 2009, pp. 187–190.
DOI: 10.1109/ICSE-COMPANION.2009.5070978.

[JKL10] Sylvain Joyeux, Frank Kirchner, and Simon Lacroix. "Managing plans Integrating deliberation and reactive execution schemes". In: *Robotics and Autonomous Systems (RAS)* 58.9 (Sept. 2010), pp. 1057–1066.

[Kat20] Kathrin Evers and Jan R. Seyler. *SS04 -SEnSEI - Skill Based Systems Engineering*. Workshop of the 25th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA). Vienna, Austria, Sept. 2020.

[KE19] Pranav Srinivas Kumar and William Emfinger. "Model-Driven Software Design Automation for Complex Rehabilitation". In: *Design Automation of Cyber-Physical Systems*. Ed. by Mohammad Abdullah Al Faruque and Arquimedes Canedo. Cham: Springer International Publishing, 2019, pp. 211–235. ISBN: 978-3-030-13050-3.

DOI: 10.1007/978-3-030-13050-3_8.
URL: https://doi.org/10.1007/978-3-030-13050-3_8.

[KMH13] Friedrich Kirchner, Karl Theodor Michaelis, and Johannes Hoffmeister. *Wörterbuch der philosophischen Begriffe*. Ed. by Arnim Regenbogen. Jubiläumsausgabe zum 150jährigen Bestehen der "Philosophischen Bibliothek". Philosophische Bibliothek. Hamburg: Felix Meiner Verlag, 2013. ISBN: 978-3-7873-3150-5.
DOI: 10.28937/978-3-7873-2113-1.

[KSB10] Markus Klotzbücher, Peter Soetens, and Herman Bruyninckx. "OROCOS RTT-Lua: an Execution Environment for building Real-time Robotic Domain Specific Languages". In: *Proceedings of the SIMPAR 2010 Workshops*. Darmstadt, Germany, Nov. 2010, pp. 284–289. ISBN: 978-3-00-032863-3.

[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: *Proceedings of the 2015 International Symposium on Rapid System Prototyping (RSP)*. Amsterdam, Netherlands, Oct. 2015, pp. 39–45.
DOI: 10.1109/RSP.2015.7416545.

[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.

[Ley01] Frank Leymann. *Web Services Flow Language (WSFL 1.0)*. May 2001.
URL: http://xml.coverpages.org/WSFL-Guide-200110.pdf (visited: Mar. 8, 2021).

[Lot+14] Alex Lotz, Juan F. Inglés-Romero, Dennis Stampfer, **Matthias Lutz**, Cristina Vicente-Chicote, and Christian Schlegel. "Towards a Stepwise Variability Management Process for Complex Systems: A Robotics Perspective". In: *International Journal of Information System Modeling and Design (IJISMD)* 5.3 (2014), 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: *Proceedings of the 6th International Workshop on Domain-Specific Languages and Models for Robotic Systems (DSLRob 2015)*. Hamburg, Germany, 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: *Proceedings of the 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.

[LP91] M. Levene and A. Poulovassilis. "An object-oriented data model formalised through hypergraphs". In: *Data & Knowledge Engineering* 6.3 (1991), pp. 205–224. ISSN: 0169-023X.
DOI: https://doi.org/10.1016/0169-023X(91)90005-I.

[LSL18] Zeljko Loncaric, Christian Schlegel, and **Matthias Lutz**. "Module für autonome kooperative und kollaborative Roboter." In: *Elektronik Praxis – Embedded System Development + IOT II* (Sept. 2018), pp. 42–44.
URL: https://www.elektronikpraxis.vogel.de/wie-universelle-module-die-entwicklung-von-robotern-beschleunigen-a-746387/.

[LSS13] **Matthias Lutz**, Dennis Stampfer, and Christian Schlegel. "Probabilistic Object Recognition and Pose Estimation by Fusing Multiple Algorithms". In: *Proceedings of the IEEE International Conference on Robotics and Automation 2013 (ICRA)*. Karlsruhe, Germany, May 2013, pp. 4244–4249.
DOI: 10.1109/ICRA.2013.6631177.

[Lub08] Boris Lublinsky. *InfoQ: Orchestration vs. Choreography: Debate Over Definitions*. Sept. 2008.
URL: https://www.infoq.com/news/2008/09/Orchestration/ (visited: Mar. 1, 2021).

[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: Mar. 8, 2021).

[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: *Proceedings of the 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+18a] **Matthias Lutz**, Dennis Stampfer, Alex Lotz, and Christian Schlegel. *RobMoSys webpage: Architectural Pattern for Task-Plot Coordination (Robotic Behaviors)*. Dec. 2018.
URL: https://robmosys.eu/wiki/general_principles:architectural_patterns:robotic_behavior (visited: Apr. 1, 2021).

[Lut+18b] **Matthias Lutz**, Dennis Stampfer, Alex Lotz, and Christian Schlegel. *RobMoSys webpage: Skills for Robotic Behavior*. Dec. 2018.
URL: https://robmosys.eu/wiki/composition:skills:start (visited: Apr. 1, 2021).

[Lut+19a] **Matthias Lutz**, J. Inglés-Romero, Dennis Stampfer, Alex Lotz, Cristina Vicente-Chicote, and Christian Schlegel. "Managing Variability as a Means to Promote Composability: A Robotics Perspective". In: *New Perspectives on Information Systems Modeling and Design*. Ed. by António Miguel Rosado da Cruz and Maria Estrela Ferreira da Cruz. IGI Global, Nov. 2019, pp. 274–295. ISBN: 978-1-52-257271-8.
DOI: 10.4018/978-1-5225-7271-8.ch012.

[Lut+19b] **Matthias Lutz**, Dennis Stampfer, Johannes Baumgartl, and Jochen Kirschbaum. *Gradual Automation of an Assembly Line*. Video. Dec. 2019.
URL: https://www.youtube.com/watch?v=Nzwjb8BaQns (visited: Mar. 8, 2021).

[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: *Proceedings of the 2016 IEEE 21st International Conference on Emerging Technologies and Factory Automation (ETFA)*. Berlin, German, Sept. 2016, pp. 1–4.
DOI: 10.1109/ETFA.2016.7733602.
URL: http://ieeexplore.ieee.org/document/7733602/.

[Mar19] Bernard Marr. *Robots As A Service: A Technology Trend Every Business Must Consider*. 2019.
URL: https://www.forbes.com/sites/bernardmarr/2019/08/05/robots-as-a-service-a-technology-trend-every-business-must-consider/?sh=c13bccb24ea2 (visited: Apr. 1, 2021).

[MC94] Thomas W. Malone and Kevin Crowston. "The Interdisciplinary Study of Coordination". In: *ACM Computing Surveys* 26.1 (Mar. 1994), pp. 87–119. ISSN: 0360-0300.
DOI: 10.1145/174666.174668.
URL: http://doi.acm.org/10.1145/174666.174668.

[Mcd93] Drew Mcdermott. *A Reactive Plan Language*. Tech. rep. Yale University, 1993.

[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.

[MHS05] Marjan Mernik, Jan Heering, and Anthony M. Sloane. "When and How to Develop Domain-Specific Languages". In: *ACM Comput. Surv.* 37.4 (Dec. 2005), pp. 316–344. ISSN: 0360-0300.
DOI: 10.1145/1118890.1118892.
URL: https://doi.org/10.1145/1118890.1118892.

[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.

[MPA] Eitan Marder-Eppstein, Vijay Pradeep, and Mikael Arguedas. *actionlib*.
URL: http://wiki.ros.org/actionlib (visited: Mar. 1, 2021).

[MS02] M. Mateas and A. Stern. "A behavior language for story-based believable agents". In: *IEEE Intelligent Systems* 17.4 (2002), pp. 39–47.
DOI: 10.1109/MIS.2002.1024751.

[Mül+20] Christopher Müller, Birgit Graf, Kai Pfeiffer, Susanne Bieller, Nina Kutzbach, and Karin Röhricht. *World Robotics 2020 – Service Robots*. IFR Statistical Department, VDMA Services GmbH, Frankfurt am Main, Germany, 2020. ISBN: 978-3-8163-0740-2.

[Näg+18] Frank Nägele, Lorenz Halt, Philipp Tenbrock, and Andreas Pott. "A prototype-based skill model for specifying robotic assembly tasks". In: *Proceedings of the 2018 IEEE International Conference on Robotics and Automation (ICRA)*. Brisbane, Australia, May 2018, pp. 558–565.
DOI: 10.1109/ICRA.2018.8462885.

[NFL10] Tim Niemüller, Alexander Ferrein, and Gerhard Lakemeyer. "A Lua-based Behavior Engine for Controlling the Humanoid Robot Nao". In: *RoboCup 2009: Robot Soccer World Cup XIII*. Vol. 5949. Berlin, Heidelberg, June 2010, pp. 240–251. ISBN: 978-3-642-11876-0.
DOI: 10.1007/978-3-642-11876-0_21.

[Ngu+13] H. Nguyen, M. Ciocarlie, K. Hsiao, and C. C. Kemp. "ROS Commander (ROSCo): Behavior Creation for Home Robots". In: *Proceedings of the 2013 IEEE International Conference on Robotics and Automation*. Karlsruhe, Germany, May 2013, pp. 467–474.
DOI: 10.1109/ICRA.2013.6630616.

[Nie+10] Tim Niemueller, Alexander Ferrein, Daniel Beck, and Gerhard Lakemeyer. "Design Principles of the Component-Based Robot Software Framework Fawkes". In: *Proceedings of the Second International Conference on Simulation, Modeling, and Programming for Autonomous Robots (SIMPAR'10)*. Lecture Notes in Computer Science. Darmstadt, Germany: Springer, 2010.

[Nie09] Tim Niemüller. "Developing A Behavior Engine for the Fawkes Robot-Control Software and its Adaptation to the Humanoid Platform Nao". MA thesis. RWTH Aachen University, Knowledge-Based Systems Group, Apr. 2009.

[NLH15] Aditya Narayanamoorthy, Renjun Li, and Zhiyong Huang. "Creating ROS launch files using a visual programming interface". In: *Proceedings of the 2015 IEEE 7th International Conference on Cybernetics and Intelligent Systems (CIS) and IEEE Conference on Robotics, Automation and Mechatronics (RAM)*. Siem Reap, Cambodia, July 2015, pp. 142–146.
DOI: 10.1109/ICCIS.2015.7274563.

[NLM21] Arne Nordmann, Ralph Lange, and Francisco Martín. "System Modes - Digestible System (Re-)Configuration for Robotics". Unpublished - Preprint. June 2021.
URL: https://www.researchgate.net/publication/348872908_System_Modes_-_Digestible_System_Re-Configuration_for_Robotics.

[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.
URL: https://joser.unibg.it/index.php?journal=joser&page=article&op=view&path%5B%5D=100&path%5B%5D=0.

[OAS07] OASIS Web Services Business Process Execution Language (WSBPEL) TC. *Web Services Business Process Execution Language Version 2.0 - OASIS Standard*. Apr. 2007.
URL: http://docs.oasis-open.org/wsbpel/2.0/OS/wsbpel-v2.0-OS.html (visited: Mar. 8, 2021).

[OB20] Helena Olsson and Jan Bosch. "Going digital: Disruption and transformation in software-intensive embedded systems ecosystems". In: *Journal of Software: Evolution and Process* 32 (Jan. 2020), e2249.
DOI: 10.1002/smr.2249.

[OroRTT] *Orocos Real-Time Toolkit Website*.
URL: http://www.orocos.org/rtt (visited: Apr. 23, 2021).

[Papy4Rob] *Papyrus4Robotics*.
URL: https://www.eclipse.org/papyrus/components/robotics/ (visited: Apr. 1, 2021).

[Papyrus] *Papyrus UML*.
URL: http://www.eclipse.org/papyrus/ (visited: Apr. 1, 2021).

[Pro19a] RobMoSys Project. *RobMoSys CARVE Integrated Technical Project*. 2019.
URL: https://robmosys.eu/carve/ (visited: Apr. 1, 2021).

[Pro19b] RobMoSys Project. *RobMoSys MOOD2Be Integrated Technical Project*. Dec. 2019.
URL: https://robmosys.eu/mood2be/ (visited: Apr. 1, 2021).

[Pro19c] RobMoSys Project. *RobMoSys webpage: Block-Port-Connector*. May 2019.
URL: https://robmosys.eu/wiki-sn-03/modeling:principles:block-port-connector (visited: Apr. 1, 2021).

[Pro19d] RobMoSys Project. *RobMoSys webpage: Glossary*. May 2019.
URL: https://robmosys.eu/wiki/glossary (visited: Apr. 1, 2021).

[Pro19e] RobMoSys Project. *RobMoSys webpage: Robotic Behavior in RobMoSys using Behavior Trees and the SmartMDSD Toolchain (MOOD2be ITP)*. May 2019.
URL: https://robmosys.eu/wiki/community:behavior-tree-demo:start (visited: Apr. 1, 2021).

[PW03] Mikel D. Petty and Eric W. Weisel. "A Composability Lexicon". In: *Proceedings of the Spring 2003 Simulation Interoperability Workshop*. 03S-SIW-023. Orlando, USA, 2003, pp. 181–187.

[Pytrees] *Py_Trees*.
URL: https://github.com/splintered-reality/py_trees (visited: Apr. 1, 2021).

[Qui+09] Morgan Quigley, Ken Conley, Brian Gerkey, Josh Faust, Tully Foote, Jeremy Leibs, Rob Wheeler, and Andrew Ng. "ROS: an open-source Robot Operating System". In: *Proceedings of the ICRA Workshop on Open Source Software*. Kobe, Japan, Jan. 2009.

[RobMoSys] *RobMoSys Project Website*.
URL: http://www.robmosys.eu (visited: Apr. 1, 2021).

[Rock] *Rock, the Robot Construction Kit*.
URL: http://www.rock-robotics.org (visited: Apr. 20, 2021).

[Rol+19] Matthias Rollenhagen, **Matthias Lutz**, Nayabrasul Shaik, Kevin Andrews, Sebastian Steinau, Manfred Reichert, and Christian Schlegel. "Towards Flexible Process Automation An Approach for Flexible Service Robot Adaptation and Allocation". In: *Proceedings of the 2019 3rd International*

Symposium on Computer Science and Intelligent Control. Amsterdam, Netherlands, Sept. 2019.
DOI: 10.1145/3386164.3387292.
URL: https://doi.org/10.1145/3386164.3387292.

[Ros] ros.org. *ROS - Concepts*.
URL: http://wiki.ros.org/ROS/Concepts (visited: Mar. 1, 2021).

[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: Apr. 1, 2021).

[ROS20] ROS Community. *ROS wiki - IDEs*. 2020.
URL: http://wiki.ros.org/IDEs (visited: Apr. 1, 2021).

[Rov+17] Francesco Rovida, Matthew Crosby, Dirk Holz, Athanasios S. Polydoros, Bjarne Großmann, Ronald P. A. Petrick, and Volker Krüger. "SkiROS - A Skill-Based Robot Control Platform on Top of ROS". In: *Robot Operating System (ROS): The Complete Reference (Volume 2)*. Ed. by Anis Koubaa. Cham: Springer International Publishing, 2017, pp. 121–160. ISBN: 978-3-319-54927-9.
DOI: 10.1007/978-3-319-54927-9_4.
URL: https://doi.org/10.1007/978-3-319-54927-9_4.

[SA98] Reid Simmons and David Apfelbaum. "Task description language for robot control". In: *Proceedings of the 1998 IEEE/RSJ International Conference on Intelligent Robots and Systems. Innovations in Theory, Practice and Applications (Cat. No.98CH36190)*. Vol. 3. Nov. 1998, pp. 1931–1937. ISBN: 0-7803-4465-0.
DOI: 10.1109/IROS.1998.724883.

[San+17] Higor Barbosa Santos, Marco Antônio Simões Teixeira, André Schneider de Oliveira, Lúcia Valéria Ramos de Arruda, and Flávio Neves. "Control of Mobile Robots Using ActionLib". In: *Robot Operating System (ROS): The Complete Reference (Volume 2)*. Ed. by Anis Koubaa. Cham: Springer International Publishing, 2017, pp. 161–189. ISBN: 978-3-319-54927-9.
DOI: 10.1007/978-3-319-54927-9_5.
URL: https://doi.org/10.1007/978-3-319-54927-9_5.

[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: *Proceedings of the 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.

[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.

[Sch+18a] Christian Schlegel, **Matthias Lutz**, Dennis Stampfer, and Alex Lotz. *RobMoSys webpage: Communication-Object Metamodel*. Dec. 2018.
URL: https://robmosys.eu/wiki/modeling:metamodels:commobject (visited: Apr. 1, 2021).

[Sch+18b] Christian Schlegel, Dennis Stampfer, Alex Lotz, and **Matthias Lutz**. *RobMoSys webpage: Component-Definition Metamodel*. Dec. 2018.
URL: https://robmosys.eu/wiki/modeling:metamodels:component (visited: Apr. 1, 2021).

[Sch+18c] Christian Schlegel, Dennis Stampfer, Alex Lotz, and **Matthias Lutz**. *RobMoSys webpage: Service-Definition Metamodel*. Dec. 2018.
URL: https://robmosys.eu/wiki/modeling:metamodels:service (visited: Apr. 1, 2021).

[Sch+20] Christian Schlegel, Dennis Stampfer, Alex Lotz, and **Matthias Lutz**. "Robot Programming". In: *Mechatronics and Robotics: New Trends and Challenges*. Ed. by Marina Indri and Roberto Oboe. CRC Press, 2020. Chap. 8. ISBN: 9780429347474.
DOI: 10.1201/9780429347474.

[Sch+21] Christian Schlegel, Alex Lotz, **Matthias Lutz**, and Dennis Stampfer. "Composition, Separation of Roles and Model-Driven Approaches as Enabler of a Robotics Software Ecosystem". In: *Software Engineering for Robotics*. Ed. by Ana Cavalcanti, Jon Timmis, Brijesh Dongol, Rob Hierons, and Jim Woodcock. Springer Nature Switzerland, 2021. Chap. 3.

[Sch04] Christian Schlegel. "Navigation and Execution for Mobile Robots in Dynamic Environments: An Integrated Approach". PhD thesis. University of Ulm, 2004.

[Sch98] Christian Schlegel. "Fast local obstacle avoidance under kinematic and dynamic constraints for a mobile robot". In: *Proceedings of the 1998 IEEE/RSJ International Conference on Intelligent Robots and Systems. Innovations in*

Theory, Practice and Applications (Cat. No.98CH36190). Vol. 1. Victoria, BC, Canada, Nov. 1998, pp. 594–599.
DOI: 10.1109/IROS.1998.724683.

[Ser] Servicerobotik Ulm. *Servicerobotik Ulm Website*.
URL: http://www.servicerobotik-ulm.de (visited: Dec. 27, 2016).

[SL19] Nayabrasul Shaik and **Matthias Lutz**. *SmartMDSD Toolchain - Modifying a Robot System: Exchanging Sensor*. Video. May 2019.
URL: https://www.youtube.com/watch?v=RHvvb6lTHG4 (visited: Mar. 8, 2021).

[SLS11] Christian Schlegel, Alex Lotz, and Andreas Steck. *SmartSoft: The State Management of a Component*. Tech. rep. ISSN: 1868-3452. University of Applied Sciences Ulm, Jan. 2011.
URL: http://www.servicerobotik-ulm.de/drupal/sites/default/files/ZAFH-TR-01-2010-ISSN-1868-3452.pdf.

[SLS12] Dennis Stampfer, **Matthias Lutz**, and Christian Schlegel. "Information Driven Sensor Placement for Robust Active Object Recognition based on Multiple Views". In: *Proceedings of the 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.

[Soe+] Peter Soetens, Takis Issaris, Herman Bruyninckx, Sylvain Joyeux, and Ruben Smits. *Orocos Project documentation*.
URL: https://docs.orocos.org (visited: Mar. 1, 2021).

[Sof] RWTH Aachen Software Engineering Group. *The MontiCore Language Workbench*.
URL: http://www.monticore.de/ (visited: Apr. 1, 2021).

[Sri+12] Siddhartha S. Srinivasa, Dmitry Berenson, Maya Cakmak, Alvaro Collet, Mehmet R. Dogar, Anca D. Dragan, Ross A. Knepper, Tim Niemueller, Kyle Strabala, Mike Vande Weghe, and Julius Ziegler. "HERB 2.0: Lessons Learned from Developing a Mobile Manipulator for the Home". In: *Proceedings of the IEEE*. 8th ser. 100 (Aug. 2012), pp. 2410–2428.

[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 the SIMPAR 2010 Workshops (International Workshop on Dynamic languages for RObotic and Sensors systems (DYROS)), 2nd Intl. Conf. on Simulation, Modeling, and Programming for Autonomous Robots*. ISBN: 978-3-00-0328. Darmstadt, 2010, pp. 274–277.

[SS13] Dennis Stampfer and Christian Schlegel. "Dynamic State Charts: Composition and Coordination of Complex Robot Behavior and Reuse of Action Plots". In: *Proceedings of the 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.

[SS14] 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.

[SSS09] Andreas Steck, Dennis Stampfer, and Christian Schlegel. "Modellgetriebene Softwareentwicklung für Robotiksysteme". In: *Proceedings of the 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.

[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 (Aug. 2016), pp. 3–19. ISSN: 2035-3928.
URL: https://www.researchgate.net/publication/305723618_The_SmartMDSD_Toolchain_An_Integrated_MDSD_Workflow_and_Integrated_Development_Environment_IDE_for_Robotics_Software.

[Sta18] Dennis Stampfer. "Contributions to System Composition using a System Design Process driven by Service Definitions for Service Robotics". Dissertation. München: Technische Universität München, 2018.
URL: https://nbn-resolving.org/urn/resolver.pl?urn:nbn:de:bvb:91-diss-20180425-1399658-1-2.

[Ste+08] Dave Steinberg, Frank Budinsky, Marcelo Paternostro, and Ed Merks. *EMF Eclipse Modeling Framework*. Ed. by Erich Gamma, Lee Nackman, and John Wiegand. Second. The Eclipse Series. Addison Wesley, 2008. ISBN: 978-0-321-33188-5.

[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: Mar. 1, 2021).

[SW99] Christian Schlegel and Robert Wörz. "The Software Framework SMARTSOFT for Implementing Sensorimotor Systems". In: *Proceedings of the 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*. Second. ISBN: 0-201-74572-0. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2002. ISBN: 0201745720.

[Vic+18] Cristina Vicente-Chicote, J. Inglés-Romero, Jesús Martínez, Dennis Stampfer, Alex Lotz, **Matthias Lutz**, and Christian Schlegel. "A Component-Based and Model-Driven Approach to Deal with Non-Functional Properties through Global QoS Metrics". In: *Proceedings of the ModComp'18 – 5th International Workshop on Interplay of Model-Driven and Component-Based Software Engineering (in conjunction with ACM/IEEE 21st Int. Conf. on Model Driven Engineering Languages and Systems (MODELS))*. Copenhagen, Denmark, Oct. 2018.

[VKG17] Hulya Vural, Murat Koyuncu, and Sinem Guney. "A Systematic Literature Review on Microservices". In: *Computational Science and Its Applications – ICCSA 2017*. Trieste, Italy: Springer International Publishing, 2017, pp. 203–217.
DOI: 10.1007/978-3-319-62407-5_14.

[Vli98] John Vlissides. *Pattern Hatching: Design Patterns Applied*. The software patterns series. Addison-Wesley Professional, 1998. ISBN: 9780201432930.
URL: http://www.informit.com/store/pattern-hatching-design-patterns-applied-9780201432930.

[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.

[W3C04] David Burdett and Nickolas Kavantzas. *WS Choreography Model Overview*. 2004.
URL: https://www.w3.org/TR/ws-chor-model/ (visited: Mar. 1, 2021).

[Wäc+16] Mirko Wächter, Simon Ottenhaus, Manfred Kröhnert, Nikolaus Vahrenkamp, and Tamim Asfour. "The ArmarX Statechart Concept: Graphical Programing of Robot Behavior". In: *Frontiers in Robotics and AI* 3 (June

2016).
DOI: 10.3389/frobt.2016.00033.

[WDW20] Dennis Wigand, Niels Dehio, and Sebastian Wrede. "Model-Based Specification of Control Architectures for Compliant Interaction with the Environment". In: *Proceedings of the 2020 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS 2020)*. Las Vegas, Nevada, USA, Oct. 2020.
DOI: 10.1109/IROS45743.2020.9340718.

[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: *Proceedings of the 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.2016.7733559.

[Whi76] James E. White. *A High-Level Framework for Network-Based Resource Sharing*. RFC 707. RFC Editor, Jan. 1976.
URL: https://tools.ietf.org/html/rfc707.

[Wir17] Bundesministerium für Wirtschaft und Energie (BMWi). *Projekt 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: Mar. 1, 2021).

[ZAFH] ZAFH Servicerobotik. *Project ZAFH Servicerobotik: Autonome mobile Serviceroboter*.
URL: http://www.zafh-servicerobotik.de/ (visited: Jan. 5, 2021).

[ZAFHI] ZAFH Intralogistik. *Project ZAFH Intralogistik: Kollaborative Systeme zur Flexibilisierung der Intralogistik*.
URL: http://zafh-intralogistik.de/ (visited: Jan. 5, 2021).

[ZMQ] The ZeroMQ authors. *ZeroMQ - An open-source universal messaging library*.
URL: https://zeromq.org (visited: Mar. 1, 2021).