

# **RealCaPP**

## Real-Time Capable Plug & Produce for Distributed Robot-Based Automation

Christian Eymüller

**DISSERTATION**  
zur Erlangung des akademischen Grades  
Doktor der Naturwissenschaften (Dr. rer. nat.)



Fakultät für Angewandte Informatik

November 2023

**RealCaPP**

Real-Time Capable Plug & Produce for Distributed Robot-Based Automation

Erstgutachter:	Prof. Dr. Wolfgang Reif
Zweitgutachter:	Prof. Dr. Bernhard Bauer
Drittgutachter:	Prof. Dr.-Ing. Dr. h.c. mult. Alexander Verl

Tag der mündlichen Prüfung: 28. November 2023





# Danksagung

Diese Arbeit entstand im Rahmen meiner Tätigkeit als wissenschaftlicher Mitarbeiter am Institut für Software & Systems Engineering der Universität Augsburg. Ich möchte an dieser Stelle den vielen Personen, die mich unterstützt haben, meinen herzlichen Dank aussprechen, ohne deren Hilfe die Durchführung dieser Arbeit nicht möglich gewesen wäre.

An erster Stelle möchte ich mich bei meinem Doktorvater Prof. Dr. Wolfgang Reif bedanken. Ihm gebührt Dank für seine Unterstützung, das entgegengebrachte Vertrauen, die interessanten Forschungsthemen und das zur Verfügung gestellte Forschungsumfeld. Des Weiteren danke ich Prof. Dr. Bernhard Bauer und Prof. Dr.-Ing. Dr. h.c. mult. Alexander Verl für die Begutachtung der Dissertation.

Mein Dank gilt auch meinen Kolleginnen und Kollegen am Institut für die Mithilfe während der Promotion. Insbesondere möchte ich dem Robotik-Team danken, das durch produktive Diskussionen und Anregungen zu vielen guten Ideen beigetragen hat. Ich möchte mich bei Daniel Bermuth, Michael Filipenko, Dr. Johannes Kurth, Dr. Andreas Schierl, Martin Schörner, Matthias Stüben und Constantin Wanninger bedanken. Insbesondere möchte ich mich bei Alexander Poeppel für die vielen produktiven Gespräche bei den "Spezi-Touren" bedanken und dafür, dass er den Publikationen immer den letzten englischen Schliff gegeben hat. Bei allen technischen Problemen in den Roboteranlagen war unser Techniker Stefan Wolff immer eine große Hilfe. Meinem Bürokollegen Julian Hanke danke ich für die vielen fachlichen und nichtfachlichen Gespräche und die gemeinsame Zeit in der WiR Zelle. Ein großes Dankeschön geht auch an Dr. Alwin Hoffmann für seine Unterstützung zu Beginn meiner Promotion. Darüber hinaus möchte ich ihm für das aufmerksame Korrekturlesen meiner Arbeit danken.

Allen Studierenden und wissenschaftlichen Hilfskräften, die mich bei meinen Forschungsarbeiten tatkräftig unterstützt haben, möchte ich danken. Eine große Hilfe waren hier Simon Erlbacher, Leonhard Heber, Lukas Kilian, Markus Kugelmann, Maximilian Müller und Nicolai Sandmann.

Mein besonderer Dank gilt meinen Eltern Luise und Bonifaz Eymüller, die mich stets unterstützt haben und mir immer mit Rat und Tat zur Seite standen. Meine Mutter hat mich während der Schreibphase durch spontane Einladungen zu leckeren Abendessen stets aufgemuntert. Mein Vater hat viel Zeit in intensives Korrekturlesen investiert.

Meine Partnerin Sarah Gamperl hat mich während der kompletten Promotionszeit liebevoll unterstützt. Ihr danke ich besonders für die vielen aufmunternden Worte und ihre Geduld vor allem während der Schreibphase. Sie war immer für mich da, wenn ich zweifelte und hat mich getröstet, wenn ich niedergeschlagen war. Das hat mir viel Kraft gegeben.

Abschließend möchte ich mich bei allen bedanken, die nicht namentlich genannt wurden, mich aber tatkräftig bei der Promotion unterstützt haben.

*Christian Eymüller*



# Abstract

Due to the 4th Industrial Revolution, the industry is currently undergoing a transition from mass production to individual production, where each product is unique. This change in production means that it is no longer possible to design and develop the entire production plant, then build the system and produce the products. It may happen that a product is not yet known at the time when the plant is planned or realized. Therefore, flexible production facilities are needed to realize this transformation of production.

Plug & Produce is a technique used for the realization of flexible production plants. Similar to the Plug & Play approach known from USB devices, for example, where devices are simply plugged in and can then be used. Plug & Produce is attempting to implement this approach for industrial components. Plug & Produce is a technique for quickly integrating production resources into systems and thus being able to produce quickly with these new resources. Production can be flexibly designed through the rapid integration or exchange of new production resources.

The central result of this thesis is the development of a Real-Time Capable Plug & Produce (RealCaPP) architecture. This architecture makes it possible to add and exchange new production resources to a production system. To react quickly to new processes by adapting the software, and to execute these processes in real time distributed across multiple compute nodes. In order to quickly integrate new resources into the production system, a resource self-introduction mechanism has been developed. For this purpose, standardized self-descriptions were defined in a machine-readable form, describing the properties and skills of a resource. These descriptions are aggregated and merged into a global knowledge base of the system. This data can be used, for example, to automatically find plant configurations for specific tasks. Another important point is the exchange of data between product resources. Since process information must be exchanged between real-time-critical processes, the communication must also be real-time-capable. Therefore, a dynamic real-time communication platform was developed that can cope with changing resources and exchange process data in real-time. A real-time service architecture was developed for flexible process adaptation. Modular software components, so-called Real-Time Service (RTS), enable the implementation of resource skills (Basic Skills) and can be combined to form more complex skills (Composed Skills). These RTSs can be easily added to a system at runtime and executed afterwards. They can also be executed in a distributed manner. All these points were evaluated on two robotic cells for different case studies. It was shown that the same services used to screw aluminium profiles could also be reused to operate a portafilter coffee machine with a robot.





# Contents

<b>1</b>	<b>Introduction</b>	<b>1</b>
1.1	Motivation . . . . .	1
1.2	Research Question . . . . .	3
1.3	Main Contributions . . . . .	4
1.4	Structure of the Thesis . . . . .	6
<b>2</b>	<b>Fundamentals</b>	<b>9</b>
2.1	Cyber-Physical Production Systems (CPPSs) . . . . .	9
2.2	Reference Architecture Model Industry 4.0 (RAMI 4.0) . . . . .	10
2.3	Real-Time Systems . . . . .	12
2.3.1	Real-Time Communication . . . . .	14
2.3.2	Real-Time Operating Systems . . . . .	15
2.3.3	Real-Time Applications . . . . .	17
2.4	Distributed Systems . . . . .	18
<b>3</b>	<b>Plug &amp; Produce in a Flexible Distributed Robot Cell</b>	<b>21</b>
<b>4</b>	<b>Concept of a Real-Time Capable Plug &amp; Produce Environment</b>	<b>29</b>
4.1	Fundamentals . . . . .	30
4.1.1	Skills, Capabilities and Services . . . . .	30
4.1.2	Asset Administration Shell . . . . .	30
4.1.3	Open Platform Communication Unified Architecture (OPC UA)	32
4.1.4	Time-Sensitive Networking (TSN) . . . . .	34
4.1.5	OPC UA over TSN . . . . .	35
4.2	Real-Time Capable Plug & Produce . . . . .	35
4.3	RealCaPP Architecture (Real-Time Capable Plug & Produce Architec- ture) . . . . .	36
4.3.1	A Uniform Communication Interface . . . . .	36
4.3.2	Asset Administration Shells for Robot Components . . . . .	37
4.3.3	Modular Real-Time Capable Software Components . . . . .	39
4.3.4	Global Knowledge of Resources, Products and Services . . . . .	39
4.4	Related Work . . . . .	40

<b>5</b>	<b>Locating and Managing Plant Components</b>	<b>43</b>
5.1	Fundamentals . . . . .	45
5.1.1	Semantic Descriptions . . . . .	45
5.1.2	OPC UA Discovery . . . . .	46
5.1.3	OPC UA Extension Objects . . . . .	47
5.2	Plug & Produce Resources . . . . .	47
5.3	Global Registry for Plant Components . . . . .	49
5.4	Semantic Self-Description of Plant Components . . . . .	51
5.5	Consolidation of Information into a Uniform Knowledge Base . . . . .	52
5.6	Related Work . . . . .	54
<b>6</b>	<b>Semantic Descriptions of Automation Plants</b>	<b>57</b>
6.1	Fundamentals . . . . .	58
6.1.1	Semantic Rules . . . . .	59
6.1.2	Querying of Semantic Networks . . . . .	60
6.2	Semantic Description of Resource Interrelationships . . . . .	61
6.3	Derivation of Connectable and Connected Resources . . . . .	63
6.4	Deriving Composed Skills . . . . .	65
6.5	Automatic Plant Configuration through Semantic Networks . . . . .	66
6.6	Related Work . . . . .	70
<b>7</b>	<b>Distributed Real-Time: Dynamic Real-Time Control Networks</b>	<b>73</b>
7.1	Fundamentals . . . . .	74
7.1.1	Real-time Communication with TSN . . . . .	75
7.1.2	Precision-Time-Protocol . . . . .	77
7.2	Time-Synchronization in Control Networks . . . . .	78
7.3	Dynamic Configuration of Real-Time Control Networks . . . . .	79
7.3.1	Setup and Configuration of TSN Communication Channels . . . . .	81
7.3.2	Realization of the Time Slot Array . . . . .	83
7.4	Related Work . . . . .	84
<b>8</b>	<b>Distributed Real-Time Execution of Component Skills in Distributed Control Networks</b>	<b>87</b>
8.1	Fundamentals . . . . .	89
8.1.1	OSGi: A Dynamic Module System . . . . .	89
8.1.2	OPC UA Programs . . . . .	90
8.2	Services: Reusable Software Components . . . . .	91
8.3	Real-Time Critical and Non-Real-Time Critical Execution . . . . .	93
8.4	The Plug & Produce Service Architecture . . . . .	94
8.4.1	A Uniform Data Representation: DataContainer . . . . .	95
8.4.2	Modular Software Components: Real-Time Services . . . . .	97
8.4.3	Distribution of Real-Time Services . . . . .	99
8.4.4	Execution of Applications . . . . .	102
8.4.5	Semantic Description of RTSs and RTS Networks . . . . .	108
8.4.6	Synchronized Distributed Control Processes . . . . .	109

8.5	Related Work . . . . .	115
<b>9</b>	<b>Implementation of Resources with the RealCaPP Service Architecture</b>	<b>119</b>
9.1	Industrial Robot Resources . . . . .	121
9.2	Sensor Resources . . . . .	125
9.2.1	Force-Torque Sensor . . . . .	125
9.2.2	Digital Input Modules . . . . .	126
9.3	Actuator Resources . . . . .	127
9.3.1	Gripper Resources . . . . .	127
9.3.2	Screwer Resources . . . . .	129
9.3.3	Automatic Tool Changer . . . . .	132
9.3.4	Digital Output Modules . . . . .	133
<b>10</b>	<b>Evaluation of the Case Studies for Robot-Based Automation</b>	<b>135</b>
10.1	Structure of the Robot Cells . . . . .	136
10.1.1	A Flexible Industrial Robotic Cell: WiR Augsburg Innovation Laboratory . . . . .	136
10.1.2	Robarista Cell . . . . .	140
10.2	Implementation of RealCaPP Concepts with Real Hardware Components	141
10.2.1	KUKA KR Industrial Robots . . . . .	142
10.2.2	Gripper of the KR90: Zimmer Group GEH6180 . . . . .	143
10.2.3	Screwer of the KR90: Stoeger SPATZ 30 . . . . .	146
10.2.4	Force-Torque Sensor of the KR90: ME-Meßsysteme K6D80 . . . . .	149
10.2.5	Grippers of the KR10 and KR6 . . . . .	149
10.3	Hand Guiding of Industrial Robots . . . . .	151
10.4	Assembly of a Circuit Board Component . . . . .	154
10.5	Assembly of Aluminium Structures . . . . .	158
10.6	Robarista: A Robot Making Coffee . . . . .	165
<b>11</b>	<b>Evaluation of the Real-Time Performance</b>	<b>171</b>
11.1	Performance Evaluation of Adding Software and Hardware Components at Runtime . . . . .	172
11.2	Test Setup for the Real-Time Performance Measurements . . . . .	173
11.3	Latency Evaluation of the Real-Time Communication and Execution . . . . .	174
<b>12</b>	<b>Conclusion and Outlook</b>	<b>183</b>
12.1	Conclusion of the Thesis . . . . .	183
12.2	Outlook and Future Work . . . . .	185
	<b>Bibliography</b>	<b>189</b>
	<b>List of Figures</b>	<b>203</b>
	<b>List of Listings</b>	<b>207</b>

<b>List of Equations</b>	<b>207</b>
<b>List of Tables</b>	<b>208</b>
<b>Supervised Theses</b>	<b>209</b>
<b>Own Publications</b>	<b>211</b>

# Acronyms

- AAS** Asset Administration Shell.
- AI** Artificial Intelligence.
- AS** Application Services.
- ATC** Automatic Tool Changer.
- 
- CBS** Credit-Based Shaper.
- CFS** Completely Fair Scheduler.
- CPPS** Cyber-Physical Production System.
- CPS** Cyber-Physical System.
- 
- DDS** Data Distribution Service.
- DoF** Degrees of Freedom.
- 
- EDF** Earliest Deadline First.
- EKI** Ethernet KRL Interface.
- ETF** Earliest TX-time First.
- 
- FIFO** First-In First-Out.
- 
- ILP** Integer Linear Programming.
- IoT** Internet of Things.
- IPC** Industrial PC.
- 
- KRL** KUKA Robot Language.
- 
- LDS-ME** Local Discovery Server with Multicast Extension.
- 
- MQTT** Message Queuing Telemetry Transport.
- 
- OOP** Object-Oriented Programming.
- OPC UA** Open Platform Communication Unified Architecture.
- OSGi** Open Service Gateway Initiative.
- OTG** Online Trajectory Generator.
- OWL** Web Ontology Language.
- 
- P&P** Plug & Produce.
- PID** Proportional–Integral–Derivative.

**PLC** Programmable Logic Controller.

**PTP** Precision-Time-Protocol.

**RAMI 4.0** Reference Architecture Model Industry 4.0.

**RDF** Resource Description Framework.

**RDFS** RDF Schema.

**RealCaPP** Real-Time Capable Plug & Produce.

**Robarista** Robot-Barista.

**ROS** Robot Operating System.

**RR** Round Robin.

**RSI** Robot Sensor Interface.

**RTS** Real-Time Service.

**RTT** Round Trip Time.

**SOA** Service-Oriented Architecture.

**SPARQL** SPARQL Protocol And RDF Query Language.

**SRP** Stream Reservation Protocol.

**SWRL** Semantic Web Rule Language.

**TAS** Time-Aware Shaper.

**TCP** Tool Center Point.

**TDMA** Time Division Multiple Access.

**TSN** Time-Sensitive Networking.

**WCET** Worst-Case Execution Time.

**WiR Augsburg** Wissenstransfer Region Augsburg.

**Summary.** This chapter provides the motivation for why real-time capable Plug & Produce is essential for flexible robot-based production. It clarifies the open research questions and how they can be solved. Finally, the structure of the thesis is presented.

# 1

## Introduction

<b>1.1 Motivation</b>	<b>1</b>
<b>1.2 Research Question</b>	<b>3</b>
<b>1.3 Main Contributions</b>	<b>4</b>
<b>1.4 Structure of the Thesis</b>	<b>6</b>

This work aims to develop a concept for real-time capable Plug & Produce. The main focus is how the control for the execution of the process can be executed in real-time in distributed robotic systems.

In this chapter, initially, motivation is given for leveraging Plug & Produce methodologies to enhance production systems' flexibility. Furthermore, a concise overview is presented to define the concept of Plug & Produce, as outlined in Section 1.1. Then, in Section 1.3, the main contributions of this work are elaborated. Finally, the entire structure of this work is detailed in Section 1.4.

### 1.1 Motivation

Industry 4.0 has already triggered significant technological changes in the industrial landscape. This disruption includes the rollout of IoT devices (Internet of Things), Big Data analytics, and the use of Artificial Intelligence (AI) to increase production efficiency and produce better and more individualized products. One goal is the shift from mass production to Lot-Size-1 production. Mass production is an approach in which products are produced in large quantities based on a fixed design and a predefined production sequence. Mass production takes the advantage of economies of scale. The goal is to drive down the production cost per product by spreading the fixed costs for planning and integration of a production line over a high volume of units per product. Lot-Size-1, on the other hand, involves producing customized products in small batches where each product shall be unique. Also, the products can be tailored to the specific needs

of individual customers. With Lot-Size-1 production, many planning and design steps must be moved from design time to run time. While in mass production, both the plant design and the manufacturing functions can be planned and optimized in the design phase, in Lot-Size-1 production, each product may be different, and in some cases, the products may not be known at the stage of design time. In order to be able to implement a Lot-Size-1 production plant, such a production system must fulfill several points:

- A flexible system that can be programmed to handle various tasks, enabling efficient customizations of products.
- A modular and adaptable production line is essential to process different products and minimize downtime. For this, an adjustable or changeable tooling for the different products is needed.
- New plant components, tools, and processes shall be easily integrated at runtime without making significant adjustments to the hardware or software. Here, the goal is to minimize setup times for new products.
- Operators and programmers must quickly understand and configure the production system. Complex programming and setup procedures can lead to errors and inefficiencies.

Plug & Produce is one of the enablers to implement Lot-Size-1 production. Plug & Produce is a concept in manufacturing where production systems and components are designed to be easily integrated, configured, programmed, and reconfigured without extensive manual programming or complex adaptations [16]. The concepts of Plug & Produce have been partially adopted from the Plug & Play approaches widely used in computer systems. Today, it is taken for granted that when a keyboard is connected to a computer, it can be used without much configuration effort or additional software installation. The same behavior is desired for a production line where a new gripper or screwdriver for an industrial robot is "plugged in". Without much configuration effort and without the need to manually integrate the devices, the gripper should be used for handling a component or the screwdriver for screwing a screw into a specific product.

Production plants are usually a conglomerate of sensors, actuators, and control components. Manufacturing processes often involve several interconnected components that need to perform tasks together. For the components to interact successfully, some processes must run synchronously. Real-time execution and communication are therefore required. For example, if the task is to pick up components from a moving conveyor belt using an industrial robot with a gripper, the robot must synchronize with the conveyor belt and close the gripper as soon as it is synchronized. If there are delays in the execution or the communication between the robot, the gripper, and the conveyor belt, the component cannot be picked up successfully. Real-time in robotic systems refers to the capability of a system to respond to inputs and produce outputs within a predictable and bounded time frame (deadline), often in the order of milliseconds or microseconds. This ensures that processes can be executed almost simultaneously with minimal delay (minimal response time). [118, pp. 2,19]



In order to be able to implement such real-time critical processes without losing the flexibility for adaptations of the system to new products or processes, a real-time capable Plug & Produce concept is necessary.

## 1.2 Research Question

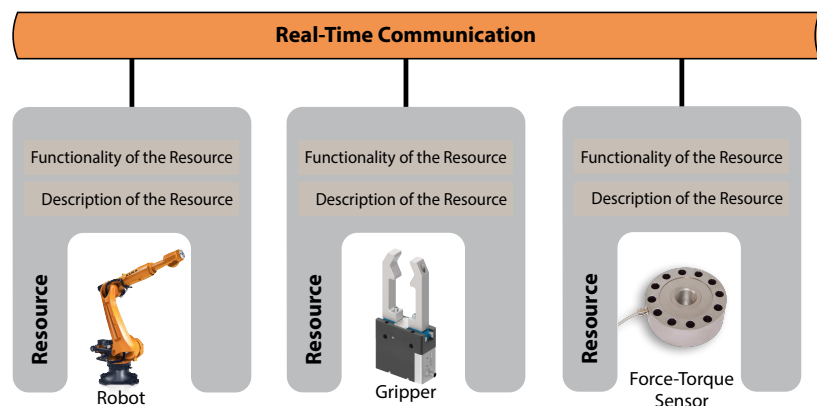
The present research work deals with how Plug & Produce concepts can be combined with real-time communication and real-time execution. Therefore, the underlying main research question is:

*How can flexible robot-based production systems be designed to add or replace production resources on the fly without complex integration and reprogramming, and how can these resources be combined into a distributed system that can perform tasks in real-time?*

Also, numerous other questions can be derived from this main research question:

- *How can resources be easily integrated into a system through self-description?*
- *What data must be included in the self-descriptions?*
- *How can data be shared among resources?*
- *Which data must be shared among resources?*
- *How can the individual resources' functionality be combined into the entire system?*
- *How can real-time critical processes be executed on distributed resources?*

In addition to the general research questions, these are presented with an example, shown in Figure 1.1. In the example, there are three new resources: an industrial robot, a gripper, and a force-torque sensor. The questions now are: *How can these resources*



**Figure 1.1.** Research questions with an example: How can the resources be integrated into the system? How can a force-controlled gripping process be realized using these resources?

*be integrated into the system? Moreover, how can a force-controlled gripping process be realized with these resources?*

Both the general research questions and the questions for the explicit example are considered in the context of the dissertation from different perspectives. Further on in the thesis, answers to all these questions will be found and explained.

### 1.3 Main Contributions

Several research results were achieved in processing and answering the research questions. The main contributions in this thesis are a **real-time capable Plug & Produce architecture**, which is the possibility to add, replace, and remove production resources at runtime. **Semantic Self-Descriptions** in the form of ontologies are used to describe resources with their skills and properties. The semantic descriptions are used to **automatically find plant configurations for required tasks**. Description and real-time critical process data can be exchanged between the resources via uniform communication interfaces. A **dynamic real-time capable communication middleware** is used to transmit process data. The software for realizing processes has a modular structure and is skill-based. A **modular Real-Time Service (RTS)** can be added to the system at runtime. These RTSs can be combined into complex processes that execute with hard real-time criteria. In addition, the processes can be **executed in a distributed manner**, which means that RTS can run on different devices that can exchange process data in real-time.

**Real-Time Capable Plug & Produce Architecture (RealCaPP)** A unified architecture has been created to add, change, or remove production resources easily. This includes uniform definitions of the resource, which are provided via an asset administration shell for the resources. The administration shell takes care of the description of the resource, how functionalities of the resource can be addressed, and provides a communication interface. There are defined procedures for how new resources can log on to the system and be integrated through a self-introduction. The architecture also offers a uniform communication interface for all components. Here, both non-real-time critical data and real-time critical process data can be transmitted on one medium via ethernet networks. In addition, there is the possibility in the architecture to load modular software components into the resources at runtime, which can be executed on the resources.

**Semantic Self-Description of Resources** Each resource has a detailed self-description. This self-description contains information about the type of resource, the properties of the respective resource, which functionalities the resource has, and how these functionalities can be addressed. The self-descriptions are defined in a uniform machine-readable format. Therefore, ontologies are used. A global registry collects the information of all resources involved and provides an accurate description of the system

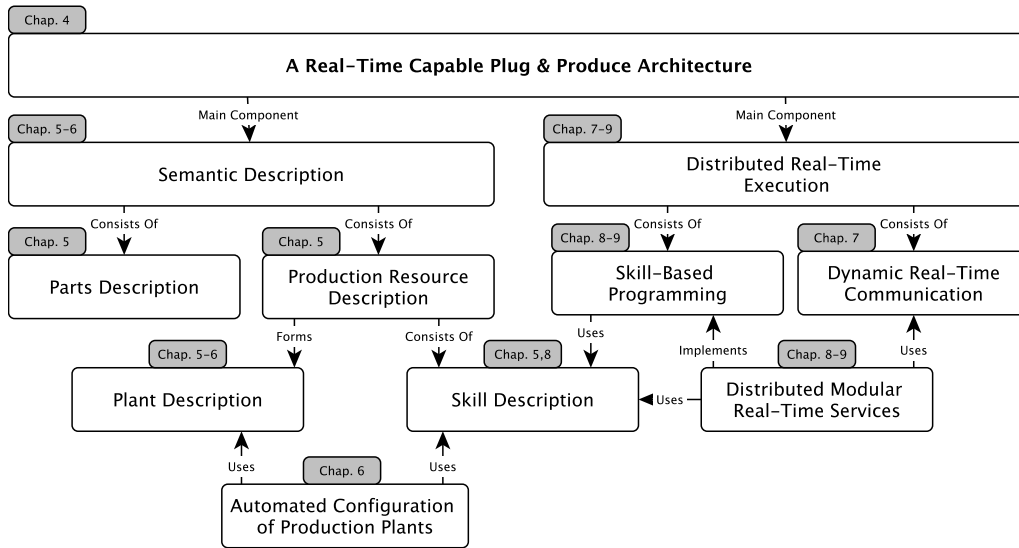
**Self-Configuration of Production Plants** The detailed description of the system and its components is used to find hardware configurations to perform a requested process step automatically. From the overall system description, it can be deduced which functionalities are required for a specific task. Resources that possess this functionality can be determined and checked to see whether these resources can be combined into a system. For example, a gripper must be attached to a robot in order to be able to move components.

**Distributed Modular Real-Time Services (RTSs)** The programming of the flexible production systems uses a skill-based approach. Each production resource has executable skills that can be combined to implement a production process. In order to be able to execute processes in the system, modular software components were developed that can be added to the system at runtime. For this purpose, a Service-Oriented Architecture (SOA) was established. The RTSs have defined interfaces and can be easily integrated and executed. The services can be executed either locally on one device or they can be executed distributed on several devices. In the local setup, shared memory is used for data exchange. In the distributed setup, a real-time capable communication middleware is used. With RTSs, it is easy to integrate the hardware and adapt the software to processes. In addition, the software components can be combined and reused as required due to their modular structure.

**Dynamic Real-Time Communication** Uniform communication is necessary so that resources can exchange information and data. To ensure that real-time critical processes can be implemented in the architecture, a real-time communication channel is set up between the resources to exchange process data. Real-time protocols are complex to configure. For the real-time communication channels, corresponding transmission time slots must be reserved for the real-time critical communication messages. The system configures automatically for the communication channels and reserves corresponding time slots for transmissions between resources. Since resources can be added or replaced at runtime, the configurations and reservations are done online without interrupting existing communications.

These individual results provide the basis for implementing a highly flexible plant structure. This means that products or production resources not known at the design time of the system can be integrated into a production system. This provides the basis for implementing Lot-Size-1 production.

Figure 1.2 shows the structure and the interrelations between the main components of the real-time capable Plug & Produce architecture. The main parts are the distributed real-time execution and the semantic description of objects, properties, and functionalities. The contributions mentioned above are based on these two parts. The semantic description includes descriptions of the parts to be produced and the description of the production resources with their respective characteristics and skills (*Skill Description*). The description of the entire plant (*Plant Description*) can be derived



**Figure 1.2.** Structure and interrelations between the main components of the real-time capable Plug & Produce architecture. The gray boxes indicate the chapters in which the individual components are described.

from the individual descriptions of the production resources. The description of the entire plant and the description of the skills can automatically be used to find plant configurations for given use cases. The distributed real-time execution consists of skill-based programming and the dynamic real-time communication. The skills are implemented in a service-oriented architecture through modular Real-Time Services that can interact with each other via the dynamic real-time communication. The relationship between the skills and the modular real-time services is mapped in the Skill Description.

## 1.4 Structure of the Thesis

The following chapters discuss how these individual contributions were achieved to create a real-time capable Plug & Produce architecture and how this answers the research questions.

Chapter 2 provides an insight into the fundamentals for developing real-time capable Plug & Produce. On the one hand, the overall integrations into the Industry 4.0 basic architecture are discussed, and the fundamentals of real-time and distributed systems are explained.

In Chapter 3, the problems associated with flexible robot cells and how Plug & Produce can be used for this purpose are explained using an example that runs through the thesis.

Subsequently, Chapter 4 introduces the real-time capable Plug & Produce architect. The basic architecture is presented, and the individual components of the architecture are discussed. Chapter 5 explains how the new components can be integrated into the system

by self-introducing the production resources. The required self-description for the self-introduction of the production resources and how these descriptions can be combined into a global knowledge base is also explained in this chapter. Chapter 6 describes how semantic descriptions can be used to represent the connection between resources and how, based on these descriptions, plant configurations for specific processes can be found automatically. How the resources can exchange process data in real-time is described in Chapter 7. The required dynamic configuration, e.g., the reservation of communication time slots of the real-time communication at runtime, is also specified here. The execution of software and how new software can be integrated into the system is explained in Chapter 8. Here, it is explained how modular software constructs can be developed based on a service-oriented architecture. It also describes how the software can be run locally and distributed to multiple resources. This concludes the description of the architecture.

In this Chapter 9, it is shown how resources with the associated RTSs can be implemented in the RealCaPP architecture. Abstract resource types are presented, from which many resources from the robotics area can be derived. For example, an abstract implementation for grippers is defined.

Chapter 10 shows how the RealCaPP architecture is applied in four different case studies. These range from classic industrial processes to making coffee with a robot. The feasibility study is supplemented by a quantitative evaluation of the implementation times in Chapter 11. This measured how long it takes to integrate new hardware components and how long it takes to set up real-time communication. In addition, the execution times for one of the case studies were measured to demonstrate the real-time capability of the architecture.

This thesis concludes with a summary of the results obtained (see Chapter 12). Furthermore, future research will be discussed.



**Summary.** This chapter describes the basis and underlying architectures in the field of Industry 4.0 that are important for Plug & Produce. It also lays the foundations for a real-time or distributed system so that it is clear what is meant by real-time and distributed when talking about real-time capable Plug & Produce for distributed robot-based automation.

# 2

## Fundamentals

<b>2.1 Cyber-Physical Production Systems (CPPSs)</b> . . . . .	<b>9</b>
<b>2.2 Reference Architecture Model Industry 4.0 (RAMI 4.0)</b> . . .	<b>10</b>
<b>2.3 Real-Time Systems</b> . . . . .	<b>12</b>
2.3.1 Real-Time Communication . . . . .	14
2.3.2 Real-Time Operating Systems . . . . .	15
2.3.3 Real-Time Applications . . . . .	17
<b>2.4 Distributed Systems</b> . . . . .	<b>18</b>

This chapter is intended to lay the basics for the further descriptions of this thesis. Plug & Produce is one of the enablers for implementing a flexible Industry 4.0-compliant production system. Plug & Produce uses many of the technologies defined in Industry 4.0 and combines them. One of the key components of Industry 4.0 are Cyber-Physical Production Systems (CPPSs), which combine the physical elements of manufacturing, such as sensors or robots, with digital technologies like IoT communication or self-descriptions to create interconnected and flexible production systems. These already researched areas can be used as a basis to implement real-time capable Plug & Produce.

Therefore, an overview of CPPSs is given in Section 2.1. Section 2.2 then discusses the reference architecture for Industry 4.0 and which areas are interesting for developing Plug & Produce methods. Section 2.3 describes what real-time systems are and what their characteristics are. It defines what real-time communication is and why special operating systems are needed for real-time execution and real-time communication. In addition, it describes what has to be considered when programming real-time applications. Finally, Section 2.4 describes what a distributed system is.

### 2.1 Cyber-Physical Production Systems (CPPSs)

A Cyber-Physical Production System (CPPS) [26, pp. 3] is a concept for integrating digital (the cyber part) and physical elements in modern production environments. Thus, it is a

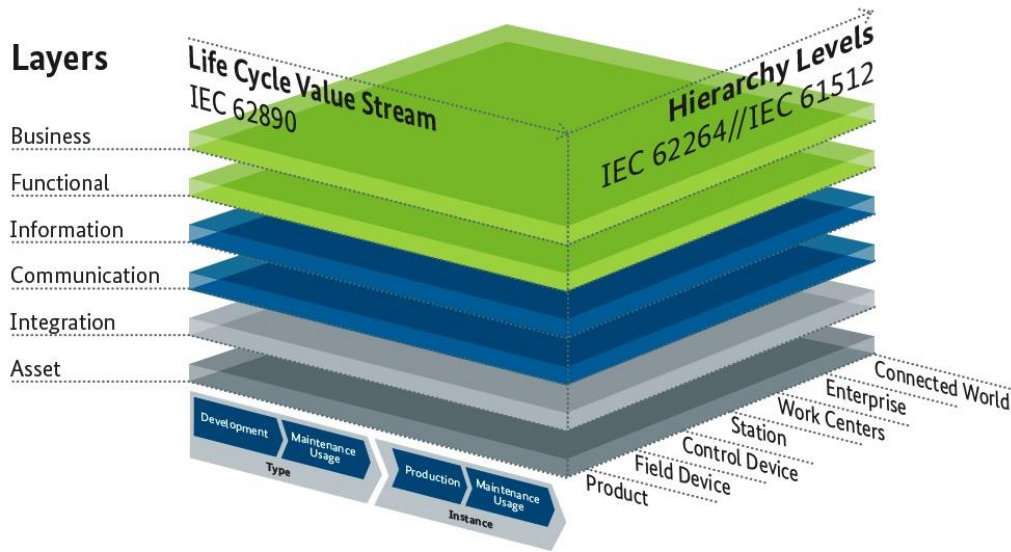
special form of a cyber-physical system in the industry context. Cyber-Physical Systems (CPSs) [199, pp. 3], as implied by its name, encompasses both cyber (e.g., software control) and physical components (e.g., a robot). These components are tightly integrated and work together to achieve a specific goal. Such systems are used in numerous domains, such as healthcare, transportation, energy, smart cities, and smart factories. The physical components of a CPS interact with the cyber part through sensors, actuators, or communication networks. The cyber part of the system gets information about the physical world through the sensors and communication channels, processes these data in real-time, makes decisions based on the processed data, and transfers control signals back to the physical part, creating a feedback loop. This seamless integration between the cyber world and the physical world enables CPSs to respond immediately to changes, making adjustments and to optimize the system. [128, 184] CPPS usually consist of embedded systems connected via wired or wireless networks and can thus exchange data and information with each other [76, pp. 2]. Internet of Things (IoT) technologies are mainly used for the data exchange and collaboration between the devices to form a coherent production system. For the exchange of data and especially for the control of the individual parts of the production plant, real-time communication and real-time control are necessary to make immediate changes to the production processes. Another essential aspect of CPPS is the recording of data and the subsequent analysis and processing of the data. The data collected from the physical processes and sensors are analyzed using analytics techniques and Artificial Intelligence (AI). This enables identifying patterns, predicting potential problems, and optimizing production processes. With CPPSs, the goal is to decentralize decision-making to a certain extent so that the individual production components can make autonomous decisions based on the data they collect and the algorithms they use. The benefits of cyber-physical production systems include greater production efficiency, reduced downtime, predictive maintenance, and the ability to adapt quickly to changing demands. They are seen as fundamental enabler for the smart factories and manufacturing processes of the future. [128]

Plug & Produce is closely related to CPPSs. A CPPS provides the technological foundation required for Plug & Produce. By implementing sensors, actuators, and real-time data analytics, a CPPS can detect and confirm the presence of newly introduced machines or components in the production. Then, CPPSs automatically adjust and integrate this new equipment into the production process to ensure a smooth and seamless incorporation. [147]

## 2.2 Reference Architecture Model Industry 4.0 (RAMI 4.0)

The Reference Architecture Model Industry 4.0 (RAMI 4.0) [189] is a standardized architectural framework that serves as a guideline for implementing Industry 4.0 concepts and technologies in manufacturing and industrial environments. RAMI 4.0 provides a structured and comprehensive model to design, plan, and implement Industry 4.0 solutions.





**Figure 2.1.** The Reference Architecture Model Industry 4.0 (RAMI 4.0) [189]

Figure 2.1 shows the three main dimensions that represent all important aspects of Industry 4.0. The shown structure can be used to evaluate and classify any Industry 4.0 technology or application. The first dimension is the layer axis that shows the information technology view of a physical asset. The layers represent the digital part of a physical object, for example, a machine. The physical objects are located on the lowest level (asset layer). The integration layer contains the interfaces to physical objects and defines which data are collected and which control commands are sent back to the physical objects. The integration layer is also used to provide computer-processable information about the asset, such as documents about the hardware, software, or function of an asset. The communication layer ensures uniform communication between the assets and defines a consistent data format to exchange information. The information layer contains all process-relevant information in the form of data. The data can be linked and preprocessed on this layer to obtain new, higher-value data. The functional layer contains all the abstract functions of assets. The functions are represented independently from physical implementations and show an abstract representation of the functionality of an asset. The business models and the resulting overall process are mapped in the business layer. In this layer, the abstract functions of the assets represented in the functional layer are orchestrated to an overall process. Legal and regulatory conditions also play a significant role here. [69]

The second axis is the life cycle and value stream axis. This axis covers the complete life cycle of a system, product, or process. The life cycle ranges from the initial idea of the asset in the planning phase, the design phase with the construction and implementation, to its use in a production plant. Of course, the life cycle also includes after-sales tasks such as service tasks or decommissioning. It is basically distinguished between the type and the instance of an asset. A type phase is again distinguished between the development,

Layers

Life Cycle and Value Stream

maintenance, and usage phases. The development represents the first idea of a product over the development and commissioning to testing and the generation of the first prototypes. The type maintenance and usage phase represents the development results, like drawings, manuals, or software models. When a product is produced accordingly, the type of the product becomes an instance. The instance phase is again divided into the production, maintenance, and usage phases. In the production phase, a unique asset is created out of the type and gets a unique identification, like a serial number. In the maintenance and usage phase, the final product or machine is represented as unique and usable. For example, an instance is delivered to a customer. For the customer, the products are initially again just types. They become an instance when they are installed in a specific system. The change from type to instance can be repeated several times. Of course, instances can also get destroyed in this phase, so maintenance is also essential to maintain production. [67, 69, 189]

### Hierarchy Levels

The third axis is the hierarchy levels. These hierarchy levels represent the different functionalities within a factory or plant. In addition, the product itself is represented, and the connected world is added. The hierarchy levels are very reminiscent of the automation pyramid. As in the automation pyramid, the plant is divided into different granularities, ranging from a field device like a sensor over an entire production line (work center) to entire enterprises. Compared to the automation pyramid, however, the goal is for all participants to network with each other across hierarchy levels. Therefore, it is possible for the production line to receive the information from a single sensor directly. The product (smart product) is also part of the network and can exchange information with the other components. In addition, the connected world offers the possibility of making data available in a factory and globally via the internet of things. [67, 69]

The standardization of information management, communication, and functionality in the individual layers of RAMI 4.0 offers a reasonable basis for implementing Plug & Produce. RAMI 4.0 provides the architectural framework and guidelines to effectively realize Plug & Produce concepts. Also, RAMI 4.0 aims to achieve seamless integration of processes and assets. This matches the goals of Plug & Produce of creating more flexible and efficient production systems.

## 2.3 Real-Time Systems

As stated in this thesis's title, the shown Plug & Produce concepts shall be real-time capable. For this purpose, it is first clarified what real-time is and why real-time is needed. A real-time system is a computer system that gets input data, processes these data, and responds to the input data within a specific time frame. A real-time system is a system where the correctness of the system behavior depends not only on the given results of the system but also on the response time of the computations. At a minimum, a real-time system consists of a controlled component, such as a robot, and a controlling component, like a real-time computer system. The real-time computer system gets physical signals from the controlled system by sensors and can interact with the environment by generating physical signals for the actuators. A real-time

computer system must respond to input from its controlled system within a given time interval. The point at which the system must deliver a result is a deadline. Deadlines can be categorized as soft if the output remains useful even after the deadline and firm if the usefulness decreases after the deadline. When missing a firm deadline can lead to significant consequences, it is a hard deadline. A computer system operating in real-time and bound by hard deadlines is referred to as a hard real-time computer system or a safety-critical real-time computer system. On the other hand, if the system does not have any strict deadlines to meet, it is categorized as a soft real-time computer system. [35, pp. 9][97, pp. 3]

An example of a hard real-time system is the airbag deployment system in modern vehicles. When a collision is detected, the airbag system needs to react and deploy the airbags within a very strict time constraint to ensure the safety of the vehicle's occupants. If the deadline is exceeded, this has a significant impact on the functionality of the system. However, numerous applications in robotic systems require hard real-time, such as sensor-guided movements. If a robot moves at a velocity of 0.5 m/s and the system has a delay of a tenth of a second (100 ms), the robot has already moved 5 cm in this time. If the task is to move the robot into contact with an object in a sensor-controlled manner, a reaction time of 100 ms is not sufficient. Values between 1 ms and 10 ms shall be reached.

The fundamental approach to designing a hard real-time system differs significantly from that of a soft real-time system. In a hard real-time computer system, there is an absolute requirement to maintain a guaranteed and precise timing behavior, even under all specified load and fault conditions. On the other hand, a soft real-time computer system can occasionally miss a deadline without severe consequences. Therefore, there are other characteristics that a hard real-time system must have: [97, pp. 8][57, pp. 9]

- Hard real-time systems must exhibit **deterministic behavior**, meaning that response times for critical actions are predictable and known, and there is a fixed upper limit on processing time that must be guaranteed to be met. The system behaves consistently, producing similar timing results under the same conditions and inputs.
- Hard real-time systems should be optimized to have a **minimal latency jitter**. For time-critical systems, a distinction is made between the delay of the calculations (latency) and the jitter of calculations. The latency refers to the time delay between the occurrence of an event and the system's response to this event. The jitter quantifies the inconsistency in the time intervals between two consecutive occurrences of an event. For a real-time system, the latency should be constant and the jitter very low. The jitter should always be a small fraction of the latency.
- Hard real-time systems must be highly **reliable** to ensure that critical tasks are always completed correctly and on time without failure or deviation.
- Hard real-time systems have to be highly **available**. The availability is quantified as the proportion of time during which the system is prepared to deliver the intended service.

- Real-time systems must **manage available resources** (Resource Management), such as CPU time, memory, and bandwidth, well to meet timing constraints.

It is also essential to distinguish between real-time and "fast" systems. A real-time system is defined by its ability to accomplish its task within the specified timeframe consistently. Even if another system can complete the same task much faster most of the time, it cannot be considered a real-time system if there is a chance of missing the deadline at any point. [49, p. 4]

### 2.3.1 Real-Time Communication

Many of the real-time systems are distributed across multiple computing units that need to exchange information in real-time. Therefore, real-time communication is an important topic in the sense of real-time systems. Real-time communication systems must meet the requirements for the timely exchange of data between two or more compute nodes: [97, pp. 167]

- One requirement is **timeliness**. The main distinction between real-time and non-real-time communication systems is the need for minimal message-transport latency and low jitter. In a distributed real-time system, the same times are required as in a locally executed real-time system, plus the transmission time between computation nodes. Information about the system must be read, for example, by sensors, the data must be processed, and the resulting control commands must be given back to the system. If reading, processing, and writing back are now done on several different processing nodes, then the data must be exchanged between each step, so the exchange times should be as minimal as possible. Real-time data needs to be accurate at the precise moment of use. In a distributed system, the only way to check temporal accuracy is by measuring the time duration between observing a real-time event by the sensor and reacting by the actuator. To achieve this, all involved nodes must have access to sufficient precision global time base. The responsibility of establishing and synchronizing this global time among the nodes lies with the communication system.
- Another requirement is the **dependability**. The system should be highly reliable, with minimal packet loss or data corruption, to prevent disruptions in communication. In non-real-time systems, this is achieved by time redundancy. If a message is lost, it is retransmitted at a later point in time. The problem here is that this increases the jitter significantly. This is especially not useful for cyclic messages, as waiting for the following message makes more sense than performing a time-consuming retransmission. Real-time communication systems often use error correction and detection techniques for robust channel encoding. For example, Forward Error Correction (FEC) is used, where redundant information (error correction codes) are added to the transmitted data. These codes enable the receiver to detect and correct errors without the need for retransmission. Another important point for dependability is determinism. Besides determinism's temporal guarantees, preserving order is also an important point. Maintaining the order of messages is crucial in many real-time applications. Deterministic

behavior ensures that messages are processed and delivered in the same order as they were generated, preventing confusion and data inconsistency.

- A further requirement is the **flexibility** of real-time communication systems. Real-time communication systems often encounter varying system configurations that change over time. An ideal real-time communication protocol should possess the flexibility to adapt to these changes without requiring software or hardware modifications. An adaptation in real-time would be ideal.

Now that the requirements for real-time communication have been defined, the different types of real-time communication will be introduced. A distinction is made between two types of messages: The event-triggered message and the time-triggered message. An event-triggered message is produced sporadically when an event occurs at the sender. The exact timing of events can be uncertain, leading to unpredictable delays in message transmission. This uncertainty can make it difficult to meet strict timing constraints in real-time systems [49, p. 43]. Furthermore, in event-triggered systems, a large number of messages can be triggered at one point in time. This can result in message overload, leading to congestion and potential data loss, impacting the system's dependability. These complexities can make it challenging to achieve robust and dependable real-time communication. In contrast, time-triggered approaches, where messages are sent at predefined time intervals, are often preferred in real-time systems, providing better determinism and timing guarantees. In time-triggered messaging, the sender and receiver agree on a precise time when messages are sent and received. Normally these agreed communication schedules are conflict-free, meaning that the time slot is intended only for a specific message. [97, pp. 167][49, pp. 42]

### 2.3.2 Real-Time Operating Systems

Now that it is clear what the requirements of a real-time system are and what possibilities there are to communicate in such a system, it is still necessary to mention where the execution of these real-time systems takes place. A real-time operating system (RTOS) [205, pp. 401][97, pp. 215][57, pp. 355] is an operating system specifically designed to meet the stringent timing requirements of real-time applications. The task of an RTOS is to respond to external events quickly. Therefore, an RTOS should have the following capabilities:

- An RTOS should have **minimal interrupt latencies**. Interrupt latency is the time span from the moment an interrupt occurs to the moment the CPU starts to process the interrupt handling. The system must support nested interrupts to ensure that the processing of low-priority interrupts does not delay the processing of high-priority interrupts.
- **Advanced task scheduling** algorithms are needed in RTOS. Tasks in an RTOS are assigned priorities, and the scheduler ensures that higher-priority tasks are executed before lower-priority ones. An important aspect of RTOS is also the preemption. Preemption allows tasks with higher priorities to preempt lower-priority tasks, ensuring critical tasks are not delayed by lower-priority activities. However, even with preemptive scheduling, tasks cannot be guaranteed to meet

their deadlines. The system must use a suitable scheduling algorithm that can achieve this goal.

- RTOS should provide mechanisms for **inter-task interaction**. Inter-task interactions in real-time systems refer to the techniques used for communication and coordination between different tasks running concurrently within the RTOS. In real-time applications, multiple tasks often need to work together to achieve a common goal, and effective communication between them is essential for proper system operation and synchronization.
- RTOS should have **short critical regions**. Critical regions refer to code sections in a task where a specific resource or data structure needs to be accessed atomically. Short critical regions are essential in RTOS applications to achieve correct and deterministic behavior.

Static Scheduling

There are multiple scheduling procedures to ensure real-time capability. Scheduling procedures for real-time systems are classified into static and dynamic scheduling. A scheduler is static when the scheduling decisions are made before the execution of a task or process. Tasks are assigned fixed priorities based on their criticality and timing requirements, and the scheduler follows a predefined scheduling policy. These processes are also deterministic. Often used static scheduling methods in real-time systems are priority-based First-In First-Out (FIFO) or priority-based Round Robin (RR). FIFO is one of the simplest scheduling algorithms and follows the principle of First-Come-First-Served. The tasks are executed in the order in which they arrive. The addition of priorities means that tasks are processed not only according to arrival but also according to priority. So, higher-priority tasks are executed before lower-priority ones. Once a task is started, it continues until it is completed or preempted by a higher-priority task. If a higher-priority task has been completed or enters a waiting state, the scheduler selects the following task with the next highest priority to run. The problem with this scheduling method is that processes cannot preempt other processes with the same priority. Prioritized RR combines the concept of priority-based scheduling with the time-sharing nature of Round Robin scheduling. The scheduler maintains separate queues for each priority level. At the beginning of scheduling, the highest-priority queue is chosen, and tasks within that queue are executed in a Round Robin manner, each receiving a fixed time slice of CPU time. If a higher-priority task becomes ready while a lower-priority task is executed, preemption occurs, and the higher-priority task takes over. Once a task is completed, the next task of the highest available priority is chosen. [24, pp. 403][97, pp. 248]

Prioritized FIFO Scheduling

Prioritized Round Robin Scheduling

Dynamic Scheduling

Dynamic scheduling involves scheduling decisions during runtime based on the current system state. Priorities and scheduling parameters can be adjusted dynamically during execution in response to changing task demands. Dynamic scheduling allows for better adaptability to varying workloads but is not deterministic. A dynamic scheduling technique is Earliest Deadline First (EDF) scheduling. EDF assigns priorities dynamically based on the deadlines of the tasks. The task with the earliest absolute deadline is given the highest priority, and preemption is used to ensure deadlines are met. [24, pp. 279][97, pp. 253]

Earliest Deadline First Scheduling

## Preempt RT Linux

A Linux system with Preempt-RT kernel [179] was used as the real-time operating system for the following experiments. Preempt-RT Linux is a specific variant of the Linux kernel that incorporates real-time preemption patches. These patches aim to enhance the real-time capabilities of the Linux operating system, making it more suitable for time-critical and deterministic applications. Other real-time approaches for Linux systems are usually based on cokernels. In cokernel approaches, an additional kernel is used to manage real-time threads besides the classic Linux kernel. The most common cokernel approaches are RTLinux [208], Xenomai [60] and RTAI [120]. Preempt-RT aims to maintain a single-kernel approach with low latencies and predictability. This makes the implementation of real-time applications similar to non-real-time applications. Only other scheduling mechanisms and priorities have to be set for real-time processes. Furthermore, there are additional protection mechanisms integrated into the kernel. For example, the system call `mlockall` disables memory swapping, which may lead to page faults, impacting the latency of a system. Another advantage of Preempt-RT is that an application can be run in real-time mode without being rewritten since no special system calls are used for real-time execution. In addition, in Preempt-RT, different schedulers are available for real-time execution. In addition to the classic Completely Fair Scheduler (CFS), there are the static scheduling methods FIFO (SCHED\_FIFO) and RR (SCHED\_RR) with priorities, and there are also approaches for dynamic schedulers with EDF (SCHED\_DEADLINE) [56]. [179]

### 2.3.3 Real-Time Applications

Various programming languages can be used to create real-time applications, but one of the most common programming languages is C or C++. The reason for this is the efficiency and performance of such applications. C and C++ allow direct memory manipulation and have direct access to hardware resources. In addition, C or C++ code is generated into highly optimized machine code, which is particularly well suited for time-critical applications. Furthermore, C and C++ provide deterministic behavior, meaning the developer has complete control over how the code executes. Real-time systems must avoid unpredictable delays caused by garbage collection or memory management overhead. C and C++ do not have automatic garbage collection, allowing developers to manage memory manually and avoid unpredictable memory allocation. Also, C and C++ have a predictable overhead, meaning the applications can be executed with high performance, even on systems with low computing power. [98]

However, these points alone do not make all programs written in C or C++ real-time capable. There are still a few restrictions to be aware of in order to write real-time applications with C or C++: [94, 185]

- Application shall have a deterministic and predictable behavior. Non-deterministic features, such as dynamic memory allocation, that lead to unpredictable execution times shall be avoided in the execution context. Therefore, all programs shall have a three-phase lifecycle with an initialization phase, a real-time execution

phase, and a termination phase. Non-deterministic executions may only be used in the initialization and termination phases.

- Memory allocations with `malloc` and `new` shall only be used in the initialization phase of an application because these operations are non-deterministic. Also, standard dynamic data structures like `std::vector` shall not be used, as they perform allocations in the background. Care should also be taken with strings. If the size of a string changes, memory can be reallocated. For applications with no predefined memory space, memory pools can be used with a fixed size of memory preallocated in the initialization phase and used in the real-time execution phase. Furthermore, the deallocation shall not take place during the real-time execution. This means that `delete` and `free` operations are only called in the termination phase of an application.
- Care should also be taken with inheritance in C++. The main problem here is with virtual functions. C++ creates the virtual method lookup table (vtable) per class and not per object. If a virtual function is called from an object, the vtable must be read from the inheriting class. This causes a back and forth jumping between object memory and the class's vtable, which is time consuming.
- When using Preempt-RT Linux the system call `mlockall` should be called at the initialization phase. The operator ensures that the memory used in the application is not swapped and unforeseen delays occur.
- Thread creations are also not inherently real-time safe. When a thread is created, the operating system must allocate resources for the thread's stack, thread control block, and other necessary data structures, which are not deterministic. Therefore, threads must be created in the initialization phase. After generating, the required scheduling procedures and process priorities shall be set. Before deleting memory in the termination phase, it is necessary to ensure that all threads accessing these memory areas have been terminated.
- In systems with multiple cores, the threads should be distributed manually to the cores. In real-time systems, the CPU cores are usually isolated from each other, and the threads are bound to a specific CPU. This eliminates the overhead of reordering threads to different cores. [95]

### 2.4 Distributed Systems

A distributed system is a type of computing system that consists of multiple interconnected independent computers that work together to achieve a common goal. In a distributed system, these computers communicate and coordinate their processes to provide a powerful functionality that a single, standalone computer could not offer. Three essential points make up a distributed system: Transparency, openness, scalability, and communication. [200, pp. 1]

#### Transparency

The main goal of a distributed system is hiding the physical distribution of its processes across multiple computers. When a distributed system seamlessly appears to users and applications like a single computer system, it is described as transparent. There are



different types of transparency. The location can be transparent. Users and applications can access computing resources without knowing where they are located or how they are distributed. Further, access to computing resources can be transparent. It should be consistent to access computing resources. Therefore, the user or application does not need to know if the computing resources are local or distributed. Additionally, there is the concurrency transparency. Concurrency transparency ensures that multiple users or applications can work with the computing resources without coordinating the access. In this case, the system hides that the resources may be shared by multiple competitive users or applications. [200, pp. 4]

Openness refers to the principle of designing and implementing systems with a focus on accessibility, interoperability, and the ability to extend or modify the system quickly. Open distributed systems rely on open standards for communication, data representation, and interfaces. These standards enable different components and systems to work together, reducing integration challenges. The open and standardized interfaces allow easy expansion or modification to incorporate new functionalities or technologies. [200, pp. 7]

Another important aspect of distributed systems is scalability. There are two directions of scalability: the vertical and horizontal scalability. In a distributed system, the individual computer nodes can be scaled and replaced by more powerful ones. This is referred to as vertical scaling. Vertical scalability is limited by the capabilities of a single computation node and may eventually reach a ceiling where further upgrades become impractical. Therefore, the system can also be scaled horizontally. Horizontal scalability involves adding more computation nodes to the distributed system to distribute the workload across multiple instances. However, achieving scalability in a distributed system often requires careful design, architecture, and consideration of data distribution, communication overhead, and synchronization. [200, pp. 9]

Interprocess communication is the core component of all distributed systems. Processes executed on different machines must be able to exchange information and process data to other processes. In distributed systems, this information and data is exchanged via a network between the computation nodes. Interprocess communication over a network is much more challenging than classical shared memory approaches, as they are used in non-distributed systems. [200, pp. 115]

Openness

Scalability

Communication



**Summary.** This chapter clarifies what Plug & Produce means for a flexible distributed robot cell. What is needed to integrate new resources, and what problems can arise? In addition, an overview is given of the case studies for which the Plug & Produce architecture is used.

# 3

## Plug & Produce in a Flexible Distributed Robot Cell

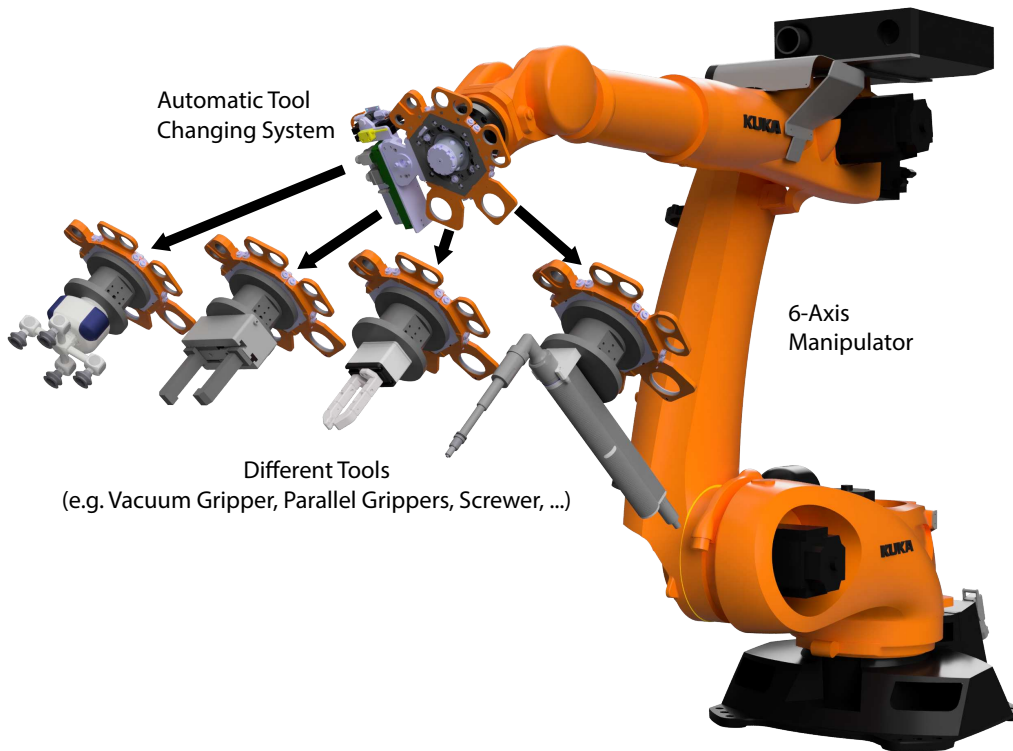
The trend in production is moving more and more towards individual production in which each product is unique. In order to respond to this high level of variability, the production facilities must also be designed to be highly flexible and customizable. Plug & Produce in the context of a flexible robot cell refers to a concept where robotic systems, machinery, and equipment can be easily interconnected and integrated without extensive manual programming or customization. This makes it possible to quickly adapt robot production to individual use cases and products without long planning and changeover times. In this chapter, an example is used to explain what a flexible robot cell is, what components make up such a system, and how the components must be able to interact with each other in order to perform a production task.

When talking about robot-based automation, the robot must be the focus. According to the definition of the Robotic Industries Association (RIA) a robot has the following definition:

*“A robot is a reprogrammable, multifunctional manipulator designed to move material, parts, tools or specialized devices through variable programmed motions for the performance of a variety of tasks” (RIA [40, p. 262])*

Based on this definition, a robot can perform flexible production tasks with different tools and special devices. What is still missing is that the robot can change tools independently and perform new tasks without reprogramming the whole system.

Figure 3.1 illustrates with an example the main components of a flexible robot cell. The basic component is the robot itself. In this example, a 6-axis manipulator also known as a 6-axis robot. These robots are widely used in various industries, including manufacturing, automotive, aerospace, and electronics, due to their ability to perform complex tasks. The combination of these six axes allows the manipulator to move in six Degrees of Freedom (DoF) to achieve almost all positions and orientations in three-dimensional space within the range of the robot. This flexibility makes these



**Figure 3.1.** Basic components of a flexible robot cell

manipulators well-suited for tasks requiring intricate movements, such as assembling small components, welding, painting, gluing, material handling, etc. Various tools can be attached to the robot. Depending on the tool, different tasks are possible. For example, a sensitive gripper can handle small components, while a glue gun is used for gluing. In order to give the robot or the robot cell the ability to change its tool independently and thus adapt to system requirements, an automatic tool changing system is necessary. An Automatic Tool Changer (ATC), is a mechanism designed to enable a robot to switch quickly and autonomously between different tools or end-effectors without manual intervention. Thus, the same robot can perform a variety of tasks by using different tools, allowing for more flexible manufacturing processes. As shown in Figure 3.1, using the robot with four different tools is now possible. In this case, it would be possible to switch between a vacuum gripper, two parallel grippers and a screwdriver.

This structure alone does not make a plant highly flexible. There are still some problems to be solved. Therefore, the problems are now discussed, and solutions to these problems are presented. Although switching between several tools is possible, the programmer must write a new robot program for each process. In addition, the control of the tool has to be integrated into the overall control of the robot cell so that the system or robot is able to interact with the tool. Even when the point is reached that all currently known tools have been integrated, the problem remains when a new tool is added for a new

---

task. Then again, the tool must be integrated, and the programming of the task must be completed.

**Solution:**

- Each component can communicate via a uniform interface.
- Each component can introduce itself (Self-Introduction) and provide the system with a self-description of the component's skills, interfaces, and properties.

Another problem can occur when one of the tools breaks. For example, if one of the parallel grippers breaks. The logical conclusion would be that the other parallel gripper is simply responsible for carrying out the activity the other gripper did first. However, this is not possible without further effort since the gripper control may be different, or the gripper dimensions may vary slightly, which in turn means that the programmer must adjust the programming or parameterization of the task. If the second parallel gripper fails, it would theoretically be possible to handle a component with the vacuum gripper. Here again, the complete control of the gripper would change, and in any case, the gripping position would change, which in turn has to be adjusted manually.

**Solution:**

- Similar skills of the components are combined and given a uniform control interface. For example, both the parallel gripper and the vacuum gripper have the skill to grasp.
- The property descriptions of the component can be used to customize the process. For instance, the dimension of the gripper can change the gripping position.

Now that the tools can be integrated and the skills, interfaces, and properties are known, there is still the problem of programming the different changing tasks. If yesterday's task was to move a product X with a parallel gripper from position A to position B and the new task is to move a product Y with a vacuum gripper from position C to position B, then the entire task must be reprogrammed.

**Solution:**

- Development of reusable parametrizable software modules that perform recurring tasks or skills, which are built upon the skills of the components. For example, a software module for the pick and place skill has positions as parameters and can be executed with different components that can grasp.

In most cases, several components are needed to implement skills. For example, if the task is to screw a screw into a component, a robot is needed to carry out the movement and a screwdriver to do the screwing. This creates a distributed system consisting of

a robot and a gripper. In such distributed systems, new problems arise. One issue is the timing dependencies between distributed components. For the given task there is an exact dependency between the pitch of the screw to be screwed in, the screw hole depth of the component to be bolted, the rotation velocity of the screwdriver, and the movement velocity of robot that moves the screwdriver. If the robot moves too fast, the robot pushes the screwdriver with too high force onto the screw and can damage the screw or the screwdriver. The same happens when the robot starts to move while the screwdriver is not yet in rotation. In the reverse case, when the screwdriver starts screwing in before the robot moves, the screwdriver can slip off the screw. It is, therefore, necessary that the components can exchange their data and control commands without delays. Especially if additional sensors, such as force or distance sensors, are added to the robotic system and the robot movement is to be influenced depending on the sensor values, the communication between robot and sensor must be carried out without long delays.

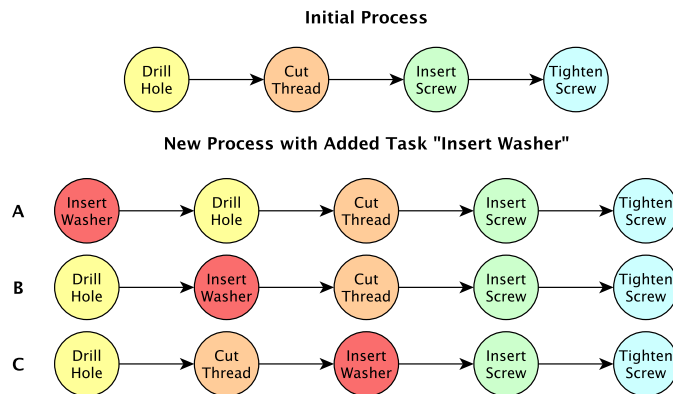
**Solution:**

- The distributed components should be connected via a unified communication channel to exchange data and control commands in real-time between the components.
- The communication channel must be dynamic and deal with changing components.
- The modular software components shall be distributable in the system as required.

A problem that remains is that the programmer must know all the components in a system and define a configuration that can perform a given task. For example, the task is to transport product X from position A to position B. Now, the programmer decides which gripper is suitable for moving the component and selects a robot that can carry the payload of the product plus that of the gripper. The programmer must also check whether the robot is suitable for attaching the gripper to the robot with all the mechanical and control interfaces and whether the robot can reach positions A and B. He can make the customized application if all boundary conditions are met.

**Solution:**

- The precise self-description of the components with their interfaces, skills, and properties allows us to find possible configurations suitable for executing a corresponding task.
- Either possible system configurations can be suggested to the programmer, or the system itself can select a possible configuration, configure itself accordingly, and then execute the task.



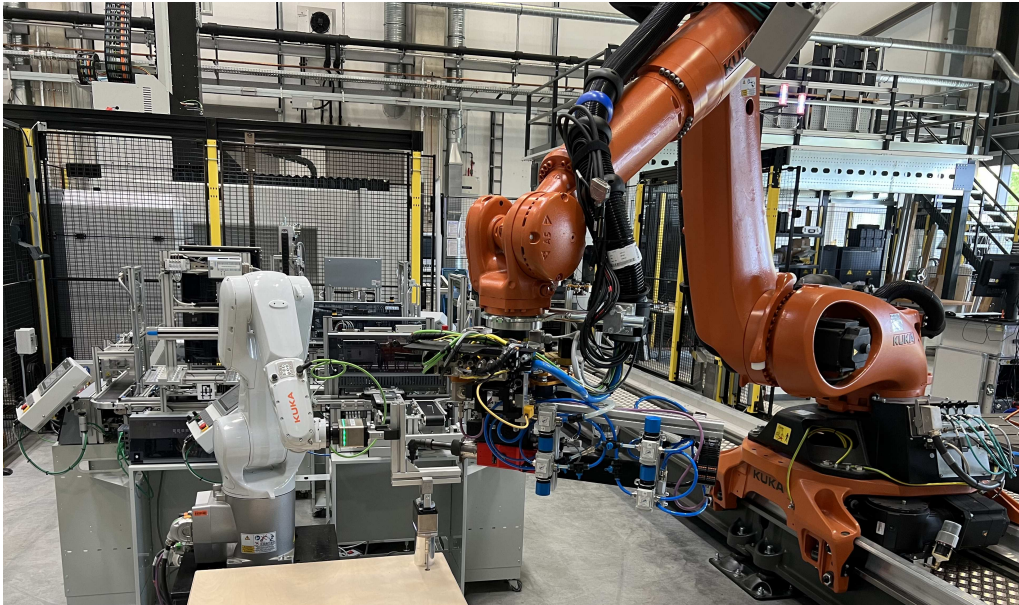
**Figure 3.2.** Possible execution sequence of the exemplary task sequence. The circle represent tasks (Task), while the arrows represent the execution order →

One last problem that remains is that if several tasks are to be executed that have dependencies on each other and cannot be executed simultaneously, the order of the tasks to be processed must be determined. For example, there is a four-step process: drill a hole, cut the thread, insert a screw, and tighten the screw. There is a clear sequence in which the individual steps must be carried out. Now the additional task of placing a washer on the screw is added. This task only has the dependency to be performed before the screw is inserted. Thus, there are already three possible sequences of execution. Figure 3.2 shows the possible execution order for the initial process and the possible execution order after adding the task Insert Washer. The problem here is that determining the order of execution is becoming increasingly complex. In addition, there are also dependencies on tools. The two inserts can be executed with a gripper. The drilling requires a drill, thread cutting requires a thread cutter, and the tightening of the screw requires a screwdriver. This influences the optimal sequence in which the number of tool changes should be minimal.

**Solution:**

- Automatically orchestration of the tasks, taking into account the dependencies of the execution order and the associated tools.
- Adding preconditions and postconditions for the tasks allows easier automatic determination of the execution order

If all these problems can be solved, we can talk of a highly flexible Plug & Produce capable robot cell. In the following chapters, the individual solutions presented briefly will be discussed in detail. The resulting solution is then examined in two different scenarios. On the one hand, an industrial case study is carried out on how the techniques can be used to handle and produce components in a highly flexible Plug & Produce



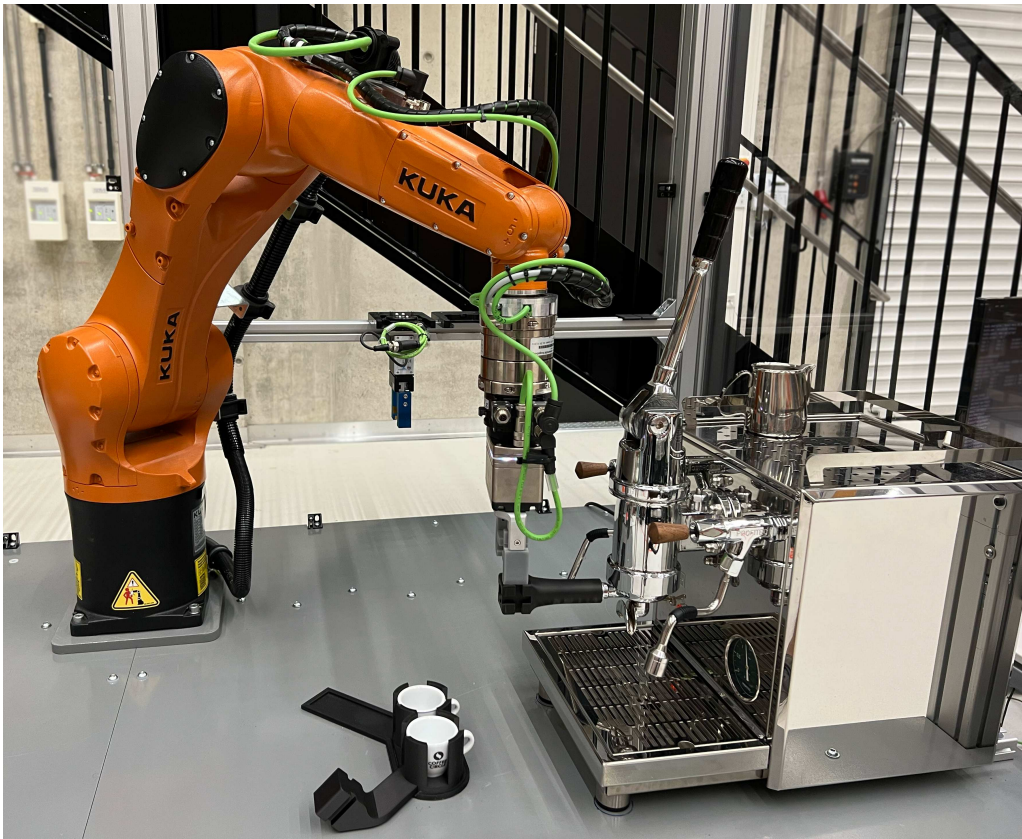
**Figure 3.3.** Industrial high flexible robot cell

capable robot cell. On the other hand, it is shown how these techniques can be applied to simple processes that everyone knows, as making coffee.

Figure 3.3 shows the industrial scenario with two robots equipped with ATCs and additional production facilities that can work together to perform a handling or production task. Figure 3.4 presents a scenario where the robot can prepare a coffee with a portafilter coffee maker. Coffee preparation also requires different tools for the individual process steps. These two completely different case studies will be used to explore how modular software constructs can be used to implement the capabilities of components and processes. It is checked how it looks with the reusability of these software modules in totally different use cases. Whether the chosen solutions are suitable for the problems shown will be clarified in Chapter 10.

The following chapters will use parts of the presented example with the basic components of a flexible robot cell shown in Figure 3.1 to explain the concepts of real-time capable Plug & Produce for robot-based automation.





**Figure 3.4.** A robot cell for making coffee



**Summary.** This chapter describes the basic concept for building a Real-Time Capable Plug & Produce environment. For this purpose, the objectives for such a concept are defined, and a uniform Real-Time Capable Plug & Produce architecture is described. Also, the main components of the architecture are explained, which will be discussed in more detail in the following chapters.

# 4

## Concept of a Real-Time Capable Plug & Produce Environment

<b>4.1</b>	<b>Fundamentals</b>	<b>30</b>
4.1.1	Skills, Capabilities and Services	30
4.1.2	Asset Administration Shell	30
4.1.3	Open Platform Communication Unified Architecture (OPC UA)	32
4.1.4	Time-Sensitive Networking (TSN)	34
4.1.5	OPC UA over TSN	35
<b>4.2</b>	<b>Real-Time Capable Plug &amp; Produce</b>	<b>35</b>
<b>4.3</b>	<b>RealCaPP Architecture (Real-Time Capable Plug &amp; Produce Architecture)</b>	<b>36</b>
4.3.1	A Uniform Communication Interface	36
4.3.2	Asset Administration Shells for Robot Components	37
4.3.3	Modular Real-Time Capable Software Components	39
4.3.4	Global Knowledge of Resources, Products and Services	39
<b>4.4</b>	<b>Related Work</b>	<b>40</b>

To make automation systems as flexible and robust as possible, it must be feasible to adapt the system at runtime, both the physical and cyber parts of the system. Such a system should have the option to add components at runtime, replace them with others, or remove them. Also, the system needs the capability to change the processes to be executed or to add new ones without significant adjustments by programmers or system integrators. For example, if a robotic system can assemble circuit board components with one gripper, the system must be able to easily reconfigure to assemble aluminum structures with gripping and screwing processes without completely reprogramming the robot cell. Therefore, the system must understand the individual components, what characteristics and skills the components have, and whether these skills match the task the system is being asked to perform. After the system is aware of whether a task can be executed, the requirement is to execute the tasks distributed even under hard real-time requirements on the created system. In this chapter, the base architecture is presented

that enables flexible and robust automation system design through real-time capable Plug & Produce.

The Real-Time Capable Plug & Produce (RealCaPP) approach presented in this thesis shows how a system must be designed with the help of descriptions of components, processes, and products, a uniform real-time capable communication interface, and a real-time capable execution concept to cope with the mentioned requirements of flexibility and robustness. First, the fundamentals necessary for such an architecture are discussed, and related work is considered (cf. Section 4.1). Section 4.2 explains what Plug & Produce is and elaborates the differences between a real-time capable Plug & Produce system. How a basic architecture for real-time capable Plug & Produce looks like and which components it consists of is explained in detail in Section 4.3.

## 4.1 Fundamentals

Before discussing the concept of a Real-Time Capable Plug & Produce environment, a recapitulation of some of the techniques used to integrate and communicate with hardware resources is described.

### 4.1.1 Skills, Capabilities and Services

In the following chapters, there will be repeated references to skills, capabilities, and services, so these terms will be briefly explained, and the respective commonalities and differences will be elaborated:

#### Capability

A capability refers to an overall potential or capacity of functionalities that a component or system possesses. Capabilities include a combination of different factors such as characteristics, hardware components, or specific technologies that enable a special functionality. Due to the driven axes of a robot, a robot basically has the capability to move.

#### Skill

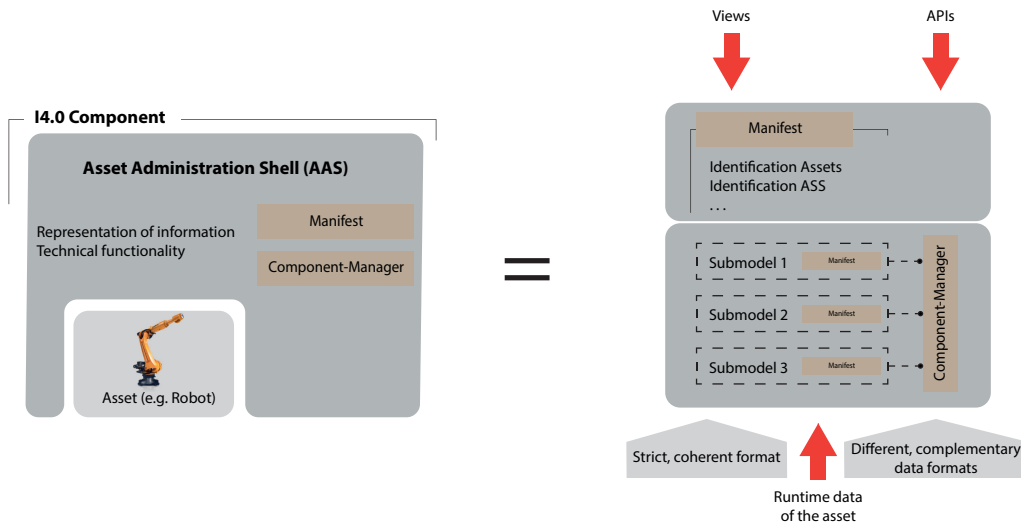
In the automation context, a skill is a specific ability that a system or component can acquire through programming. These skills enable the system or component to effectively perform certain tasks or actions. For instance, a robot might possess the skill to grasp objects with a certain type of gripper. The distinction between capability and skill is often very difficult, as there is a smooth transition here.

#### Service

A service refers to a specific task, function, or activity that a component or system performs to achieve a particular goal. Services use the skills and functions of a system or component to perform tasks. A service, for example, would be the executable skill of a robot to move an object. [96, 186]

### 4.1.2 Asset Administration Shell

In order to enable multiple components from different manufacturers to work together and exchange information, a uniform description of the individual components must be available. For this purpose, so-called *Asset Administration Shells* are used. An Asset Administration Shell (AAS) [20] is a digital representation of objects with their characteristics and behaviors. Figure 4.1 shows the structure of an AAS.



**Figure 4.1.** Structure of the Asset Administration Shell (Adapted from [22])

In this context, an asset is an object and can be either abstract, for example, a part of software, or physical, for example, a robot or a tool of a robot. An AAS consists of an externally accessible table of content for the self-description called manifest that links to the functional and non-functional information of the asset, a component-manager that manages the exchange with the actual asset, and multiple submodels, which represent different aspects of the asset. The submodel can be descriptive, for example, what is the maximum payload or the maximum velocity of a robot, or it can contain process capabilities, such as moving a robot from a start configuration to a target configuration. These submodels make it simple to add descriptions or behaviors for appropriate use cases. For example, if information about the energy efficiency of a robot is required, a submodel can be added for this use case. In addition, the AAS offers several views that provide the data from the asset and the AAS to other components. Via views, it is possible to provide different data to different user groups [160]. For example, a developer of an asset is shown more information than an operator of an asset. Also, there are APIs for accessing the AAS and the asset. [22]

The combination of an asset with its AAS is an Industry 4.0 Component (I4.0 Component). An I4.0 component must be able to communicate with other components with either a passive communication capability (e.g., communication with RFID tags), active communication capability (e.g., fieldbus communication), or an Industry 4.0 compliant communication capability (e.g., OPC UA communication). [43] OPC UA (see Section 4.1.3) serves as one of the exchange formats for I4.0 components, but also data formats like XML, JSON, Resource Description Framework (RDF) or AutomationML can be used. [20]

It is also possible to nest multiple I4.0 components into a more comprehensive I4.0 component. With nesting, it is possible to describe even more complex composite I4.0

Submodel

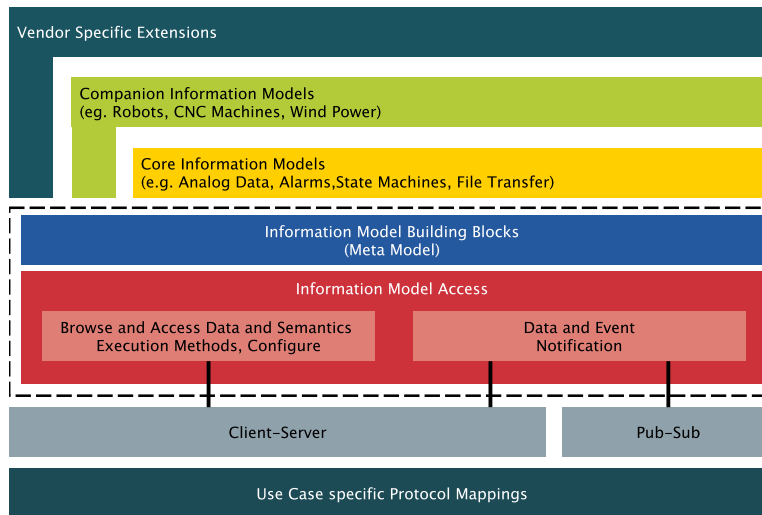
I4.0 Component

components by Composite Administration Shells, for example, a robot with its tools and sensors can be combined into a robotic system, which has higher-level descriptions or capabilities. For example, a robotic system consisting of a robot and a gripper can perform the skill to move an object from a starting point to a target point. [21]

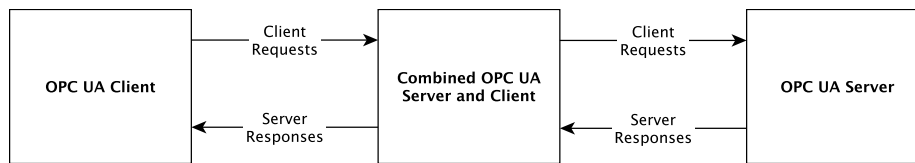
### 4.1.3 Open Platform Communication Unified Architecture (OPC UA)

An essential part of the developed architecture is the Open Platform Communication Unified Architecture (OPC UA) standard [81] that is considered as a key technology of Industrie 4.0 for connected manufacturing [73]. OPC UA is a platform and vendor-independent interoperability standard for data exchange in industrial automation systems. The OPC UA standard consists of several specifications developed in collaboration with various representatives from industry and research. OPC UA goes beyond a pure data exchange standard. It is much more a standard that contains a comprehensive information model and services to access and exchange data between multiple information models that can be available locally or distributed on different devices. The information model contains the actual data, the associated metadata, and links between the data in an object-oriented manner.

Figure 4.2 shows the system architecture of OPC UA. There are two basic concepts for exchanging information via OPC UA. There is client-server communication and publish-subscribe (Pub-Sub) communication. Several protocols can be used for the actual network transmission, which can be selected depending on the use case. The communication interfaces can access the data of the information model via an abstraction layer (Information Model Access). In order to simplify standardization, there are predefined information model blocks that can be used (Information Model Building



**Figure 4.2.** OPC UA system architecture for the combination of communication interfaces and the information model (Adapted from [143])



**Figure 4.3.** OPC UA Client-Server interaction (Adapted from [135])

Blocks (Meta Model)). Based on this metamodel, the information can be stored in the information model (Core Information Models). To avoid that every manufacturer or developer arranges his information model in different structures, there are companion specifications that standardize the information model for a domain (Companion Information Models). For example, a companion specification for robotics defines which information a robot needs and in which structure the data is stored. Based on these basic information models, it is possible to add additional vendor-specific information. [143]

The OPC UA Information Model utilizes the concept of Object-Oriented Programming (OOP) for the representation of information. OPC UA uses OPC UA Nodes as fundamental representations for information and behaviors in the information model. Nodes can represent information like variables, data types, methods, or events. Via relations, nodes can be interconnected with each other. In addition, classic OOP functionalities such as inheritance, polymorphism, abstraction, and encapsulation can be used. Each data element that can be accessed in the information model is defined as an OPC UA Object which inherits from OPC UA Node. An OPC UA Object can contain variables, methods, and events with the corresponding metadata, like data types or method parameters. This allows all information about an automation system to be stored in linked OPC UA Objects containing information and providing executable methods. [141]

OPC UA  
Information Model

Via the communication interfaces, it is possible to access objects of the information model or to synchronize the information over several distributed information models. An OPC UA system that communicates via client-server communication is composed of OPC UA Servers and OPC UA Clients that can interact with each other. Each system can contain multiple clients and multiple servers. Each server can interact with one or more clients. The same applies to the interaction between clients and servers. Figure 4.3 shows the client-server interaction in an OPC UA system. If communication occurs across multiple servers, a component must provide both a server and a client. For communication, the client sends requests in form of service requests to the server. The server decodes the request and locates the service to be executed. These services range from simple read and write commands for data over method calls to services for reconfiguring the server [139]. These services can access OPC UA objects of the information model and interact with them. If a service is found and executed successfully, the server sends a corresponding response to the client. Due to the requests, client-server communication is not suitable for constantly changing data and many-to-many data exchange. In addition, each request and service decoding takes time, resulting in higher latencies, which is especially problematic for real-time critical transmissions [37].

OPC UA  
Client-Server

Client-Server  
Subscriptions

The OPC UA Client-Server communication also has a link to data and event notifications (cf. Figure 4.2). It is possible to make so-called *Subscriptions* on data elements and events of the server. This Subscription is generated by the client and is stored in the server. For the observed elements, either on change or at regular intervals, messages are sent to the client. All transmissions are connection based, so it is more suitable for a one-to-one relationship between client and server. [135, 139]

OPC UA Pub-Sub

OPC UA Publish-Subscribe (Pub-Sub) allows many-to-many communication and has the advantage of not being request-based. Therefore, it is particularly suitable for constantly changing data, such as sensor values. OPC UA Pub-Sub uses a message-oriented middleware for the communication. There are two types of communication participants: OPC UA Publishers and OPC UA Subscribers. Both publisher and subscriber are OPC UA Servers and have their own information model. OPC UA Publishers access the information model, wrap the specified OPC UA objects into a message and transmit these messages to the message-oriented middleware, not knowing who receives the messages. OPC UA Subscribers can express interest in the information, connect to the message-oriented middleware and access the required data, unpack the OPC UA Objects and write them into the information model. Therefore, the subscriber does not need to know who the publishers are. Pub-Sub communication thus has the advantage that the system remains scalable, as new publishers or subscribers can be added easily, and the cycle times for transmissions can be scaled to very high frequencies because asynchronous communication is used. Depending on the application, different message-based middlewares can be used, ranging from broker-based communications such as MQTT (Message Queueing Telemetry Transport) [70, p. 9] to broker-less communications that use the transport medium as middleware, such as UDP multicast. [136]

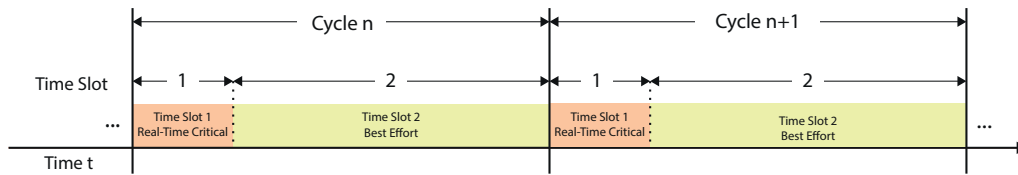
#### 4.1.4 Time-Sensitive Networking (TSN)

Time-Sensitive Networking (TSN) refers to a series of standards that add real-time capabilities to standard ethernet. The goal of TSN is to provide a deterministic communication standard that enables both real-time critical and non-real-time critical communication over ethernet. This means that packet transport with limited latency, low variation in packet delay, and very low packet loss shall be guaranteed. [78] Since even with priority approaches like Class of Service (CoS) [11], latencies could not be guaranteed, as already sent messages with low priority can not be aborted [113].

Credit-Based  
Shaper

In the TSN standards a distinction between three traffic classes are made: Best-Effort traffic, Credit-Based traffic and Time-Aware traffic. Best-Effort is the lowest-priority traffic class. It has no real-time guarantees and is used for traditional data communication, like sending an email. Credit-Based is a mid-priority traffic class in TSN. It can be used for soft real-time transmissions. Credit-Based Traffic uses a Credit-Based Shaper (CBS), published as Standard IEEE 802.1Qav [5]. Each data stream with the Credit-Based priority can reserve bandwidth for the transmission. The CBS then allocates the total bandwidth fairly among competing Credit-Based streams. Therefore, streams with a higher credit rate can send their data earlier. An example of Credit-Based traffic would be the camera data for monitoring a process.





**Figure 4.4.** TSN time division multiplexing with reserved time slots to enable the transmission of periodic real-time data. In this example, one time slot is reserved for real-time critical data transmission, and the rest of the cycle can be used for best-effort data traffic. (Adapted from [77])

The TSN standards also guarantee a bounded end-to-end latency for real-time sensitive data traffic. The most important component of the TSN standards in terms of bounded end-to-end latencies is the Time-Aware Shaper (TAS), published as Standard IEEE 802.1Qbv [6]. TSN uses Time Division Multiple Access (TDMA) [127, p. 33] as a fundamental concept. It divides time into cycles, discrete segments of equal length. Each cycle is divided into time slots. These time slots can be used to transmit data packets with real-time requirements. An example can be seen in Figure 4.4. The TAS temporarily interrupts conventional Best-Effort ethernet traffic so that real-time critical data traffic can be transmitted within a reserved time slot. [6]

Time-Aware  
Shaper

In the further course of the work, the focus is on the hard real-time transfer with the TAS. In Section 7.1.1, further fundamentals of TSN are shown.

#### 4.1.5 OPC UA over TSN

The two technologies OPC UA and TSN can be combined to OPC UA over TSN, which enables data transfer in industrial production in real-time using a uniform, vendor-independent and open standard. OPC UA over TSN uses the TSN-network as message-based middleware for the transmission over OPC UA Pub-Sub. This means that only the data-link-layer of the OSI model (Open Systems Interconnection model) [194] adds the TSN mechanisms. In this process, the raw data is packed directly into ethernet frames without the semantic enrichments of the information model and sent via TSN as real-time critical data. For the transmission of the data the highest priority is used and the data is sent as Time-Aware traffic. By combining the two techniques it is possible to get all the advantages of both techniques. TSN ensures a deterministic communication, while OPC UA provides a standardized information model for the interoperability of production components. Even large distributed production plants with real-time communication can thus be implemented in a scalable manner. [31, 32]

## 4.2 Real-Time Capable Plug & Produce

First, it must be clarified what is meant by the term Plug & Produce. *Plug & Produce*, also known as *Plug & Work*, defines the automatic integration, modification or removal of production resources in industrial plants [17]. Approaches for Plug & Produce require an

Plug & Produce

environment where smart devices can be easily plugged in and communicate with other devices, exchange products, device descriptions, or process models in order to begin producing a particular product. While most Plug & Produce approaches focus on how devices can be found or removed, these approaches neglect the fact that hard real-time execution is required for later execution in many application domains. Since industrial plants are mainly distributed systems, the communication between the individual devices must also be capable in real-time to guarantee execution in real-time.

In this thesis, an approach is developed how robots and industrial components can autonomously assemble into a smart robot-based manufacturing system. Such a manufacturing system consists of a set of industrial components and robots, each offering different manufacturing services (skills). These services can be used individually or in combination to perform manufacturing tasks in the system. For this purpose, the following objectives must be met to realize a real-time capable Plug & Produce architecture:

- Robots and industrial components can be added and removed at system runtime.
- The robots and industrial components are controlled and monitored via services.
- Each service shall be executable in real-time.
- Combinations of services can be executed locally on one device or distributed on multiple devices in the system without losing real-time guarantees.
- All production processes are implemented as a combination of services and can be added, changed, or removed during the system's runtime.
- Information on processes, products, and resources are merged into a unified data source and can be used for execution.

In the following, the concepts that meet the required goals are shown.

### **4.3 RealCaPP Architecture (Real-Time Capable Plug & Produce Architecture)**

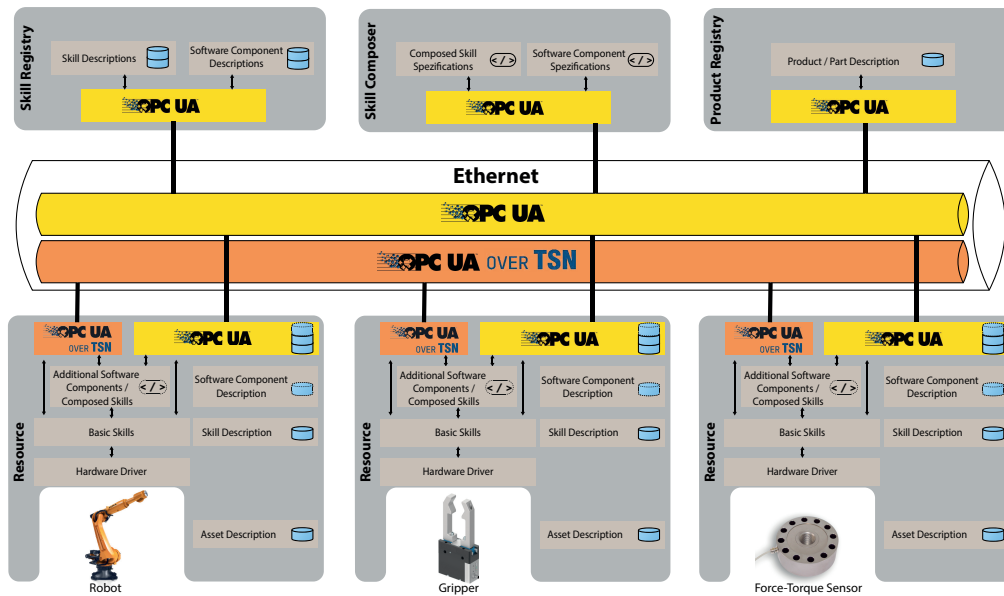
The approach presented in this thesis aims to design and implement a Plug & Produce environment with intelligent cyber-physical systems for an adaptive industrial production plant. Figure 4.5 shows the basic system architecture for real-time capable Plug & Produce. The Plug & Produce environment consists of a uniform communication interface to communicate and exchange data between components, several resources, e.g., a robot or a gripper, that combine into a robotic system, and management components that collect and link data, analyze the system, or handle the addition or removal of resources and skills.

#### **4.3.1 A Uniform Communication Interface**

Non-Real-Time  
Communication

A uniform communication interface is essential for an architecture in which components can act in a distributed manner. A middleware via ethernet is used as a connecting link between all architecture components. A basic distinction is made between two types of communication: Non-real-time capable communication is used for the configuration of the system and the exchange of information and non-real-time critical data. For this

### 4.3 RealCaPP Architecture (Real-Time Capable Plug & Produce Architecture)



**Figure 4.5.** System architecture for a real-time capable Plug & Produce environment using OPC UA and OPC UA over TSN as middleware. (See Eymüller et al. [54])

type of communication, OPC UA client-server communication is used, which does not support transmissions in real-time. This type of communication is represented in yellow in Figure 4.5. The advantage of this communication layer is that a connection between components can be established without prior configuration. This also facilitates the addition of new components to the system.

The second type of communication contains real-time critical data. Particularly, for control or feedback tasks of robotic industrial plants, hard real-time is required. Sensor-controlled movements are one example of hard real-time critical data exchange, where the sensor value must be read, communicated, processed, and transmitted to the robot in one cycle. Otherwise, jerky paths can occur, which can damage the robot's gears. Hard real-time communication structures require a complex configuration, e.g., the reservation of communication times or message priorities. For the configuration of the real-time communication channels, the non-real-time capable communication via OPC UA is used. OPC UA over TSN is used for hard real-time communication. The OPC UA over TSN communication paths are represented in orange in Figure 4.5. After the configuration, the OPC UA publish-subscribe mechanism is applied, and data are transmitted via TSN scheduled traffic via reserved time slots to avoid any delay from the network.

Real-Time  
Communication

#### 4.3.2 Asset Administration Shells for Robot Components

The system consists of different resources. A resource is a hardware component in an automation system, for example, a robot, a sensor, or a robot tool. Each resource

has an asset administration shell describing the device and its capabilities. This AAS consists of three essential parts: the control services, a description of the resource and its capabilities, and a uniform communication interface. The resources and the AASs are shown at the bottom of Figure 4.5.

Basic Skill

The first part of the AAS is the *control services*. The control services provide the direct interface to the actual hardware. There are low-level hardware drivers for controlling and interacting with the hardware resources. Since the user of such a system does not want to deal with these low-level drivers later on, so-called Basic Skills are used. A *Basic Skill* is an executable service that hides the hardware control from the user. For example, a robot has the Basic Skill to move to a position or a gripper has the Basic Skill to grip or release. This abstraction makes it possible to replace different resources with other resources of the same type since the Basic Skills are identical. For instance, a gripper of vendor A can be replaced by a gripper of vendor B, because both grippers have the Basic Skills grip and release. In order to be able to perform more complex skills, it must be possible to combine several Basic Skills into more complex skills. For this purpose, there are so-called *Composed Skills*. Composed Skills are executable services that use Basic Skills and supplements them with logical operations or sequences. For example, the Composed Skill to move an object from a start position to a target position can be combined through a sequence of the Basic Skills grip, move to position, and release. Here again, interchangeability is important since the definition of the Composed Skill is not dependent on explicit hardware. For adding Composed Skills or other additional software components there is the possibility in the AAS to add software components at runtime. This allows production processes to be adapted at runtime or add new processes or skills to the system.

Composed Skill

Additional  
Software  
Component

Self-Description

Another important part of the AAS is the description of the resource and its capabilities, the so-called *self-description*. A precise description of the resource is required for easy interchangeability of devices and for finding devices for certain applications. With this detailed description, it is then possible to customize executable skills. If, for example, a gripper is replaced, the new gripping position can be adjusted via the description of the new gripper without having to adjust the logic of the skills. In addition, it is also possible to use the description to search for suitable components for a production process. For instance, a component with a fixed weight needs to be moved, then robot resources can be searched for that have the appropriate payload based on the description. However, not only is the description of the resource essential, but also the description of the skills and the description of the additional software components is essential. In the description of software components, dependencies between software modules are described, and how the corresponding interfaces for the execution of services are defined. Supposed that a description of the Composed Skill move an object has to be made, it must be clearly defined that it depends on the Basic Skills move to position, grip, and release. Furthermore, it must be defined that the parameters of the Composed Skill are the object to be moved and the target position. Therefore, software components that are loaded at runtime must also be equipped with a corresponding description.

AAS Communi-  
cation Interface

The third part of the AAS is the unified communication interface. The communication interface creates a network between multiple devices and allows other elements in the

system to access descriptions and control services of other resources. Each resource AAS has the OPC UA interface for non-real-time critical data, like the transmission of descriptions across multiple AASs. In case data needs to be exchanged between control services, e.g., Basic Skills or Composed Skills, that are distributed across multiple resources, the OPC UA over TSN communication channel is used for the real-time critical transmission of the data.

### 4.3.3 Modular Real-Time Capable Software Components

To ensure real-time execution of skills, Composed Skills and other additional software components, the software components must be designed accordingly. Furthermore, there is the additional requirement that the individual software components can also run on several distributed resources. For example, it must be possible for an application consisting of a robot, a force-torque sensor, and a gripper (as depicted in Figure 4.5) to set down an object in a force-controlled manner. The application would consist of the Basic Skills move to position, release, and measure force, which are available on three different resources. Depending on the resource on which the Composed Skill place object force-based is loaded, the information of the other Basic Skills must be transferred to the corresponding resource via the communication interface without losing the real-time guarantees.

For this purpose, modular real-time capable software components were developed, so-called *Real-Time Services* (see Section 8.4.2 for more details), which can be loaded to a resource at runtime, are executed with real-time guarantees and can call other Real-Time Services, that may be available local or on other resources. All Basic Skills, Composed Skills, and additional software components in the system are implemented through Real-Time Services. This unification of the software components makes it easy to replace or add new software components. By standardizing the service, the service call interfaces are also standardized, which makes distribution easier. Whether the service call is executed locally or distributed, the information transfer between the services varies. In the local case, the shared memory of a resource is used with very low deterministic latencies. In the distributed case, the information of the Real-Time Services are transferred in real-time via the OPC UA over TSN middleware with slightly higher but also deterministic latency.

Real-Time Service

### 4.3.4 Global Knowledge of Resources, Products and Services

In addition to the resources with its AAS and a uniform communication channel, it is necessary to have global knowledge about the system. A global collection point for knowledge about all resources, their skills, the additional software components, and the products to be produced. In Figure 4.5 these global collection points are shown in the upper part above the communication middleware.

Each system requires a *Skill Registry*. The skill registry collects all available descriptions of skills and additional software components. Global requests can be made to the skill registry regarding which resources provide which skills or software components.

Skill Registry

Another member of a system is the *Skill Composer*. The skill composer ensures that

Skill Composer

several skills can be combined into more complex skills. Therefore, the skills have to interact with each other. The skill composer provides additional software component specifications for resources that can be loaded into a placeholder for additional software components of a resource. For instance, there is a software component specification for the Composed Skill move an object. This Composed Skill is dependent on the Basic Skills move to position, grip, and release. If this Composed Skill is to be used in the system, it must first be checked whether all Basic Skills are available in the system and which resources provide these skills, this can be requested from the skill registry. Further, it must be checked whether the resources with these Basic Skills can be connected to each other. For example, a gripper weighing 10 kg cannot be operated on a robot with a payload of 5 kg. Electrical or data interfaces must also be checked to see if a connection of multiple resources is possible. If the connectivity of the resources is guaranteed, the software component specification is loaded into one of the software component placeholders of a resource. Assuming there is a matching gripper and a robot in our system that work together, the Composed Skill is loaded, for example, to the robot resource that has the Basic Skill move to position. This allows direct access to the locally available Basic Skills. For the Basic Skills that are not available locally, a OPC UA over TSN connection is established automatically to the resource holding the required skill. In the example, a OPC UA over TSN connection to the gripper would be established in order to access the Basic Skills grip and release. Subsequently, the Composed Skill can be executed under real-time conditions.

#### Product Registry

Another component of the system is the *Product Registry*. The product registry holds information about the products and parts to be produced. This information can be used to parameterize the skills accordingly. For instance, if a part is to be moved from a start position to a target position, then the gripping position can be derived from the description of the part, or the appropriate robotic systems can be configured to have a payload greater than the weight of the part. This creates a very close coupling between resources, processes, and products, which is extremely important for production management [42, 156].

### 4.4 Related Work

The project openAAS [148] offers a reference implementation for the AAS and especially the interoperability of AASs. These reference implementation contains a technology-neutral model and can be implemented with different technologies. In addition, it is already possible to use OPC UA for secure communication between administration shells. The project offers good starting points for the implementation of administration shells, but unfortunately, there is still no concept for the execution of real-time applications.

Prinz et al. [164, 165] presents a concept for real-time capable I4.0 components that use TSN for the transfer of real-time critical data. Each real-time capable I4.0 component has a submodels to establish TSN connection with other components. There are also real-time capable I4.0 components that provide capabilities. In the evaluation shown, real-time capable rotational and translational axes can return the position and have the real-time operation "move". The axes are controlled with a central real-time runtime

engine via the dynamically established TSN connection. In the approach it was also shown that components can be added and faulty components can be replaced by others with the same capabilities. Unlike the approach shown in this thesis, capabilities are only single operations, and there is no distribution of execution logic. Instead, the execution logic is centralized in the real-time runtime engine.

There are numerous publications and projects dealing with Plug & Produce. For example, the IDEAS project [134] uses a multi-agent system to have self-configuring modules. Here, concepts for self-descriptions of the components with their capabilities are presented. However, the execution of the skills is implemented centrally by so-called Coalition Leader Agents (CLA). In addition, no reference was made to real-time execution. Since the implementation is based on JAVA, no hard real-time can be achieved. A similarly agent-based approach without real-time consideration using OPC UA as communication interface is demonstrated by Bennulf, Mattias and Danielsson, Fredrik and Svensson, Bo [23].

Pfrommer et al. [157] describes a system that creates executable resource-specific actions partially automated, with the help of processes, resources, and product descriptions. This description is used for orchestration and is then executed centrally via service calls. Here, OPC UA is used to trigger these services without real-time communication.

Profanter et al. [170] for instance, have developed a Plug & Produce system that has a device adapter for each hardware component, for example, a gripper or a robot, representing the hardware and a set of skills. Skills can be composed using software components that can call skills of the device adapter. OPC UA method calls and OPC UA client-server communication are used for this purpose. Due to the large non-deterministic latencies of the OPC UA client-server communication ( $> 50$  ms) [215], such approaches are only suitable for applications that are not real-time critical. Since all software components are connected via the OPC UA middleware, scalability to very large systems with many components is difficult.

Ye et al. [207] use AASs that use the Automation Markup Language (AutomationML) for the description of resources. As communication interface OPC UA client-server communication is used. In this work, robotic systems consisting of sensors, tools, and the robot itself are considered as complete systems that are managed with a single AAS. There is also no concept presented for combined capabilities. Also Schleipen et al. [187] use the combination of OPC UA and AutomationML for the realization of Plug & Produce systems without real-time guarantees.

Another project that shows capability-based interoperability of manufacturing resources is the project BaSys 4.0 [1, 153]. BaSys 4.0 aims to develop a reference implementation for Industry 4.0, focusing on the adaptability of manufacturing plants. Several companies from the automation technology sector and research institutes are working together to achieve these goals. The project investigates, for example, how semantic descriptions of resource skills can be used. It also shows how Basic Skills can be orchestrated into more complex skills. The middleware BaSyx created by the project also offers the possibility to manage resources via AASs and communicate via OPC UA [92]. In this project, the main focus is not on the distributed real-time execution of Basic Skills and combination

of skills. However, there are basic considerations that shall also support real-time critical communication via TSN [202].

Walter et al. [204] shows how to use program blocks according to the IEC 61499 standard in combination with OPC UA as communication layer for Plug & Produce. The emphasis here is on decentralized control through a service-oriented architecture. A description and adaptation of the capabilities based on a knowledge base was not taken into account, and although OPC UA Pub-Sub was used for communication, deterministic executions are not possible. Zimmermann et al. [213] follows a very similar approach. Here, the goal is to achieve the combination of skills that are either IEC 61131-3, IEC 61499, or C/C++ programs. These skills can be combined into more complex higher-level skills provided by a OPC UA server. Finally, a superordinate control system can orchestrate the skills via the OPC UA interface. However, the paper points out that an integration of TSN is considered useful for real-time critical executions, but this has not been implemented. Koziol et al. [99] have similar approaches and offer the reference implementation OpenPNP for the integration of field devices. The paper also investigated the scalability of such systems with respect to OPC UA client-server communication.



**Summary.** An important aspect of Plug & Produce is adding new components to the system. This chapter describes how resources register with the system through a self-introduction. Furthermore, it is described how the individual resources' self-description is structured and how a description of the entire system can be formed from these self-descriptions of all resources.

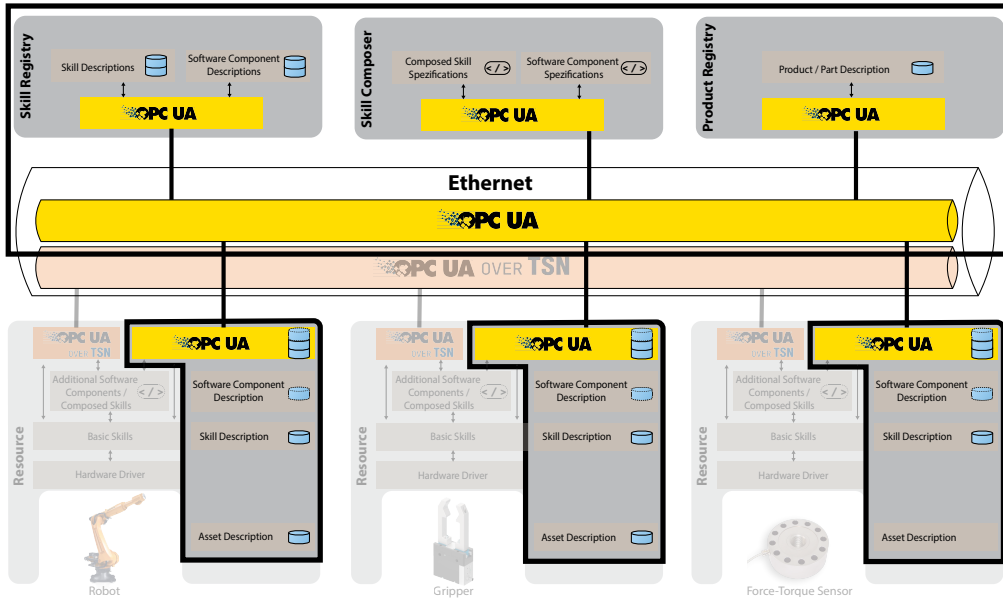
# 5

## Locating and Managing Plant Components

<b>5.1</b>	<b>Fundamentals</b>	<b>45</b>
5.1.1	Semantic Descriptions	45
5.1.2	OPC UA Discovery	46
5.1.3	OPC UA Extension Objects	47
<b>5.2</b>	<b>Plug &amp; Produce Resources</b>	<b>47</b>
<b>5.3</b>	<b>Global Registry for Plant Components</b>	<b>49</b>
<b>5.4</b>	<b>Semantic Self-Description of Plant Components</b>	<b>51</b>
<b>5.5</b>	<b>Consolidation of Information into a Uniform Knowledge Base</b>	<b>52</b>
<b>5.6</b>	<b>Related Work</b>	<b>54</b>

In order to implement Plug & Produce so that resources can be easily added, replaced, or removed during runtime, locating and integrating new plant components is essential. Not only when people unknown to each other shall work together successfully in a meeting a round of introductions is important. A kind of round of introductions is also necessary when integrating new machines and components into production cells. There are already common practices, such as the integration of new Plug & Play devices via USB on the computer. This chapter describes how discovery mechanisms already integrated in OPC UA can be used to implement a Plug & Produce approach and to locate and manage new resources in a system. For this, it is important where the information about the individual resources is located and how these can be integrated into a uniform knowledge base so that, subsequently, this knowledge base can be used to make the entire system more flexible.

This chapter is organized as follows. First, the fundamentals needed for locating and managing new plant components are described (see Section 5.1). For this purpose, the basics of semantic descriptions are described, and it is explained how OPC UA-capable devices can be found in networks and how larger amounts of data can be transferred



**Figure 5.1.** System architecture for a RealCaPP environment focusing on the description of the resources, the discovery via OPC UA and the global knowledge bases (registries) (cf. Figure 4.5)

via OPC UA. In Section 5.2, a description is given of how Plug & Produce resources are implemented, distinguishing between active and passive Plug & Produce resources. Subsequently, Section 5.3 presents how these discovery approaches can be applied to the RealCaPP architecture in order to integrate new resources. Each resource contains a self-description, how this self-description is structured is defined in Section 5.4. It is described how the information of the individual resources can be merged into a uniform data model (see Section 5.5). Finally, in Section 5.6, it is shown what related work is available for discovering and managing industrial components.

In Figure 6.1, the overall RealCaPP architecture, as already illustrated in Figure 4.5, is shown with the highlighted focus of this chapter. In this chapter the focus lies on adding resources to the system. Each resource has an description from itself, that contains the description of the asset, the skills and the additional software components as shown in the lower part of the figure. When registering the resources via the OPC UA discovery mechanism, the descriptions are transmitted via OPC UA client-server communication. Therefore, the non-real-time middleware is highlighted. Subsequently, the data is collected and processed in global registries shown in the top part of Figure 6.1. In addition, the chapter describes how the individual registries are interrelated.

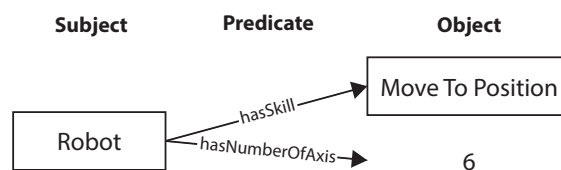
## 5.1 Fundamentals

Before describing how new plant components can be located and managed, a summary of some of the techniques used to discover and describe components is presented.

### 5.1.1 Semantic Descriptions

The *Resource Description Framework (RDF)* was originally a general framework for describing any internet resource, such as a website and its content. With RDF, the linking structure of the web was extended to name relationships between things. This allows to represent structured and semi-structured data. [119] However, this technology can also be used to network information in other areas, e.g., in automation technology [62].

RDF describes data relationships as triples, with a subject, a predicate, and an object. Here, a subject is a resource, the predicate is the named link, and the object can either be another resource or represent a value. A set of these triples forms a directed, labeled graph. The edges in the graph represent the named link between two resources that are represented as nodes. Figure 5.2 shows an example RDF graph describing a robot with six axes and the skill to move to a position. This example also shows that objects can reference other resources (e.g., Move To Position) as well as contain literals (e.g., 6). [44, p. 15]



**Figure 5.2.** Example of an RDF-Graph

The *RDF Schema (RDFS)* extends RDF with a uniform vocabulary. The vocabulary contains a uniform representation of classes and properties so that the information on resources can be stored in a manner similar to object-oriented programming. Class hierarchies, for example, can also be represented using RDFS. [29]

An *Ontology* was initially the philosophical study of being and has later become increasingly popular in computer science [115, p. 1964]. Gruber provides a good definition for an ontology in the sense of computer science:

*“A specification of a representational vocabulary for a shared domain of discourse – definitions of classes, relations, functions, and other objects – is called an ontology.”* (Gruber [63])

An ontology can thus be used to model a knowledge base about resources, their attributes, and their relationship to other resources. Ontologies also have the advantage that they are independent of the actual implementation of the system. In terms of the power of expression, ontologies are very close to the expressiveness of first-order logic. Due to

RDF

RDFS

Ontology

these properties, ontologies are particularly suitable for integrating heterogeneous data sources and the interoperability between different systems. [115, pp. 1965]

OWL

RDF and RDFS define a syntax with a uniform vocabulary for the knowledge representation and thus provide a good basis for an ontology, but a way to describe dependencies between relations is still missing. The *Web Ontology Language (OWL)* supplements these techniques with rules to represent dependencies between relations that are machine-interpretable. OWL allows equivalences to be represented, boolean operators as intersections, unions, and complements. Also, explicit quantifiers for properties and relations are possible. Furthermore, properties for relations can be applied, such as reflexivity, transitivity, or symmetry. [124] The representation of equivalences makes it possible to combine several ontologies to build up an even larger knowledge base with comprehensive relations.

Reasoning

The semantic of data can be extended in two ways: the first one is the tagging of information to get a common understanding of the meaning and the second way is inferring meaning through logical constructs. This inference of meaning is called reasoning [44, pp. 61]. Therefore, reasoning uses the description of relations by description logic and can thus derive further conclusions. Let us assume an ontology where a robot is a machine consisting of at least three driven axes. If a machine is subsequently added to the ontology and six driven axes are added to this machine via relations, it can be inferred via reasoning that the machine is a robot. Reasoning can be used not only to infer but also to verify constraints that have been set. For example, if a robot is added to the ontology with only one driven axis, the ontology will recognize that the conditions that a robot must meet are not satisfied. To reason OWL statements, there are adapted description logic reasoners such as Pellet [193] or HermiT [129]. Both have been extended to handle all specifications of OWL.

### 5.1.2 OPC UA Discovery

OPC UA LDS

OPC UA LDS-ME

GDS

The OPC UA specification on discovery and global services [140] describes how OPC UA applications can find other OPC UA servers and how to connect to the discovered servers. For this purpose, discovery servers are used, which can be divided into three categories: There is the *Local Discovery Server (LDS)* that holds the discovered information of the host the server runs on. The second category is the *Local Discovery Server with Multicast Extension (LDS-ME)*, which can discover information about applications and servers on the local multicast subnet. Moreover, there is the *Global Discovery Server (OPC UA GDS)* that can discover servers and applications in an administrative domain. For example, the GDS can be used for the discovery across the company boundaries in different location-independent company sites.

Since only local networks are considered for implementation in a real-time Plug & Produce scenario, focus is limited to LDS-ME. Each OPC UA server in the system must know the LDS-ME, e.g., the address of the LDS-ME must be known. The newly added OPC UA server can register with the LDS-ME via this address using the `RegisterServer` service. For this purpose, the address of the new server and the server capability identifier are transferred to the LDS-ME. These server capability identifiers are globally

known OPC UA features, for example, if a server can provide data or supports a special information model. This information is stored in the LDS-ME. A client can now make a request with the FindServersOnNetwork service of the LDS-ME to receive information about which servers are available on the local network. Subsequently, the client receives a list of addresses with available servers. These addresses can be used to query the endpoints in the server (GetEndpoints service) and then to establish a connection (CreateSession service) and retrieve or send the information. [140]

### 5.1.3 OPC UA Extension Objects

In order to not only rely on the expressiveness of the OPC UA information model, it must also be possible to use other descriptions, such as semantic descriptions represented by RDF or OWL. Therefore, the OPC UA datatype *OPC UA Extension Objects* are used. Extensions objects are a sequence of bytes identified by a NodeId and information about the encoding and the data length. This data type, therefore allows the integration of other information and data into the OPC UA environment, and there is no need to use other data transfer protocols. [142]

## 5.2 Plug & Produce Resources

Basically, resources such as a robot or a gripper can be divided into two classes. There are active Plug & Produce resources and passive Plug & Produce resources. *Active Plug & Produce Resources* are a fixed combination of the resource itself, for example, a gripper, and its AAS, with the description of the resource, the corresponding communication interfaces, skills, and the drivers. This combination allows the components to integrate themselves. After being plugged into the system, they can actively communicate with other system participants and components and introduce themselves to the system.

Active Plug &  
Produce Resource

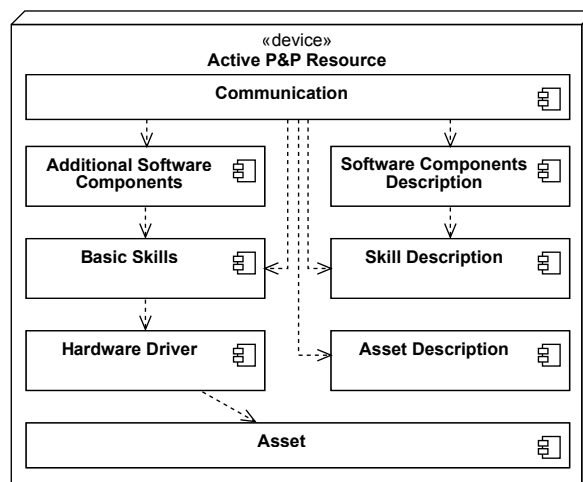
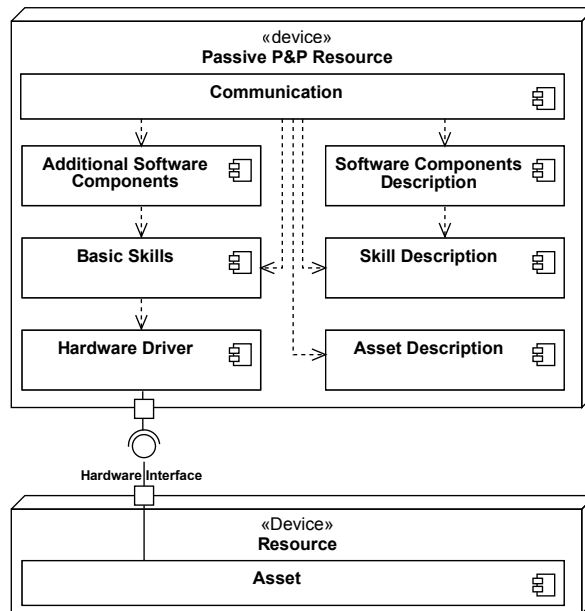


Figure 5.3. Deployment diagram of an active Plug & Produce resource

Active Plug & Produce resources are thus capable of self-introduction. Figure 5.3 shows the deployment diagram of an active Plug & Produce resource. As described in Section 4.3.2, a resource consists of several components. Components are the asset itself, hardware drivers to interact with the asset, Basic Skills of the resource, additional software components like Composed Skills, descriptions of the asset, the skills, and additional software components, and the communication to interact with the system. With active Plug & Produce resources, all these components are executed on a single device.

Passive Plug &  
Produce Resource

*Passive Plug & Produce Resources* implement a concept to separate the asset administration shell from the actual asset. This means that even resources, like a gripper, that are not actively Plug & Produce capable can be used with the help of asset administration shells on another device. Separating the asset from the AAS also makes it possible to make small hardware components Plug & Produce capable. Especially with small devices, for example, a small sensor, it is not physically possible to attach an active control component to the sensor that has the computing power to execute skills, store descriptions, and is also TSN and OPC UA capable. Even with robotic end effectors, such as a gripper, it can happen with small robots that the cables through the robot are limited. For example, it is not possible to route a power cable and an ethernet cable through the robot arm. In this case, the passive Plug & Produce resource can be attached to the base of the robot, and only the hardware interface needs to be routed through the robot to the gripper. With a passive Plug & Produce resource device, connecting



**Figure 5.4.** Deployment diagram of a system with an passive Plug & Produce resource in combination with another hardware device

multiple assets is possible. For example, complete systems consisting of multiple assets can be managed. If a resource consists of a robot with a fixed force-moment sensor, it is possible to use a passive Plug & Produce resource to manage the AAS of the robot and the AAS of the sensor on one device. Figure 5.3 shows the deployment diagram of a system with an asset and a passive Plug & Produce resource. The passive Plug & Produce resources contain the same components as the active ones, except for the asset itself. The asset component can be deployed on another device and is linked to the passive Plug & Produce resource via the hardware interface. The hardware driver component can implement this interface and thus have a continuous architecture. As with the active Plug & Produce resource, self-introduction is possible. Furthermore, it is possible to register passive Plug & Produce resources before the actual hardware device is available to the system. An example is a robotic tool lying on a tool stand. This means that the tool is not capable of communicating with the system. With an automatic tool changer on the robot, the tool can be picked up in the process, and a communication channel to the tool can be created. If a passive Plug & Produce resource is already available for the tool, then the skills of the tool can be used directly afterward. In case the passive Plug & Produce resource is loaded, although no hardware device is available, it is noted in the self-description that no hardware is available yet. This leads to the fact that some skills are not executable. As soon as the hardware is connected, the skills are unlocked.

### 5.3 Global Registry for Plant Components

For adding new plant components, for example, a new robot or tool of a robot, a *global registry* is used for the entire Plug & Produce system. In the global registry, all information about the entire system is collected. This includes information about all resources available to the system, information about the products and parts to be produced, and information about the skills of the individual resources and composed skills consisting of multiple skills of one or more resources. The global registry thus combines the **resource registry**, **skill registry**, and **product registry** with each other. Figure 5.5 shows the component diagram of the global registry. The individual registries provide interfaces for accessing the information. In addition, there are the dependencies between the registries. In this way, the resource registry provides all the skills that are available from the available resources. The product registry also provides the skills needed to produce a part or product. In the skill registry, the matchmaking between the

Global Registry

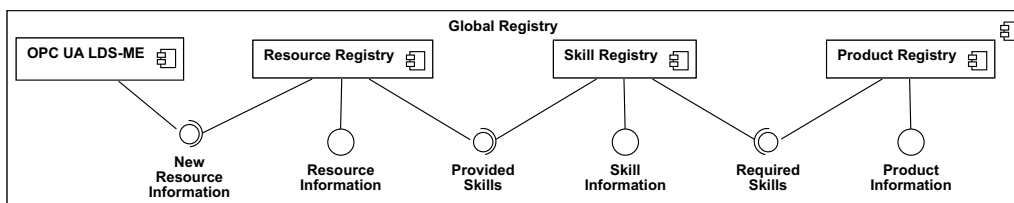


Figure 5.5. Component diagram of the global registry

provided and required skills is done. In addition, the global registry is the starting point for each new resource through the OPC UA LDS-ME.

Adding New Resource

In order to add a resource to the system, the resource must simply be connected to the network where the global registry is located and start with the registration of the resource. This process is identical for active and passive Plug & Produce resources. The registration and adding of the resource to the resource registry is shown in the sequence diagram in Figure 5.6. At first, the added resource needs a unique IP address on the network. Therefore, the resource requests a new IP address from the DHCP server. After the resource has received its IP address, it can register with the OPC UA LDS-ME of the global registry. The only information needed to add the new resource is the IP address of the OPC UA LDS-ME. Once the resource has been added, the IP address of the new resource is stored in the OPC UA LDS-ME. To distinguish between several types of components later on, a new custom OPC UA application type RESOURCE has been added. OPC UA application types are used to identify the different types of applications in OPC UA [139]. This makes it possible, for example, to distinguish between servers,

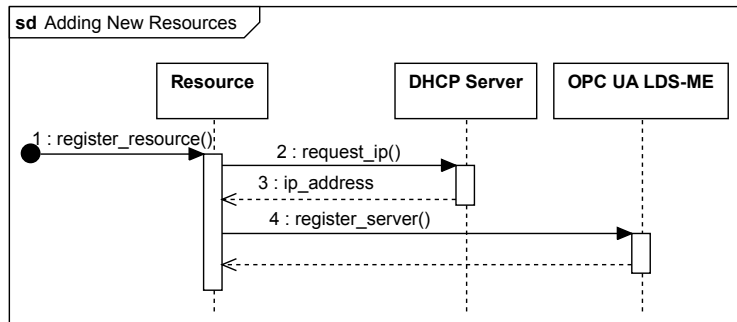


Figure 5.6. Sequence diagram of adding a resource

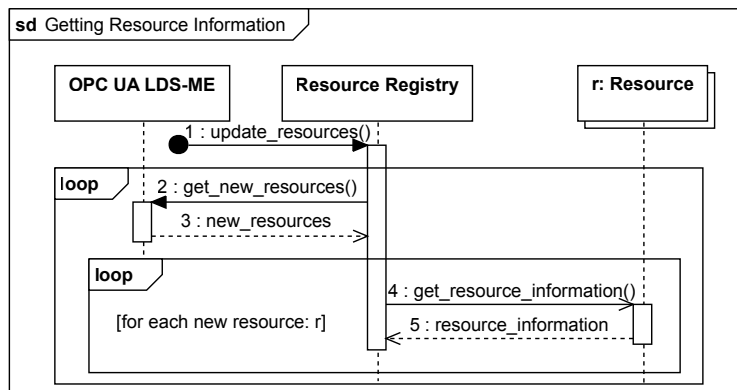


Figure 5.7. Sequence diagram of getting resource information of the new resources



clients or discovery servers. A cyclic update routine is executed to ensure that the resource registry knows when new resources are added to the system, see Figure 5.7. An OPC UA client runs in the resource registry and requests the OPC UA LDS-ME in every cycle what resources have been added. In case new resources have been added, an OPC UA connection to each new resource is established via the stored IP address, and the corresponding information about the resource is collected. Since the collection of information and the establishment of the connection takes a long time ( $> 100$  ms), and there are no dependencies between the resources, the establishment of the connection and the retrieval of the data are parallelized in order to cope with a large number of new resources, especially at system startup.

## 5.4 Semantic Self-Description of Plant Components

With the presented routine for adding resources, it is now possible to establish a physical connection to the resources and retrieve initial information, but for Plug & Produce, more is needed. To reiterate the example of the introduction round at a meeting (see Chapter 5). With the current routine for adding resources, we have just found the right meeting room where the meeting takes place. Nevertheless, to have a successful meeting with people we do not know, we need a round of introductions, a kind of self-disclosure, to learn about the other people involved. The same applies to systems where not all resources are known. Each resource, therefore, requires a kind of self-disclosure in the form of a machine-readable self-description. For the self-description, it is important to get information about the resource itself, what kind of resource it is, e.g., a robot, what properties this resource has, e.g., the payload of the robot, and what skill a resource has, e.g., the skill move to position. This information must be available in such a way that both people, the system, and other resources can work with it. Therefore, techniques like RDF and OWL are very suitable for generating machine-readable semantics. RDF-graphs provide a unified format for representing the dependencies of information. This is supplemented by OWL statements, which can add additional constraints, and conclusions can be drawn based on the information with the help of reasoning.

The two main concepts of self-description are properties and skills. *Properties* are mostly fix defined describing information that belongs to a certain object or thing. The instances to be described can be either resources, skills, or other objects. Many things can be represented with the help of properties. Geometric descriptions can be made, for example, dimensions of a resource. Status descriptions, like the filling level of a container, can be mapped by properties. However, also classifying descriptions or characteristics of skills can be represented by properties. Preconditions for the execution of a skill or execution times would be an example of this.

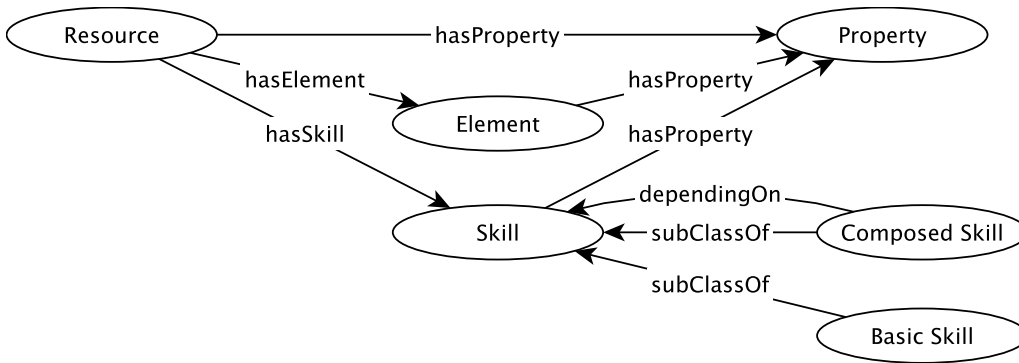
*Skills* are abstract representations of the functionality a resource. Through abstraction and generalization, it is also possible to describe and use the skills of resources without knowing the exact implementation of the skill, which is extremely important for Plug & Produce systems. Since skills can be composed of other skills, it is also necessary to know which skills the Composed Skills depend on.


Figure 5.8 shows an RDF-graph of the basic class structure for describing resources. The

Properties

Skills

Self-Description  
Class Structure



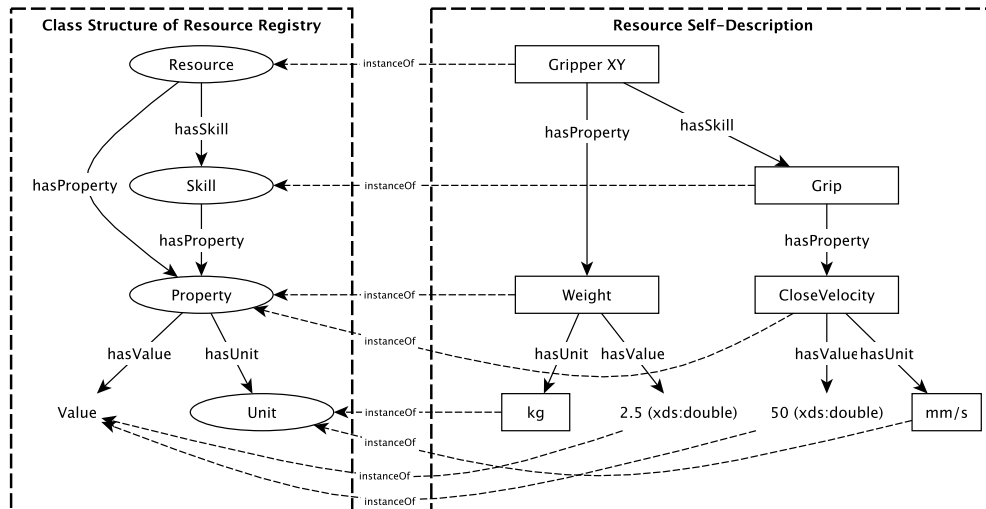
**Figure 5.8.** RDF-graph of the basic class structure of the RealCaPP ontology. Classes are represented as ellipses:  Relations between classes are shown as arrows:  $\longrightarrow$

main class is the resource itself, for example, a parallel gripper. A resource can have relations to skills, properties, and elements. The skills can either be Basic Skills that the resource itself has or Composed Skills. For example, the gripper can have the Basic Skills grip and release. The `dependingOn` relation can be used to map dependencies between skills. Since resources can have relations to other elements, for example, products, product parts, or other objects, the relation `hasElement` was added to the RDF-graph. To give an example, if the gripper grasps a cube, there must be a way to establish the relationship between the gripper and the cube. Furthermore, some properties of resources serve the description of the resource. In the case of the gripper, a property can be the maximum gripping force or the weight of the resource. However, properties can be used not only for describing a resource but also for describing skills and elements. For example, the cube element gets a property of how heavy it is. For the skill of gripping, the velocity at which the gripper jaws close or the maximum gripping force could be added as properties.

With the help of this basic class structure, very detailed self-descriptions of the resources can be created. If all resources are described in this class schema, a uniform description is obtained on the one hand, and on the other hand, it is possible to add further relations to the resources through the relation structure and to use this self-description as a subgraph of a central knowledge base. This also provides a good extensibility of the semantic information.

## 5.5 Consolidation of Information into a Uniform Knowledge Base

In many cases, the information about a single resource is not sufficient. Usually, the interaction between several resources results in a good overall system. As a consequence, it is necessary to collect the required data to consolidate them and thus to be able to make statements about the entire system. For the implementation, a global unified knowledge base is used to collect all information.



**Figure 5.9.** Distributed knowledge base: separation of instantiation on the resource from the class structure on the resource registry. Classes are represented as ellipses: Instances are represented as rectangles: Relations between classes are shown as arrows:  $\longrightarrow$  Instantancements are represented as dashed arrows:  $-\ - \longrightarrow$

As mentioned in Section 5.3, the resource information is passed to the resource registry when a new resource is registered. Due to the uniform class structure of all resources, it is not necessary to transfer the class structures and only transfer the instances without the class definition for the description. As a result, the self-description requires less memory, which is important for embedded systems and thus can be transferred even faster to the resource registry. This has already been shown in preliminary work, see Eymüller et al. [53]. Figure 5.9 shows how the instantiation can be separated from the class structure. The left side depicts the part of the ontology stored on the resource registry with classes and relations for descriptions of resources. The right side describes an explicit resource with its skills and properties. In the example shown, a gripper is described that has the skill to grip. In addition, it is known that the gripper has a weight of 2.5 kg and can close its gripper jaws at a velocity of 50 mm/s. Combining the abstract and concrete parts of the ontology makes it possible to create a knowledge base of the entire system. OPC UA extension objects are used for the transmission of the self-description, which contain the descriptive RDF triples for a resource in XML format. This means that no additional communication path has to be established for the transmission of the self-description, and OPC UA client-server communication can be used. In the resource registry, all instantiations of the resources used in the system are collected and loaded into a global ontology. In addition, information such as the IP address of the resource is added to the overall ontology to enable quick access to the resource later on. Moreover, connections between the resources are also mapped in the ontology. For example, if a gripper is connected to a robot, this connection

must be represented. These connections can either be set manually but can also be set automatically. If an automatic tool changer is used, which can switch between several tools, the connection can be set automatically when a change operation is performed. With the help of the globally established knowledge base, it is now possible to find specific resources with corresponding skills or properties in the system. But not only single resources can be found via the connection of the resources, also information about resource groups can be queried. For example, it is possible to query the total weight of a robot-gripper combination, which results from the individual weights of the combined resources.

## 5.6 Related Work

Numerous works also use OPC UA discovery to find components in the sense of Plug & Produce. Especially in the OpenMOS project [2], several approaches have been published on how new components can be found and integrated into a system with the help of OPC UA LDS-ME. Profanter et al. [167] also use OPC UA LDS-ME for locating new components in the network. They also describe that they use a staggered approach in which devices are grouped into workgroups in a separate network, which in turn are discoverable in a second network (common manufacturing service bus). However, the focus was only on findability and not on the exchange of later information. Later on, Madiwalar et al. [117] extended the system with Software Define Networking (SDN) to include the configuration of networks based on data flows. In addition, the approach contains information about skills, which are stored in the OPC UA information model of the device and can be queried after discovering the device. Nevertheless, it was also not shown how more complex descriptions can be transferred.

For the description of components, there are different approaches with ontologies. Lemaignan et al. [112] describe the MANufacturing's Semantics ONtology (MASON), a common semantic net for the manufacturing domain. The ontology focuses primarily on the link between resources, products, and processes. Detailed descriptions of these instances are not yet considered. In the approach shown, there is no mention of skills yet, but processes are mapped as operations, which is very similar to the skill concept. In addition, no concept is presented that makes it possible to combine operations.

Chungoora et al. [38] presents an approach for the description of resources and the link to their capabilities based on a global ontology. The focus is placed on the fact that a lot of information is already stored in other standards, and the ontology is used to link information from different standards in the field of industrial automation systems. This ensures that data is not redundant and that information from existing knowledge bases can also be used. Due to the strict data representation by the standards, it is difficult to adapt the ontology to a given use case. For future considerations, it would be interesting to see how data already represented by standards can be integrated into the overall system.

The Semantic Sensor Network ontology (SSN) [39] is an ontology for representing sensor data and observations in a machine-readable format. In addition to the description of the sensors through a set of classes and properties, a detailed description of the

measurement procedure is also included in the ontology. Unfortunately, the process description is limited to the skill to measure. However, in case these specific concepts can be transferred to other fields. Thus, the detailed model for the description of the measurement processes serves as basis for the description of properties of skills for the ontology shown in this thesis.

The Manufacturing Resource Capability Ontology (MaRCO) [89] was developed by Järvenpää et al.. The MaRCO ontology is a manufacturing ontology that combines resource descriptions, capability descriptions, and a product model into one ontology to enable a reconfigurable manufacturing system. Since many of the considerations and interconnections of the information model are very valid and considerations of combined capabilities are already integrated, this ontology was used as a basis but was slightly adapted to the required needs.

At a basic level, semantic web ontologies are usually a linked knowledge base that is distributed across multiple systems. It is therefore also necessary to consolidate information from different systems and bring it into a uniform knowledge base, for which several approaches have already been shown. Pinto et al. [159] distinguish between two ways to consolidate ontologies: There is the process *integration*, which means that an ontology is integrated into another ontology and there is the *merge* where two ontologies are merged into a single ontology and the data are unified. Especially for the merging of several ontologies, there are numerous approaches [100, 166, 198] These processes are somehow different from the approach shown in this thesis, since in related work multiple ontologies tend to be merged rather than one ontology being distributed is merged. However, some approaches can be adapted for the slightly different use cases.



**Summary.** This chapter demonstrates the usefulness of the data collected. Applying rules and inferences to the database adds value to the data. This added value can be used, for example, to find case-specific plant configurations.

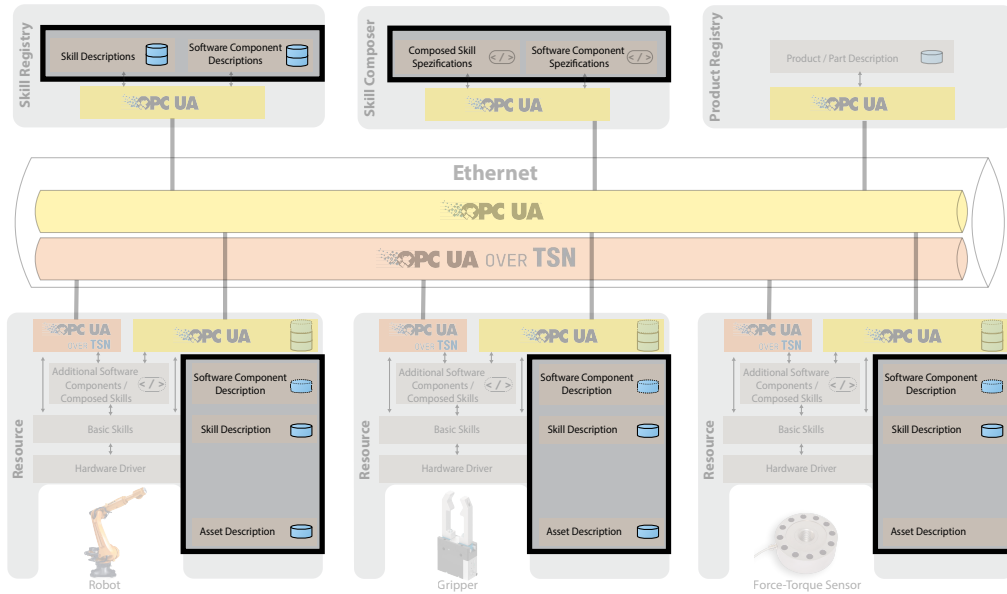
# 6

## Semantic Descriptions of Automation Plants

<b>6.1</b>	<b>Fundamentals</b>	<b>58</b>
6.1.1	Semantic Rules	59
6.1.2	Querying of Semantic Networks	60
<b>6.2</b>	<b>Semantic Description of Resource Interrelationships</b>	<b>61</b>
<b>6.3</b>	<b>Derivation of Connectable and Connected Resources</b>	<b>63</b>
<b>6.4</b>	<b>Deriving Composed Skills</b>	<b>65</b>
<b>6.5</b>	<b>Automatic Plant Configuration through Semantic Networks</b>	<b>66</b>
<b>6.6</b>	<b>Related Work</b>	<b>70</b>

Clive Humby’s quote “*Data is the New Oil*” is often used in connection of Industry 4.0. This statement is true in several respects. It is true that both resources are very valuable and usually difficult to find. However, the more important aspect is that data and oil are of little use value in their raw form. Oil, for example, must be converted into petrol, plastic, or chemicals to create a valuable unit. The same applies to data. Only through linking, analyzing, and interpreting the data the value will increase. This means that once a unified knowledge base of all resources is available, it is important to determine how to combine this information to add value to the system. The networked data can finally be used to generate conclusions for the system via rules and reasoning. By drawing conclusions from the interconnected data, it is possible to query whether the system is capable of executing a given process. The whole thing can even be taken a step further, that it is even possible to find a system configuration via the descriptions which is able to perform a given task.

The structure of the chapter is as follows. First, Section 6.1 will cover the basics needed to create rules based on ontologies and how to query data from graph structures. Subsequently, Section 6.2 explains how ontologies can be used to describe relationships between resources. On the basis of the ontology, it is possible to deduce which resources are connectable to each other and in existing configurations, also to deduce



**Figure 6.1.** System architecture for a RealCaPP environment focusing on the description of the resources and the knowledge base of the skill registry (cf. Figure 4.5)

which resources are connected to each other, see Section 6.3. Section 6.4 illustrates how skill combinations can be derived via rules. Based on the ontology and through the corresponding derivations, it is possible to create automatic plant configurations based on the knowledge base as shown in Section 6.5. Section 6.6 discusses related work in this research topic.

Figure 6.1 shows the overall RealCaPP architecture already shown in Figure 4.5, with the areas relevant to this chapter highlighted. This chapter focuses on how the collected data of the resources can be used, extended and queried. The aim is to create added value for the plant from the data of the individual resources (see descriptions of the resources). By rules and queries on the data, it is possible to apply Composed Skills to a given hardware configuration (cf. Skill Composer). In addition, an approach is presented for using this data to automatically generate plant configurations for a given skill set using the Skill Registry.

## 6.1 Fundamentals

Before describing how the information of the individual resources can be actively used to describe the entire system, the fundamentals are created, which techniques can be used to define rules on the knowledge base, and how the knowledge can be queried.



### 6.1.1 Semantic Rules

Often, the expressiveness of OWL is not sufficient to derive more complex relationships of properties in ontologies. For this, there is the possibility to add additional rules in the ontology. One way to add rules for deriving properties are *property chains*. Property chains are chains of properties that are combined in a new property. Reasoner can then collapse that chain into a single predicate, facilitating access to data via a specified shortcut. Equation 6.1 shows an example of a property chain of the relation `hasGrandparent`. If a node `N1` in the graph has the relation `hasParent` to another node `N2` and `N2` also has a relation `hasParent` to a node `N3`, then the reasoner inserts the relation `hasGrandparent` between `N1` and `N3`. [124]

$$\text{hasParent} \circ \text{hasParent} \rightarrow \text{hasGrandparent}$$

**Equation 6.1.** Example property chain for the relation `hasGrandparent`.

Property chains can simply be used to summarize chains, but often, it is also necessary to be able to map conditions or multiple dependencies between relations. For more complex rules, the *Semantic Web Rule Language (SWRL)* [74] can be used. SWRL is an expressive rule language that combines the power of the semantic web and the Rule Markup Language (RuleML) [27] and provides a high-level abstract syntax for Horn clause rules [75], which can be used in the ontology. The syntax of SWRL is similar to first-order logic, with rules consisting of an antecedent and a consequent. The antecedent delineates the conditions that must be fulfilled for the rule to be applicable, while the consequent outlines the actions or deductions that should be made when the rule is put into effect. By applying SWRL rules, new knowledge can be inferred based on the existing knowledge represented in the ontology. Equation 6.2 shows an example of an SWRL rule. The rule states that if a robot has a weight greater than 200, then it is classified as a heavy-duty robot. So, when applied to a knowledge base or ontology, this SWRL rule classifies all robots as heavy-duty robots for which the conditions apply. [74]

$$\begin{aligned} &\text{Robot}(\text{?robot}) \wedge \text{hasWeight}(\text{?robot}, \text{?weight}) \wedge \\ &\text{swrlb:greaterThan}(\text{?weight}, 200) \\ &\rightarrow \text{HeavyDutyRobot}(\text{?robot}) \end{aligned}$$

**Equation 6.2.** Example SWRL rule for classifying heavy-weight robots.

In addition to the expressiveness and flexibility added by SWRL rules, there is also the advantage that new RDF instances can be created by SWRL with some reasoners. This can be used, for example, to create new instances for groupings derived from rules. The relation `swrlx:makeOWLThing(?var1, ?var2)` is used to create new instances. The

Property Chain

SWRL

Creating New  
Instances

relation has two parameters: the first parameter contains the new instance, and the second parameter specifies on which basis the instances are created. For each instance in the second parameter, a new instance is created. Equation 6.3 shows an example of an SWRL rule where new instances are created. In the example for each existing instance of the class **Robot** a new instance is created. This new instance is then set in relation to the robot by the **hasController** relation. After applying the rule, each robot has a new controller instance. [111]

```
Robot(?robot) ∧ swrlx:makeOWLThing(?new,?robot)
→ hasController(?robot,?new)
```

**Equation 6.3.** Example SWRL rule for creating new instances. For each robot in the ontology a new instance is created and the new instance is linked to the robot instance by the relation **hasController**

### 6.1.2 Querying of Semantic Networks

#### SPARQL

In order to use the information stored in semantic networks represented by RDF-graphs, the graph-based query language *SPARQL Protocol And RDF Query Language (SPARQL)* [172] is used. SPARQL is syntactically similar to the Structured Query Language (SQL) for querying relational databases. In SPARQL, there are four types of queries: There is the classic SELECT query, where the query returns a table in XML format with the queried information. The CONSTRUCT query returns an RDF-graph with the requested information. There is the DESCRIBE query for unknown ontologies, which gives an RDF-graph describing a given with all relations to this resource. In addition, there are ASK inquiries, which return a boolean answer if a query can be answered successfully. As with SQL, not only pure queries can be executed, but also calculations and aggregations on the data. [172] Listing 6.1 shows the basic anatomy of SPARQL queries. If the shown query is executed as SELECT query, a table with two columns is returned containing nodes of the graph, where the first item is connected to the second item over the relation `hasSkill`.

#### Property Path

Another useful feature of SPARQL are *property paths*. Property paths are possible routes in a graph between two graph nodes. In the simplest case, two graph nodes are directly connected via a property, as shown in the upper example. Property paths allow to search for paths of arbitrary length in the graph. A path description is from the definition very similar to regular expressions, with the only difference being that it is based on RDF properties. For example, a relation `isConnectedTo` is defined, which shows which nodes are connected in a graph. Thus, it is possible to use a property path `?node1 isConnectedTo* ?node2` to check if there is a path between two nodes that has zero or more occurrences of `isConnectedTo`. In addition, further operations are available for the creation of property paths and allow, for example, fixed numbers of occurrences of certain relations. [190]

1	<code>PREFIX name: &lt;...&gt;</code>	} Optional: Declare prefix shortcuts.
2	<code>...</code>	
3	<code>SELECT CONSTRUCT DESCRIBE ASK</code>	} Query type.
4	{	
5	<code>?resource,?skill</code>	} Query result clause.
6	<code>...</code>	(Example: Two variables representing graph nodes)
7	}	
8	<code>WHERE</code>	
9	{	
10	<code>?resource hasSkill ?skill.</code>	} Triple pattern.
11	<code>...</code>	(Example: Node <code>?resource</code> must has a relation <code>hasSkill</code> to the node <code>?skill</code> )
12	}	
13	<code>GROUP BY ...</code>	
14	<code>ORDER BY ...</code>	
15	<code>LIMIT ...</code>	} Optional: Query modifiers.

Listing 6.1. The anatomy of a SPARQL query. (Adapted from [44, pp. 308])

## 6.2 Semantic Description of Resource Interrelationships

After having explained in Chapter 5 how the self-description of a single resource is built up, it is shown how this descriptions can be combined into a coherent ontology to describe a system with all its resources, the skills of the single resource, skills that are composed of the skills of several resources, physical connections of the resources, etc.

Figure 6.2 shows an expanded view of the ontology for the representation of resources and their interrelationships. With the ontology presented, it is now possible to represent physical connections between resources, as is the case with a gripper attached to the flange of a robot. In this case, the gripper and the robot are resources with independent

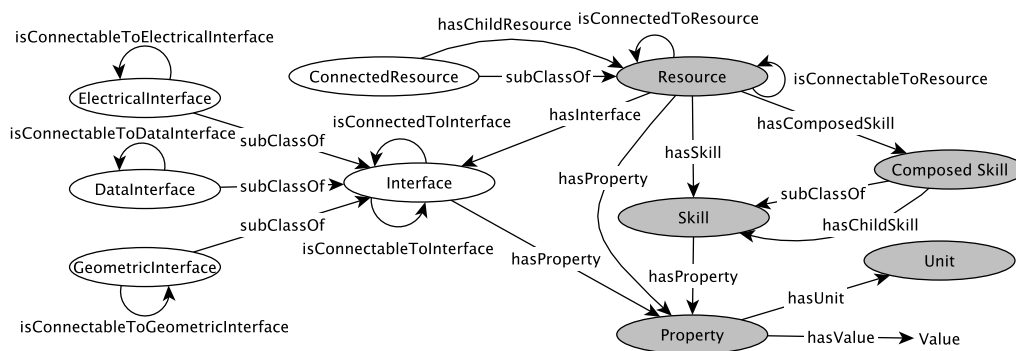


Figure 6.2. Ontology for describing resources and their interrelationships. Classes are represented as ellipses: Relations between classes are shown as arrows:  $\rightarrow$  Already mentioned concepts are greyed out.

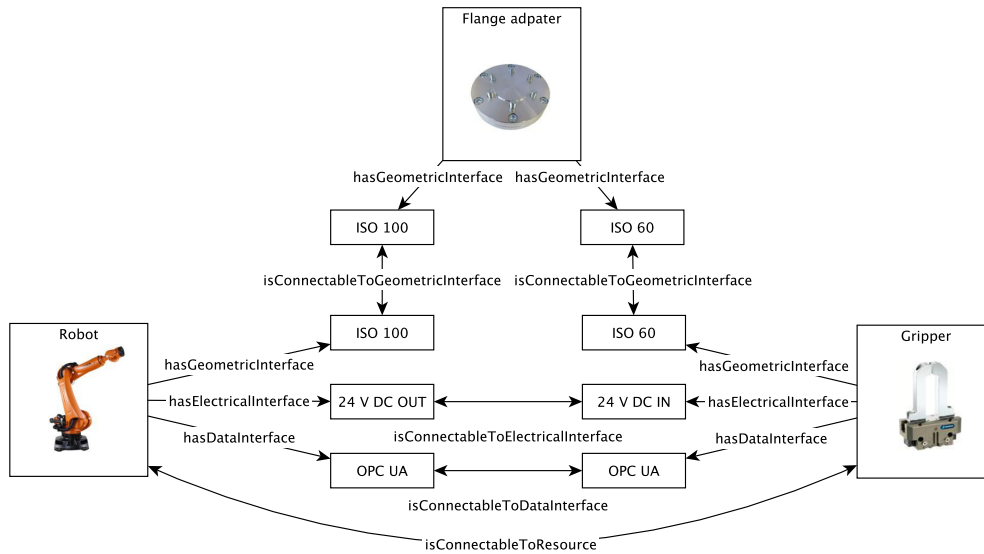
## Interface

skills and properties that can be connected physically via an interface. *Interfaces* can be divided into three interface classes: electrical, geometric, and data interfaces. The geometric interface describes the mechanical fastening between resources. These geometric interfaces vary from automatic tool changing systems over screw connections to plug connections. Now that geometrical connections can be mapped, many resources still need connectors for power and data supply. The electrical interface represents the power supplies. The electrical connections are used to represent different power supplies, e.g., alternating or direct current, and which voltages are present. There is also the possibility to describe data connections over data interfaces. This is mainly for integrating devices that are not yet Plug & Produce capable or OPC UA capable and use proprietary data protocols. However, also OPC UA or OPC UA over TSN data connections can be described with the data interfaces. All interfaces can be described using properties in order to ensure that the interfaces are described as flexibly as possible. Depending on the level of detail, several interface classes can be combined with each other. For example, an interface can be described that uses the EtherCat [83] fieldbus (data interface) as data protocol and transmits the data via an RJ45 connector (geometric interface).

## Connectivity

With the help of the description of the interfaces, connections between interfaces can now also be described. Not only actual connections can be described, but also connections that would be possible in principle. For each interface type, there is a relation `isConnectableTo[...]`, specifying which interfaces are generally connectable. For example, an RJ45 plug can be connected to an RJ45 socket. The actual connection can be displayed using the relation `isConnectedTo[...]`. Not only on the interface level the connectivity description is necessary but also on the resource level, it is possible to describe a physical connection between two or more resources via the two relations. In order to be able to treat connected resources as a higher-level resource, the class `ConnectedResource` has been added.

Figure 6.3 shows an example of a connected resource with a robot and a gripper with the respective interface descriptions. In the simplified example, the robot provides a flange with a hole circle of 100 mm to attach tools. It also has a 24 V direct current power supply and can communicate with other resources via OPC UA. The gripper also can communicate via OPC UA and requires a 24 V power connection. The gripper also has a mounting plate, but only with a hole circle diameter of 60 mm. It is only possible to establish a direct data connection and power supply between robot and gripper. A direct geometric connection cannot be established. Therefore, it is possible that different connection types can also be connected via different paths. For instance, there are often solutions with adapters since some interfaces cannot be connected directly. A flange adapter, as shown in the figure, which reduces the hole circle diameter from 100 mm to 60 mm, is a good example of a geometric adapter, but there are also bus couplers for translating data interfaces or voltage transformers for adapting electrical interfaces. In addition, if the interfaces all match, resources can be described as connectable to each other (see `isConnectableToResource`). This allows to describe that the robot resource can be connected to the gripper resource. For the description of the actual physical connection between interfaces and resources, it would be necessary to replace



**Figure 6.3.** Simplified example of a connected resource with the different interface descriptions. Instances are represented as rectangles: Instance Relations between classes are shown as arrows:  $\rightarrow$  Symetric relations between classes are shown as arrows with two arrowheads:  $\leftrightarrow$

the relations `isConnectableTo[...]` with the `isConnectedTo[...]` relation in the example, but this has been omitted for the sake of clarity.

### 6.3 Derivation of Connectable and Connected Resources

It is possible to manually enter all possible connections describing the connectivity of resources and interfaces, but it is desirable if most connections are generated automatically. For this purpose, rules and relation chains are created that enable automated reasoning of relations. The description of which interfaces fit together can also be used to determine which resources can be connected. If a resource can be connected to another resource geometrically, electrically, and in terms of data, the two resources can be connected in principle.

First of all, the type of connections between resources is checked. For this purpose, a combination of the transitivity property of relations and defined rules is used. The transitivity of the `isConnectableToResource` relation can be used to express the following: If a resource R1 is connectable to another resource R2 and if this resource is connectable to another resource R3, then R1 is also connectable to R3. Now, with the help of the definition which interfaces are connectable, it is derived which resources are connectable. For this purpose, the property chain for the geometrical interface shown in Equation 6.4 is added to the ontology. The rule expresses that if a resource has an interface that is connectable to another interface that belongs to a resource, then the two

Derivation of Connectivity

**hasGeometricInterface**  $\circ$  **isConnectableToGeometricInterface**  $\circ$   
**inverse(hasGeometricInterface)**  $\rightarrow$  **isGeometricConnectableToResource**

**Equation 6.4.** Property chain for the relation, that two resources are geometrically connectable. This rule also exists for electrical and data connections.

resources are connectable. This rule is applied to all types of connectivity (geometrical, electrical, data). With the additional transitivity property, it is now possible to state automatically which resources are connectable with which type of connection. Connections via adapters are also taken into account. Since a resource can only interact with another resource if it is geometrically, electrically, and data-technically connectable, The SWRL rule shown in Equation 6.5 was added to the ontology, which describes the general connectability of resources.

**isGeometricConnectableToResource**(?resource1,?resource2)  $\wedge$   
**isElectricalConnectableToResource**(?resource1,?resource2)  $\wedge$   
**isDataConnectableToResource**(?resource1,?resource2)  
 $\rightarrow$  **isConnectableToResource**(?resource1,?resource2)

**Equation 6.5.** SWRL rule for the **isConnectableToResource** relation, that two resources are generally connectable to each other.

Complementary to the connectivity descriptions, adapted rules, and property chains are used to derive the connection between two resources. This is realized on the basis of the actually connected interfaces. With the derived connected resources, creating a new instance of the **ConnectedResource** class is useful from the compound. The connected resources are added to this new instance as children using the **hasChildResource**. The

**isConnectedToResource**(?resource1,?resource2)  $\wedge$   
**hasSkill**(?resource1,?skill1)  $\wedge$  **hasSkill**(?resource2,?skill2)  $\wedge$   
**swrlx:makeOWLThing**(?new,?resource1)  
 $\rightarrow$  **ConnectedResource**(?new)  $\wedge$  **hasChildResource**(?new,?resource1)  $\wedge$   
**hasChildResource**(?new,?resource2)  $\wedge$  **hasSkill**(?new,?skill1)  $\wedge$   
**hasSkill**(?new,?skill2)

**Equation 6.6.** SWRL rule for the **isConnectableToResource** relation, that two resources are generally connectable to each other.

SWRL rule shown in Equation 6.6 is used to derive instances of the class **Connected-Resource**.

All resources that are physically connected are combined into a new resource, and the corresponding skills of the resources are also combined. In addition, the new resource gets the set of all skills of the individual resources. The individual resources are also added as child resources to the added `ConnectedResource`. Because the class `ConnectedResource` inherits from `Resource` and the `hasChildResource` relation is transitive, `ConnectedResource` individuals can be composed of more than two individual resources.

Thus, if all the resources from the example in Figure 6.3 were connected, a new instance of the class `ConnectedResource` would be created containing the resources `robot`, `gripper`, and `flange adapter` as child resources. If the robot now has the skill `move to a position` and the gripper has the skill `grip` and `release`, the new instance would also have all these skills.

## 6.4 Deriving Composed Skills

Since a description is now available of which components are connected and which are theoretically connectable to each other, it is now possible to derive Composed Skills that consist of several individual skills. To take up the example already mentioned, if a system has the skill to move to a position, grip a component, and release it again, the system also has the skill to move a component to a position by combining the individual skills. This composed skill can also be described as `pick and place`. For the derivation of skill combinations, SWRL rules can be used. In contrast to the connection rules, the difference here is that it must be possible to add these rules at runtime since not all combinations of skills are already available at the design time. Especially when a use case changes and suddenly a process has to be switched to another, other skills may be necessary. The advantage of SWRL rules is that they can be easily added and applied by reasoning.

If a new skill combination is added to the system, only a new SWRL rule has to be created. Equation 6.7 shows a rule for the combined skill `pick and place`. Because a

$$\text{hasSkill}(\text{?resource}, \text{Grip}) \wedge \text{hasSkill}(\text{?resource}, \text{Release}) \wedge \text{hasSkill}(\text{?resource}, \text{MoveToAPosition}) \rightarrow \text{hasSkill}(\text{?resource}, \text{PickAndPlace})$$

**Equation 6.7.** Exemplary SWRL rule for the derivation of the skill `pick and place` from the individual skills `grip`, `release` and `move to a position`.

`ConnectedResource` is a specialization of a `Resource`, combinations of resources may be recognized that relate only to an individual resource and combinations that relate to a group of resources. With the help of the added rules, it is now possible to deduce which resources are theoretically capable of executing a composite skill, but the logic of how the Composed Skill is implemented must also be added. This is shown in Section 8.4.5.

## 6.5 Automatic Plant Configuration through Semantic Networks

Now that it is known how to derive connectivity and Composed Skills from several individual skills automatically, it will be discussed how this can be used to perform automatic plant configurations. The input for the automatic configuration is the specification of a skill that shall be executed. In the simplest case, it is a straightforward search in the ontology which resources have a corresponding skill. However, if there are no resources that provide the required skills, it is necessary to search for combinations of resources that contain the skills. Preliminary work for this is shown in Eymüller et al. [55]. Thereby, the following requirements shall apply for a resource combination:

- All required Composed Skills and Basic Skills must be available in the found configuration
- The configuration should be as minimal as possible, e.g., it should consist of as few resources as possible
- Additional constraints for the configuration must be taken into account. For example, the payload of the robot must be considered.
- Already existing configurations shall be preferred.

If a corresponding task is to be executed with a system, this task must first be broken down into skills. Once the skills are available, a configuration must be found that maps these skills. For example, the task is to move a workpiece from a start position to a target position. This task consists of the Composed Skill pick and place. If Composed Skills are added to the ontology, the skill dependencies are also added, see relation `hasChildSkill`. Thus, it is known that a configuration must be found that has the skills move to a position, grip, and release.

By the preconditions given, which resources are connectable with each other, it is now possible to search for connectable resources that contain the required skills for a task. For this purpose, first, a subgraph is created using SPARQL. Listing 6.2 shows a SPARQL query that generates a subgraph containing all paths of connectable resources that contain resources with the required skills. The listing shows a case where three skills are needed for a task. If more or fewer skills are required for a configuration, the query is automatically supplemented or reduced by the system. To find a configuration that can implement the combined skill pick and place, the placeholders for filtering the skills (see Listing 6.2 ①) must be set accordingly to move to a position, grip and release. Through the creation of a subgraph, the queries remain scalable and can, if necessary, be used for multiple configuration requests. If the query does not return any values (empty subgraph), it is known that no suitable configuration can be found in the knowledge base of the current system.

If a corresponding subgraph is generated, it is known that there is a possible configuration solution. Since sometimes several configuration paths are found, a minimal solution should be preferred, meaning the configuration should be composed of as few resources as possible. For this reason, the subgraph is used to find a minimal solution. Of course, it is also possible to apply other optimization criteria. To use the metric of the connection



---

```

1  CONSTRUCT
2  {
3    ?r_1 hasSkill ?s_1.
4    ?r_2 hasSkill ?s_2.
5    ?r_3 hasSkill ?s_3.
6    ...
7    ?mid1_1 isConnectableToResource ?mid2_1.
8    ?mid1_1 hasSkill ?mid1s_1.
9    ?mid1_2 isConnectableToResource ?mid2_2.
10   ?mid1_2 hasSkill ?mid1s_2.
11   ...
12 }
13 WHERE
14 {
15   ?r_1 isConnectableToResource* ?mid1_1.
16   ?mid1_1 isConnectableToResource ?mid2_1.
17   ?mid2_1 isConnectableToResource* ?r_2.
18   ?r_1 isConnectableToResource* ?mid1_2.
19   ?mid1_2 isConnectableToResource ?mid2_2.
20   ?mid2_2 isConnectableToResource* ?r_3.
21   ...
22   ?r_1 hasSkill ?s_1.
23   ?r_2 hasSkill ?s_2.
24   ?r_3 hasSkill ?s_3.
25   ...
26   optional{?mid1_1 hasSkill ?mid1s_1}.
27   optional{?mid1_2 hasSkill ?mid1s_2}.
28   ...
29   filter(?s_1=<SKILL1>).
30   filter(?s_2=<SKILL2>).
31   filter(?s_3=<SKILL3>).
32   ...
33 }

```

Skills with associated resources.  
 One for each searched skill.

Paths between connectable resources  
 with their skills.

Creates paths between connectable  
 resources.

Skills with associated resources.

Optional: Skills that are available  
 on the resource path.

Filtering the skills needed. ①

---

**Listing 6.2.** Simplified SPARQL query to create a subgraph with resources that are connectable and have specified skills (see placeholder <SKILL1>, <SKILL2> and <SKILL3>).

path length between resources, the SPARQL query shown in Listing 6.3 can be used. This allows to calculate the distance between two resources in the subgraph. Due to the fact that the calculation of the path lengths in large graphs is very time-consuming, the advantage of the subgraph becomes apparent here since the length does not have to be calculated between all possible configurations in a system but only in a subset with matching skills. The query returns a table with configurations containing at least two required skills. A configuration is a list of resources listed in the correct order. For each configuration, the number of resources that make up a configuration, the start and end resource of the configuration chain, the skill searched for in the start and end resources,

---

```

1 SELECT
2   (GROUP_CONCAT( ?mid; SEPARATOR=";"))
3     AS ?config)
4   (GROUP_CONCAT( ?midskill; SEPARATOR=";"))
5     AS ?additionalSkills)
6   ?startresource
7   ?startskill
8   ?endresource
9   ?endskill
10  (COUNT(?mid)
11    AS ?numberOfResources)
12 WHERE
13 {
14   ?startresource isConnectableToResource* ?mid.
15   ?mid isConnectableToResource+ ?endresource.
16
17   ?startresource hasSkill ?startskill.
18   ?endresource hasSkill ?endskill.
19
20   FILTER(?startskill IN (<SKILL1>,<SKILL2>,<SKILL3>,<...>)).
21   FILTER(?endskill IN (<SKILL1>,<SKILL2>,<SKILL3>,<...>)).
22   FILTER(?endskill != ?startskill).
23
24   OPTIONAL{
25     ?mid hasSkill ?midskill.
26     FILTER(?midskill IN (<SKILL1>,<SKILL2>,<SKILL3>,<...>)).
27     FILTER(?midskill != ?startskill
28             && ?midskill != ?endskill)
29   }
30 }
31 GROUP BY ?startresource ?endresource ?startskill ?endskill

```

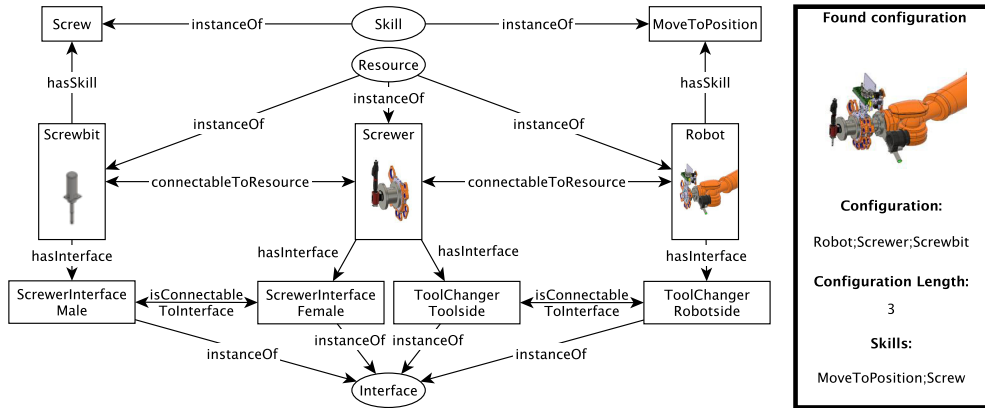
---

Returns resource paths as list.  
 Returns a list of needed skills on the path.  
 Returns the calculated path length.  
 Creates paths between connectable resources.  
 Filtering of the start and end skill. ①  
 Filtering of the skills on the path. ②

**Listing 6.3.** SPARQL query for searching suitable configurations with the path length of the configuration. In the query there are again placeholders (<SKILL1>, <SKILL2> and <SKILL3>) for the required skills.

and the skills searched for contained in the path are specified. As in the example above, the query is with three required skills. The whole code passage ② can be removed if only two skills are needed. The query can be easily extended for more than three skills. For this, only the skill lists in ① and ② of the query must be supplemented. This is indicated by <...>.

Figure 6.4 shows an example of an extracted subgraph for a task that requires the skills move to a position and screw, as a result of the SPARQL query shown in Listing 6.2. The subgraph consists of the three resources Robot, Screwer, and Screwbit, which are connectable to each other. In addition, the interfaces are also given in the example so that it is clear how the connectability was derived. The robot has the skill instance



**Figure 6.4.** Example for a subgraph with the required skills screw and move to a position. For clarification, the interfaces were also specified. The right side shows a possible found configuration. Classes are represented as ellipses: (Class) Relations between classes are shown as arrows: → Symatric relations between classes are shown as arrows with two arrowheads: ↔

?config	?startresource	?startskill	?endresource	?endskill	?numberOfResources
Screw;Screwbit	Robot	MoveToPosition	Screwbit	Screw	2
Screw;Robot	Screwbit	Screw	Robot	MoveToPosition	2

**Table 6.1.** Result table of the SPARQL query for suitable configurations on the example subgraph. Empty columns were omitted.

MoveToPosition, and the screwbit has the skill instance Screw. If the user wants to screw in a screw at a certain position, a configuration with both skills must be found. In order to meet these requirements, the system performs the SPARQL query shown in Listing 6.3 with the required skill set (Screw, MoveToPosition). This results in a table of possible configurations as shown in Table 6.1. In the shown result, only one possible configuration is specified with different start and end resources. From the table, it can be concluded that there is a valid configuration in which the robot is connected to the screw, the screw is connected to the screwbit, and the resulting configuration length is three. It is often not as clear as in this example, and there are several possible solutions, in which case additional measures are necessary.

On the basis of the existing table, it is checked whether a configuration can be determined that contains all the skills required. It may be necessary to combine several configurations that contain a subset of the required skills to obtain a configuration that contains all the required skills. If no further constraints are separated, the configuration with the lowest number of resources can be used for this purpose. If additional constraints are to be taken into account, additional constraints can be queried based on the configurations, and if necessary, it can also be checked whether a corresponding

configuration is also practically feasible. However, these additional constraints are highly case-dependent. An example of an additional constraint would be that the power supply is sufficient to operate the resources, or the payload is sufficient to carry the resources. Since the possible configurations with the respective resource instances are already known, the query for the additional constraints can be executed on the entire knowledge base without performance losses.

In order to be able to use the shown approach for an automated reconfiguration of a system, it is necessary to describe the interfaces accordingly, which interfaces can only be connected manually by a human being, and which interfaces can be connected automatically. As shown in the robot and the screwdriver example, all interfaces can be connected automatically through automatic tool change systems. This classification of connectivity is another possible constraint. Preference should be given to configurations that can be performed automatically.

## 6.6 Related Work

In the field of interface description and derivation of connectivity, there is preliminary work worth mentioning by Siltala et al. [191]. The authors involved the development of an ontology for formally describing hardware interfaces. This ontology defines categories of mechanical, electrical, service, and communication interfaces. However, it does not demonstrate how these interfaces can be configured or parameterized. Furthermore, the ontology predominantly relies on standardized interface descriptions, which poses challenges when dealing with non-standardized interfaces. To address this, another ontology was integrated to consider the relationship between capabilities and components within the overall architecture [192]. Järvenpää et al. [90] also use the descriptions of connections and added rules to find corresponding configurations, but a configuration always contains only one capability. Also, additional constraints like the payload of a system are considered [89]. In addition, no resource chains are considered in connection with capabilities. This makes matchmaking much easier. However, the approaches shown served as a basis.

Several methods use semantic approaches to account for geometric constraints that are useful in component design or automating robot assembly. Early pioneers, Ambler and Popplestone [13], used CAD data to create assembly programs for robots. Building upon this foundation, continuous advancements have been made, incorporating additional relationships among the components to be assembled, the required resources (such as robots or production machines), and the skills possessed by these resources [149, 152]. Typically, these approaches establish a strict one-to-one correspondence between a resource and a skill. However, there are also approaches that assign combinations of multiple components to a single skill [169]. For instance, a robot equipped with various end effectors can perform different skills. Nevertheless, there are limitations even in such cases. For instance, Profanter et al.'s work [169] restricts the use of tools to those compatible with a corresponding tool change adapter, and all tools have identical interfaces in terms of data and power supply. Romiti et al. introduced a reconfigurable collaborating robot (cobot) featuring different joint modules and end

effector modules [182]. However, even in this approach, the individual modules adhere to a uniform interface for mounting, data exchange, and power supply, simplifying the creation of configurations.



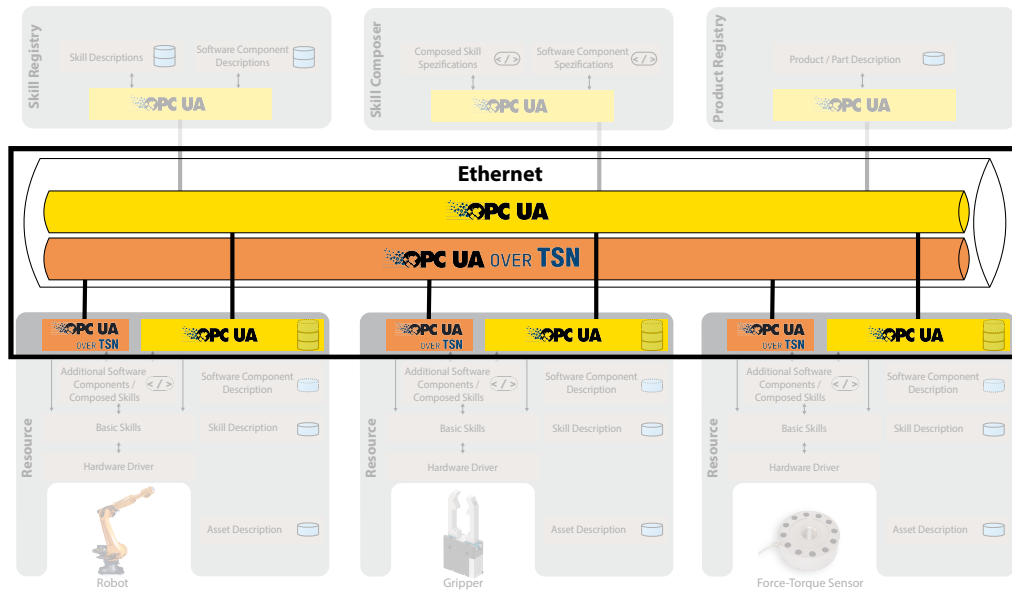
**Summary.** A standard communication protocol is necessary for the exchange of data between resources. This chapter outlines the structure required for a communication middleware that facilitates real-time data exchange between resources. The system has been designed to respond to changes as resources can vary in a Plug & Produce system.

# 7

## Distributed Real-Time: Dynamic Real-Time Control Networks

<b>7.1</b>	<b>Fundamentals</b>	<b>74</b>
7.1.1	Real-time Communication with TSN	75
7.1.2	Precision-Time-Protocol	77
<b>7.2</b>	<b>Time-Synchronization in Control Networks</b>	<b>78</b>
<b>7.3</b>	<b>Dynamic Configuration of Real-Time Control Networks</b>	<b>79</b>
7.3.1	Setup and Configuration of TSN Communication Channels	81
7.3.2	Realization of the Time Slot Array	83
<b>7.4</b>	<b>Related Work</b>	<b>84</b>

Distributed systems consist of multiple interconnected components that need to work together in order to achieve a certain task. Numerous network protocols can be used to exchange data between components. However, if the data is to arrive at the other component within a defined time (real-time), the number of possible protocols becomes smaller. Most of these real-time network protocols are very complex to configure and are usually setup at design time and configured accordingly for a specified topology and use case. In contrast, there are highly flexible and dynamic Plug & Produce systems in which the communication partners can change. For example, new resources can be added, the exchanged data can change by adaption of the task, or the network topology can change due to the scaling of the system. Consequently, more distribution components, such as network switches, may be required. This discrepancy must be resolved to realize industrial-grade Plug & Produce systems. Therefore, it should be possible to reconfigure the real-time production system while it is running. It is necessary to configure new devices and their network interfaces correctly at runtime and to add or remove real-time communication channels as needed. To overcome these obstacles and enable Plug & Produce scenarios with real-time capabilities, an approach is presented that fully relies on the OPC UA communication models with the TSN extension.



**Figure 7.1.** System architecture for a RealCaPP environment focusing on the description of the communication middleware with OPC UA and OPC UA over TSN (cf. Figure 4.5)

The following section outlines the structure of this chapter, delineating the key components and their interconnections. In Section 7.1, the fundamentals of this chapter are first discussed. Based on the Precision-Time-Protocol mentioned in the fundamentals, the time synchronization required for real-time communication is discussed in Section 7.2. Section 7.3 explains the structure for dynamically adding communication participants and how to configure the real-time network channels for the new members. Finally, in Section 7.4, related work is mentioned that deals with real-time communication in general in production environments. that also deals with TSN communication. In addition, related work in the field of TSN communication was considered.

Figure 7.1 displays the complete RealCaPP architecture, which was previously depicted in Figure 4.5. In the context of this chapter, specific sections relevant to the topic have been emphasized. The chapter primarily concentrates on the dynamic communication middleware for the non-real-time and real-time communication between the resources.

## 7.1 Fundamentals

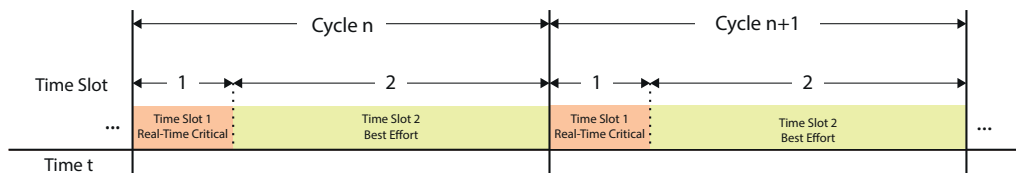
Before discussing communication in distributed Plug & Produce systems, the details of TSN explained. Furthermore, a technology is introduced to ensure a uniform time base in the distributed system. The Precision-Time-Protocol is also one part of the TSN standards.



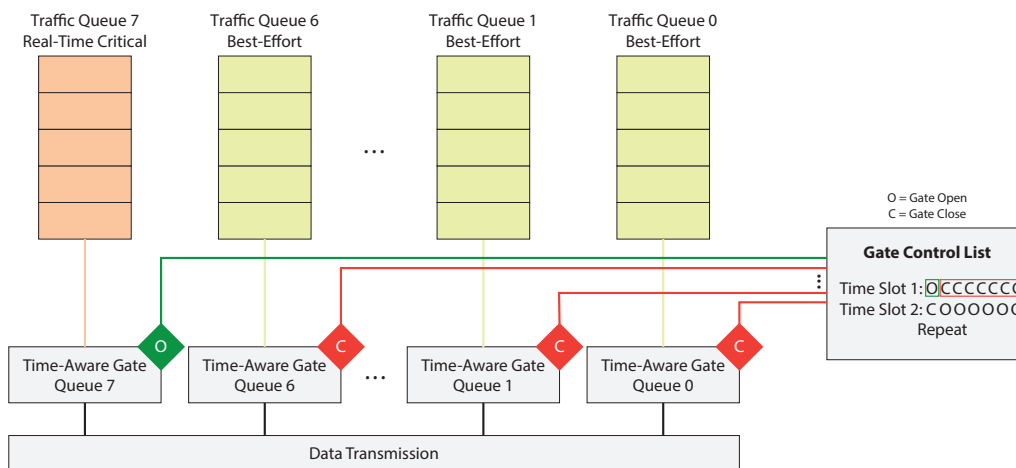
### 7.1.1 Real-time Communication with TSN

After the basics of TSN have been explained in Section 4.1, the technical implementation of TSN and in particular the Time-Aware Shaper (TAS) will be explained in more detail. Figure 7.3 shows the functional principle of the TAS. For the realization of the time slots CoS priorities are used to order the traffic in different traffic queues. For example, real-time critical data have other priorities than best-effort traffic. Each traffic queue has its own time-aware gate. This gate may be either open or closed. If the gate is open, network packets are sent from the corresponding queue via the network card into the network. Otherwise, no network packets will be sent from this queue if the gate is closed. The time-aware schedule determines at which time which gates are open and when they are closed. A gate control list is created. The gate control list defines

Gate Control



**Figure 7.2.** TSN time division multiplexing with reserved time slots to enable the transmission of periodic real-time data. In this example, one time slot is reserved for real-time critical data transmission, and the rest of the cycle can be used for best-effort data traffic. (Adapted from [77])



**Figure 7.3.** TSN Time-Aware Scheduler with the realization of the Time-Aware Gate Control. The example from Figure 7.2 is continued. The real-time critical data is placed in traffic queue 7, and Best-Effort data is placed in the traffic queues 0 to 6. The Gate Control List contains the two time slots and is cyclically executed. In the first time slot only the gate of queue 7 is opened. In the second time slot all other gates are opened, and the gate of queue 7 is closed. (Adapted from [77])

at which time which gates have which status. For the implementation of the example from Figure 7.2, in time slot 1 only the gate of the real-time queue would be open and all others closed, and in time slot 2 the real-time queue gate would be closed, and all other gates open. The gates ensure that only ethernet frames with the appropriate priority are transmitted in the reserved time slot. [77] Since transmissions of an ethernet frame that have already started can only be stopped to a limited extent, a so-called *guard band* is placed in front of each time slot for real-time transmissions. The guard band has the length needed to send an ethernet frame with maximum length (1542 bytes [9]). No further transmissions of best-effort messages are started within the guard band, thus ensuring that no best-effort transmits within a real-time critical time slot. [6] To reduce the size of the guard band, frame preemption can be used. Frame preemption is a method that allows interruption and later continuation of the transmission of a frame. Therefore, an ethernet frame is divided into multiple frame parts, so-called framelets, with a size of maximum 64 bytes. Framelets may be transmitted separately. Thus, each ethernet frame can be interrupted every 64 bytes. This reduces the guard band to the time it takes to transfer 64 bytes. [7]

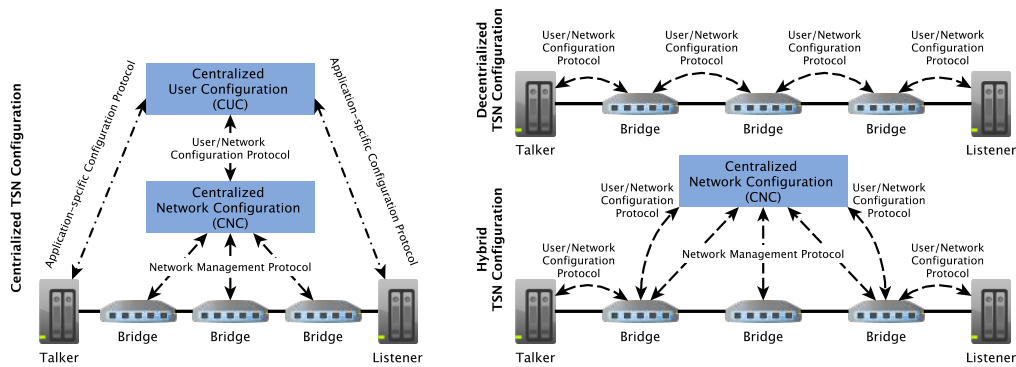
Guard Band

Frame Preemption

Configuration Distribution

The concepts shown only work for a point-to-point connection. If more complex network topologies are used, it must be considered that the time slots for sending real-time critical data are available in each node (endpoint or switch). This enables an end-to-end connection with guaranteed latencies. Since each network component slightly delays the message, it is necessary to know for each component how high the delays are and what the network topology between two communication partners looks like to be able to define time slots for the entire system. Three approaches for configuring the entire network are defined in the standard IEEE 802.1Qcc [8].

Figure 7.4 shows the three configuration approaches. There is a centralized model (left side of Figure 7.4) with Centralized User Configuration (CUC). The CUC manages the communication on the user or application level. The user or the application specifies at



**Figure 7.4.** The different TSN configuration approaches. The left side shows the centralized configuration approach with CUC and CNC. On the right side on the top the decentralized approach is shown. On the right side on the bottom a hybrid configuration approach is shown with a CNC. (Adapted from [77])

the CUC which data is to be exchanged between which devices. This data is processed and passed on to the Centralized Network Controller (CNC). The CNC acts as proxy for the network. Based on the data, who wants to communicate with whom, the CNC determines traffic scheduling, bandwidth allocation, and other network parameters. The individual gate control lists are generated by the CNC and distributed to all TSN participants. In addition to the distributed approach, there are also decentralised approaches where there is no global instance and the network determines its own schedules (top right of Figure 7.4). Of course, there are also mixed forms of centralised and decentralised approaches, which are called hybrid approaches (bottom right of Figure 7.4). [77, 161]

All approaches follow the same basic concept. First, the components that are TSN capable are checked, and the TSN mechanisms are activated. Next, the sending device, called the talker, provides information about the data stream it wants to transmit. This information includes, in particular, identified characteristics such as the destination MAC address and CoS priorities. An end device that is interested in a data stream, the so-called listener, can advertise for the data packets associated with the data stream using the advertised information. Based on this information, the time slots for each individual component are calculated and distributed in the system. [77]

### 7.1.2 Precision-Time-Protocol

The IEEE standard 1588 [10] - IEEE Standard for a Precision Clock Synchronization Protocol for Networked Measurement and Control Systems - defines the *Precision-Time-Protocol (PTP)* as a time synchronization protocol for distributed systems to synchronize clocks over the network. This synchronization is achieved by defining one device as the master clock (grandmaster) and having slaves synchronizing their clocks to the master's time. PTP uses timing messages via UDP multicast with timestamps to exchange the time information between the master and the slaves. In addition, the transmission time between the master and the slave is determined by roundtrip time measurements. On the slaves, the clocks are adjusted to the timestamp of the master by offsetting the time stamp of the master with the transmission time. With PTP, the difference between the synchronized clocks is in the sub-microsecond range.

Figure 7.5 shows the PTP synchronization procedure. Each synchronization broadcast of the master begins at time  $T_1$  with a Sync message to each slave. At  $T_2$  in slave time the slave receives the Sync message and saves the receive time stamp. Then the master sends a FollowUp message with the time stamp  $T_1$ . At this point the slave can calculate the offset  $o$  between the two clocks ( $o = T_2 - T_1$ ). After offsetting the slave clock by  $o$  the clocks would not be completely synchronous. With this correction, the delay  $d$  in the network would not be taken into account. Therefore, the second synchronization part is performed. At  $T_3$  the slaves sends a DelayRequest to the master and saves the time stamp. When the message is received by the master, the time stamp  $T_4$  is recorded by the master. This time stamp is send to the slave by a DelayResponse message. Now the slave has all timestamps:  $T_1, T_2, T_3$  and  $T_4$ . Now the actual offset  $\tilde{o}$  can be calculated. So following applies:  $o = \tilde{o} + d$  and  $T_4 - T_3 = -\tilde{o} + d$ . From this, the actual offset  $\tilde{o}$  can now be calculated as follows:  $\tilde{o} = \frac{1}{2}(T_2 - T_1 - T_4 + T_3)$ . [10]

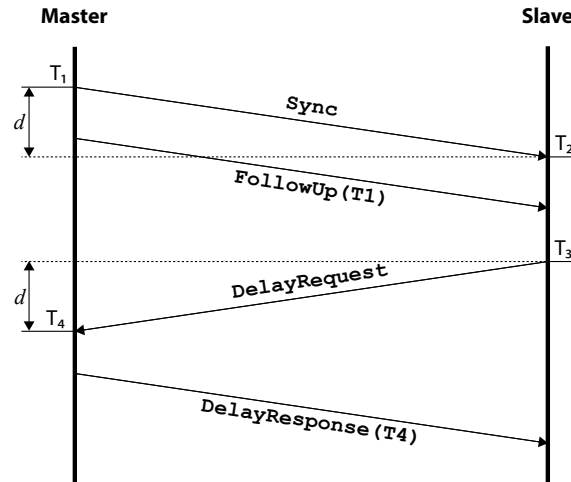


Figure 7.5. PTP synchronization procedure (Adapted from [10])

Time-synchronization is a fundamental requirement for TSN networks. It is needed, for example, for the schedule-based sending of data. As defined in the IEEE standard 802.1AS [18] TSN uses PTP to synchronize the clocks of devices in the local network.

## 7.2 Time-Synchronization in Control Networks

Especially in real-time systems with hard deadlines, clocks are essential. It must be checked whether a deadline has been exceeded, when the next execution cycle starts, or when a sensor value was renewed. If a system runs on a single control component, this is relatively trivial since the system only has one clock to which all programs can refer. A uniform time basis must be created if these real-time systems are to be distributed. This uniform time base is achieved by synchronizing the clocks running on the distributed control components. For this purpose, PTP is used. This time synchronization is necessary for a distributed real-time system, as the following basics are created:

- All components of the system have a consistent and accurate understanding of time, enabling precise timing calculations and meeting deadlines.
- Uniform real-time cycles can be used and monitored in the distributed system.
- Time slots can be defined and reserved for executions and transmissions that are valid throughout the entire system. This is particularly important for implementing TSN [77].
- Time-triggered executions become possible. Executions can be started almost simultaneously.
- Timestamps of data are valid in the entire system. This facilitates or enables the fusion of the data.

Time synchronization is started when new components are added to the system. Components can be new resources, control computers, or added TSN switches. This ensures that all components in the network have a uniform time base. If there is no PTP master in the system, a PTP master is started on the newly added component. If a PTP master already exists, the added component becomes a PTP slave and synchronizes to the master's clock.

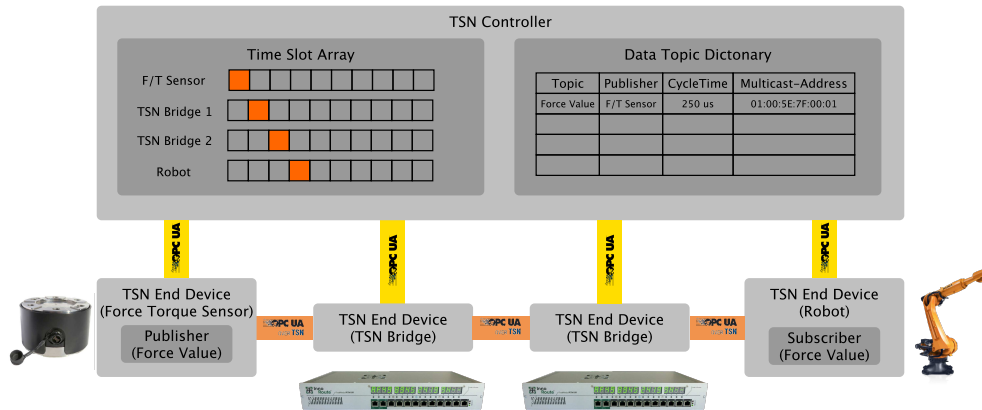
### 7.3 Dynamic Configuration of Real-Time Control Networks

After ensuring that all components in the network have a uniform time definition, a real-time capable communication channel between the added resources must be established and configured in order to establish a distributed industrial robot control. The following contents, how such a dynamic communication platform is setup, have already been documented in preliminary work (see Eymüller et al. [52]). Because OPC UA Pub-Sub over TSN is the basis for this work, the following terms will be used in the further course of this chapter: Network switches with TSN capabilities are referred to as *bridges* or *TSN bridges*. The information released by a publisher is denoted as a *data topic*. These data topics are both published and subscribed to. Multiple subscribers can subscribe to a single data topic. Furthermore, a distinction is made between two types of participants within the system. The first type is the *TSN End Devices*, which can be, for example, a resource (e.g., a sensor or an actuator), a controller or a TSN bridge of a distributed industrial robot control system. The second type of participant is the global *TSN Controller*, responsible for registering new devices and managing the establishment and configuration of real-time communication channels. The default non-real-time OPC UA client-server-communication is used for the configuration of the real-time communication between the TSN End Devices and the TSN Controller.

TSN End Device

TSN Controller

For the discovery of newly added TSN End Devices and TSN Controllers, the OPC UA discovery server is used as shown for adding new resources (see Section 5.3). A new device must first register with the OPC UA LDS-ME to join the system. To facilitate this process, two new custom OPC UA application types were introduced: `TSN_END_DEVICE` and `TSN_CONTROLLER`. This makes it easy to distinguish between resources, TSN End Devices, and TSN Controller in the OPC UA LDS. Once a device is registered, it can retrieve information about other devices from the OPC UA LDS. This allows a TSN End Device to obtain information about the TSN Controller, and vice versa. Once the participants in the system are identified, and information about other devices is available, the configuration of the TSN communication channels can be initiated. OPC UA Pub-Sub over TSN is utilized for real-time transmission of process data, leveraging the Time Aware Shaper (TAS) [6] from TSN to ensure minimal latency in transmission. In order to utilize the TAS, each data topic must be allocated a fixed time slot for cyclic transmission. The allocation of these time slots for each TSN End Device is the responsibility of the TSN Controller. Two operations result in time slot reservations: adding topics for publishing and subscribing to data. When adding a data topic, only the time slot of the publishing TSN End Device needs to be included. On the other hand, when subscribing



**Figure 7.6.** Exemplary structure of a dynamic real-time network with four TSN End Devices. A force-torque sensor is connected over two TSN bridges to a robot. The TSN Controller is the managing instance for real-time communication. The goal is to transfer the force sensor value from the sensor to the robot.

to a data topic, a time slot must be added to the subscribing TSN End Device as well as to every other TSN End Device along the direct communication path between the publisher and subscriber. In case all time slots have been successfully assigned, the reserved time slots are used for the real-time deterministic transmission of the process data. For each device, a corresponding *time slot array* is created, which defines which time slots are reserved and which are free.

In order to be able to provide data topics dynamically and subscribe to data topics, a uniform description is required of which data topics are available in the system. This problem is solved by a *data topic directory*. As mentioned in the OPC UA Specification Part 14 for Pub-Sub [136], a directory is necessary to facilitate the discovery of published data topics and enable other TSN End Devices to subscribe to them. This directory stores information about the publisher of a data topic and the frequency at which the data is transmitted. The directory is centrally stored in the TSN Controller. The directory concept allows TSN End Devices to search for publishers of data topics they want to subscribe to. When a member requires information about a published data topic, it can query the global directory of the TSN Controller. If more detailed information about a data topic is needed, a direct request can be sent to its publisher. The directory follows a structure similar to the OPC UA discovery service, meaning that there are not only central directories but also distinctions between local directories for devices in the same subnet and global directories for communication beyond subnet boundaries.

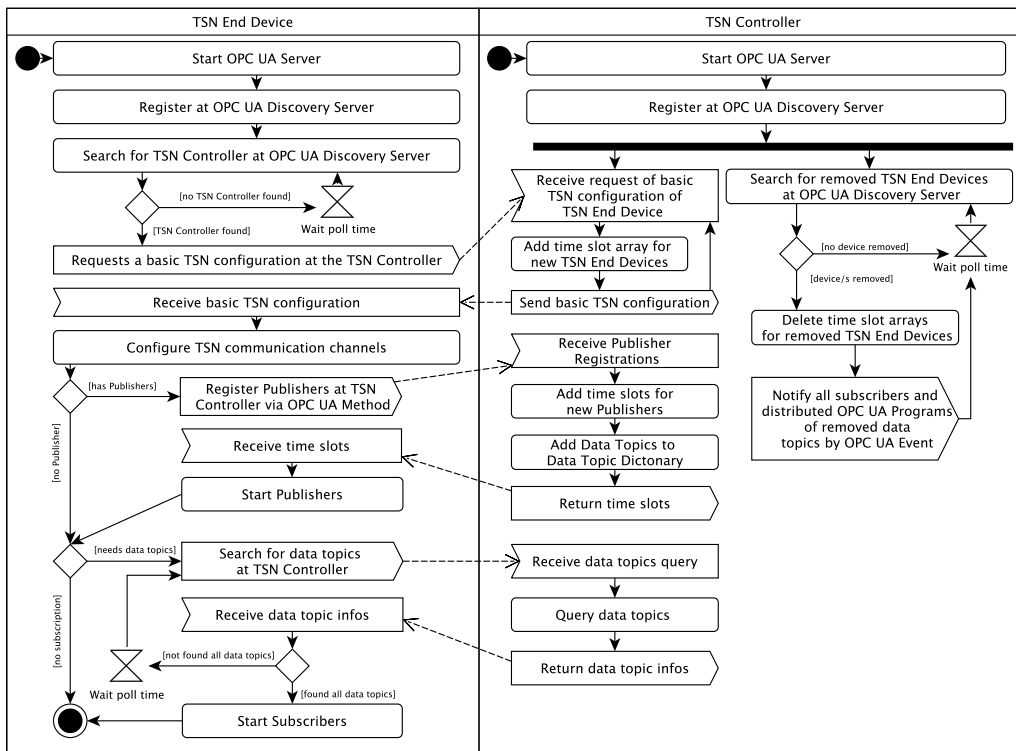
Figure 7.6 shows a sample for a dynamic real-time control network. The network shown consists of four TSN End Devices managed by a TSN Controller. The four TSN End Devices consist of a force-torque sensor, two TSN network switches (TSN bridges), and an industrial robot. The sensor is attached to a TSN bridge, which in turn is connected to another TSN bridge, to which the robot is connected. A realistic use case

in such a setting would be a force-controlled movement of a robot, where a robot moves depending on sensor values. For this, the force value of the sensor would have to be transmitted cyclically to the robot. The force-torque sensor would, therefore, serve as a publisher of the data topic force value, and the robot (Subscriber) would subscribe to this data topic. In this case, it would be the task of the TSN Controller to assign a corresponding address to the publisher, to mediate between the publisher and the subscriber, to find a transmission path between the publisher and the subscriber, and to reserve corresponding time slots for the transmission of the data topic. The required information is stored in the TSN Controller. In the data topic directory, all possible data topics are stored with the available information, and in the time slot array, the current allocation of all time slots for each participating TSN End Device is stored. How the configuration and storage work in detail is explained below.

### 7.3.1 Setup and Configuration of TSN Communication Channels

First, the implementation of the two main components is explained. Figure 7.7 shows the activity diagram of the main processes of the TSN End Devices and the TSN Controller, and how they interact with each other. In the TSN Controller, first of all, the basic parameters for the TSN communication have to be set. This contains the global TSN cycle time in which the data is exchanged and a multicast address range. Each TSN communication participant requires a unique multicast address for publishing data topics. When starting a component, an OPC UA server is started, which then registers with the OPC UA LDS. TSN End Devices register as application type `TSN_END_DEVICE` and TSN Controller as application type `TSN_CONTROLLER`. After completing the registration process, it becomes possible to query devices and establish a non-real-time communication path between the components. When the TSN Controller successfully registers, two parallel tasks are started. The first task involves receiving registrations for new TSN End Devices, allocating and reserving time slot arrays for these devices, and sending them a basic TSN configuration. The second task is to cope with removed TSN End Devices. In this case, the time slot array of the removed TSN End Device is deleted, and all TSN End Devices that have subscribed to a data topic of the removed device are notified through an OPC UA event.

After the basic implementation of the TSN Controller is explained, the implementation of the TSN End Devices is introduced. The TSN End Device searches for a running TSN Controller after successful registration at the OPC UA discovery. In case a TSN Controller is found, a basic TSN configuration is requested. This basic TSN configuration contains information about the global set TSN cycle-time and a start time of the TSN cycle, which is necessary for the synchronization to the TSN cycle. Next, the basic configuration of TSN communication channels is performed. This involves preparing the TSN-capable network interface for real-time message transmission by adding appropriate schedulers, for example, the Earliest Tx-time First scheduler (ETF), to the transmission queues (TX-queue) of the network interface. Additionally, different virtual networks (VLANs) are established for transmitting real-time data, ensuring that messages with varying real-time priorities can be sent. Upon completing the basic configuration of the communication channels, the TSN End Device registers all its data topics to the TSN



**Figure 7.7.** Activity diagram of the management of TSN End Devices by the TSN Controller . It shows that the startup routine of the TSN End Device is combined with the main sequence of the TSN Controller. (Adapted from Eymüller et al. [52])

Controller. The OPC UA method RegisterPublisher of the TSN Controller has the name, the data type, and the cycle time of the data topic as input parameter and returns an updated time slot array for the TSN End Device and a unique multicast address. When registering the publisher with the TSN Controller, an entry with the specified data is added to the data topic directory, the new time slots are reserved for the publisher in the time slot array, and a free multicast address is reserved for the publisher from a defined range. Afterward, the publisher is configured and started in the TSN End Device with the received multicast address and the corresponding time slot.

After the publishers have been added, the TSN End Device can request the required data topics by its data topic name from the TSN Controller. If the required data topics are found, the TSN Controller returns the information required to start a publisher. This information contains the multicast addresses of the topics and the offsets of the reserved time slots for subscribing to the data topics. If all requested data topics are found the subscriber start. Registering a subscriber may result in the reservation of time slots on multiple devices, depending on the number of hops between the subscriber and the publisher. The TSN Controller will notify all relevant hops and set up the appropriate forwarding of the data topic on all affected devices.

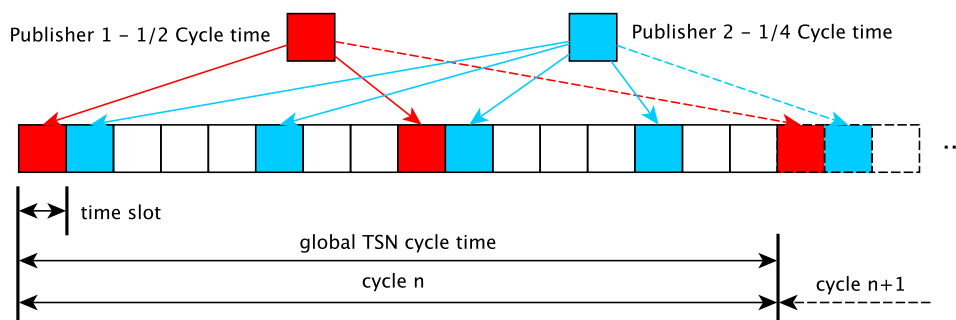


### 7.3.2 Realization of the Time Slot Array

In the preliminary work (see Eymüller et al. [52]), a simple algorithm was presented, which can be used in small systems with one to three hops. The attempt tries to reserve time slots iterative, one after the other. Only if no more possible time slot solution is found the time slots are rearranged. This is only suitable to a limited extent for large systems, but there are already other approaches to this problem, as shown in Section 7.4. The provided example in Figure 7.8 illustrates how a time slot array for a TSN End Device is implemented. Initially, the overall TSN cycle time is divided into time slots of simultaneous size. To achieve this division, a power of two is chosen as the divisor for the global cycle time ( $divisor = 2^m$ ). In this specific example, the global TSN cycle time is divided into 16 time slots, where  $m$  equals 4. It is important to note that the cycle time of a publisher must not exceed the global TSN cycle time, and the global TSN cycle time must be divided into power of two intervals for the publisher's cycle time. This division is given by the following formula:

$$cycletime_{publisher} = \frac{cycletime_{global}}{2^n}$$

Where  $n$  can take any non-negative integer value up to  $m$ . These restrictions make it easier to fully allocate the time slots. When new time slots are reserved, they are assigned to the first available slot within the time slot array. Referring to the example, Publisher 1 utilizes the initial time slot and fills in the required subsequent time slots. Following this, Publisher 2 is assigned the second time slot and fills in the necessary slots accordingly. In situations where overlapping reservations occur, an effort is made to redistribute the entire array. Due to the predefined division of the time slot array, modifications are not required for all subsequent cycles.



**Figure 7.8.** Example of a time slot array for one TSN End Device with two reservations. The global TSN cycle time is therefore divided into equal reservable time slots. Publisher 1 publishes two times, and Publisher 2 publishes four times per global TSN cycle. (See Eymüller et al. [52])

## 7.4 Related Work

Scheduling critical traffic in deterministic networks is a highly researched area. Not only in TSN, but also in other real-time network protocols, like Profinet IO IRT [68] or Flexray [209] the scheduling of real-time messages is a key issue. These approaches all rely on the existing information about what data should be exchanged during the initial configuration of the system. This means that the scheduling procedures can also be executed during the configuration phase. There are also scheduling mechanisms for static TSN networks that can be scheduled before deploying a real-time application. There are various approaches to this, such as solving constraint satisfaction problems with Satisfiability Modulo Theories (SMT) solvers, as shown, for example, by Craciunas et al. [41]. The paper addresses the computation of fully deterministic schedules for 802.1Qbv-compliant TSN multi-hop switched networks. Other approaches formulate the TSN streams as Integer Linear Programming (ILP) problem and use it to find an optimal schedule, see Dürr and Nayak [47]. Unfortunately, the computation times of such approaches are highly dependent on the number of streams and cannot be executed in constant time, so they are not suitable for dynamic networks with changing network nodes. Moreover, these approaches do not include already existing TSN streams and schedules. As a result, newly optimal solutions can trigger a complete change of the schedule. This means that existing streams cannot continue to communicate in the event of replanning. Gutiérrez et al. [64] and Pop et al. [161] have shown how dynamic TSN networks can be configured through a configuration agent architecture based on the IEEE 802.1Qcc [8] Stream Reservation Protocol (SRP) and OPC UA. Like the architecture shown in this work, Gutiérrez et al. and Pop et al. define a global managing instance called Configuration Agent (CA) that plans the communication paths and reserves the corresponding time slots. Communication requests can be made to the CA via OPC UA communication. The configuration of the network is done by existing communication protocols like NETCONF. A heuristic is used to add new streams to the current schedule. If the heuristic fails to add the new stream, a completely new schedule is created. These works have served as the inspiration for the dynamic configuration of the TSN communication channel. What is not integrated, however, is the management of the data topics (e.g., a data topic dictionary) and the interaction of the data topic dictionary with the time slot reservation. Recent publications also show that there is still a need for optimization in scheduling TSN time slots at runtime and adding streams at runtime. Gärtner et al. [65, 66] have presented an flexcurve concept to support incremental TSN reconfigurations at runtime. A flexcurve is a flexible curve that is limited by a number of addable streams along a communication path. Only when this curve is fully exhausted the time slots have to be rescheduled. Unfortunately, this approach is still limited by fixed cycle times of the streams. Especially for larger networks, it is conceivable to use optimized approaches.

Besides the management of time slots for a deterministic transmission, the management of data topics is an important aspect. In the OPC UA specification part 14 [136], already an OPC UA Pub-Sub Directory was introduced. The task of the directory should be the registration of publishers with their published datasets and the communication parameters, and it should be possible to query this information. However, the specification

states that the concept will be updated in later iterations. Therefore, an attempt was made to stick to the known specifications and to supplement the non-existent specifications accordingly. There are also approaches from other infrastructures for creating data topic directories in communication systems. In the Robot Operating System (ROS) [176] is an open-source framework used for building robotics systems. It comprises a range of software libraries and tools that simplify the creation of robot applications. ROS also uses a communication middleware to communicate between different components of a robotic system, so-called nodes. Nodes can be, for example, sensors, robots, or computation components. The ROS master is a component within the ROS framework that is responsible to facilitate the communication between different components. The ROS master also has a node registration that registers new nodes with their associated information, such as name and communication addresses. The message routing is also a part of the ROS master. When a node wants to communicate with another node, it first communicates with the ROS master, which manages the communication setup. Due to the similar structure, suitable concepts could be adopted.



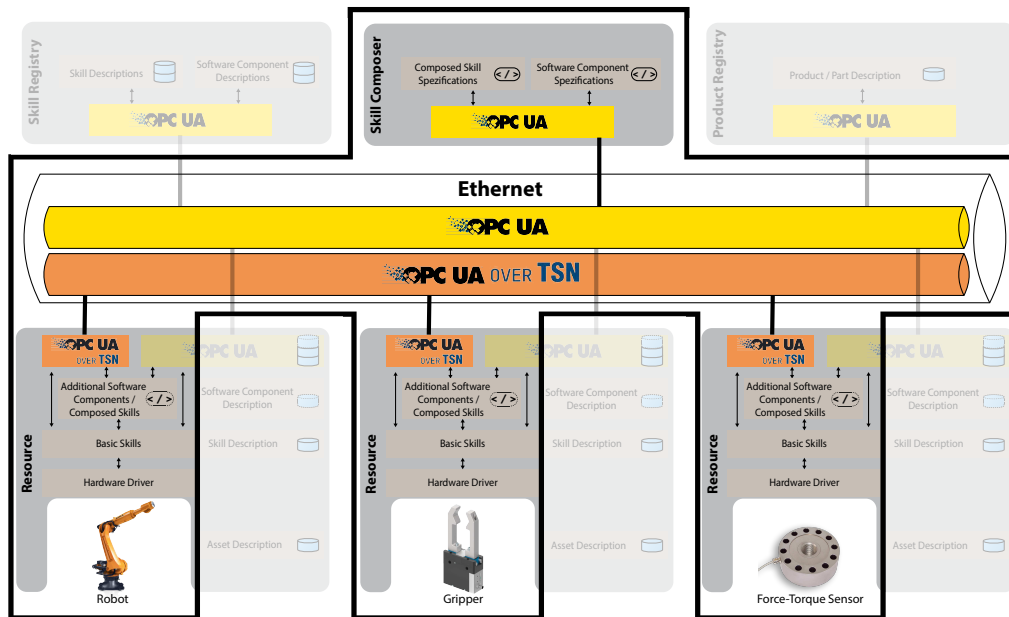
**Summary.** The system must be easily modified to respond efficiently to new production tasks. This chapter outlines a service architecture that enables smooth integration and modification of modular software components. Additionally, the crucial importance of software component reusability is discussed. The presented architecture facilitates local and distributed process executions on one or multiple computing nodes.

# 8

## Distributed Real-Time Execution of Component Skills in Distributed Control Networks

<b>8.1</b>	<b>Fundamentals</b>	<b>89</b>
8.1.1	OSGi: A Dynamic Module System	89
8.1.2	OPC UA Programs	90
<b>8.2</b>	<b>Services: Reusable Software Components</b>	<b>91</b>
<b>8.3</b>	<b>Real-Time Critical and Non-Real-Time Critical Execution</b>	<b>93</b>
<b>8.4</b>	<b>The Plug &amp; Produce Service Architecture</b>	<b>94</b>
8.4.1	A Uniform Data Representation: DataContainer	95
8.4.2	Modular Software Components: Real-Time Services	97
8.4.3	Distribution of Real-Time Services	99
8.4.4	Execution of Applications	102
8.4.5	Semantic Description of RTSs and RTS Networks	108
8.4.6	Synchronized Distributed Control Processes	109
<b>8.5</b>	<b>Related Work</b>	<b>115</b>

In the context of Plug & Produce, the goal is to have a flexible and adaptable system that can be specifically adapted to new tasks. The changeover from mass production to Lot-Size-1 production necessitates that the system has to adapt to each product and therefore the execution logic must be changed. Since such a system consists of numerous distributed components, which together have to perform a coordinated task, it is necessary to have a system that allows flexible distributed executions of skills in distributed systems. The shift from a highly optimized manufacturing system that produces identical parts in large quantities to a flexible and adaptable system capable of producing customized individual parts also shifts the design and layout of manufacturing systems from the design phase into the runtime of the system. It is not desirable that a small change in the process means a complete plant redesign, including conception and programming. Also, it is essential to make the execution as independent as possible



**Figure 8.1.** System architecture for a RealCaPP environment focusing on the distributed real-time execution of modular software components (cf. Figure 4.5)

from the resources used. This is achieved through the abstraction and generalization of skills. Therefore, it should be possible to format a task that runs with one corresponding resource configuration with a different resource configuration without having to change the software manually. Since the number of resources or calculation components in such flexible production facilities can also vary greatly, the software must be able to be distributed as required. Nevertheless, it must be ensured that this distributed execution also takes place under real-time conditions.

In the following chapter, Section 8.1 first discusses the fundamentals for the execution of modular software and how processes can be designed and executed in OPC UA. Section 8.2 explains what services and service-oriented architectures are and why they are well suited for the implementation of resource skills. Since not all executions require real-time, Section 8.3 will discuss the differences between non-real-time executions and real-time critical executions. Section 8.4 presents the services architecture, which is the basis for the distributed execution of modular software components such as skills. The chapter concludes with related work for real-time critical execution in local and distributed systems (see Section 8.5).

Figure 8.1 displays the complete RealCaPP architecture, as previously depicted in Figure 4.5, with emphasis on the specific sections pertinent to this chapter. Within this chapter, the attention is directed toward the execution of the skills and how Composed Skills can be executed across multiple resources by using the dynamic real-time communication middleware. It is therefore clarified how the Basic Skills of the resource can

be implemented, how Composed Skills can be developed on the basis of these skills and how these skills can exchange data via the real-time middleware.

## 8.1 Fundamentals

Before going into the distributed execution of processes in the RealCaPP environment, a few more important technologies are introduced. A technology is presented to implement modular software modules in the form of services. In addition, OPC UA Programs will be described, a way to integrate executable programs into OPC UA.

### 8.1.1 OSGi: A Dynamic Module System

The *Open Service Gateway Initiative (OSGi)* [122] is a framework for building modular and dynamic applications based on JAVA. In 1999, the OSGi Alliance was founded to create an open specification for delivering managed services to local networks and devices. OSGi provides a standardized approach to develop, manage, and deploy modular software components called *Bundles*. These OSGi bundles can be downloaded onto OSGi-compliant devices where they can be installed, started, paused, updated, and removed at runtime without requiring a restart of the entire system. In order to achieve this, the OSGi specification also manages the dependencies among bundles. By combining bundles, it is possible to develop services for devices with less memory that can be deployed on a large scale. [12]

The OSGi specification [146] defines a layered architecture for the realization of the functionality as shown in Figure 8.2. The *Service Layer* offers a flexible and standardized programming model for developers creating bundles. It simplifies the development and deployment process of service bundles by separating the service's specification, defined by a JAVA interface, from its actual implementations. This approach enables bundle developers to interact with services solely based on their interface specifications. Which implementation will be used later can be decided use-case dependent at runtime. The *Life Cycle Layer* provides an API for managing the life cycle of bundles. It defines

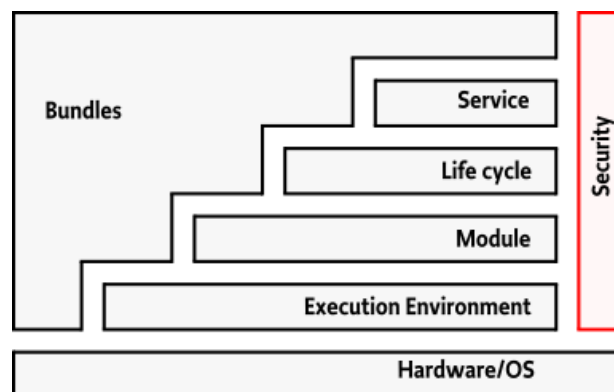


Figure 8.2. OSGi System Layers [146]

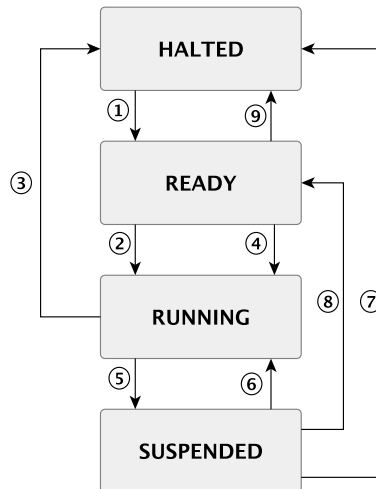
how bundles are installed, started, stopped, updated, and uninstalled. This layer also allows the control of the bundle's life cycle via events. The *Module Layer* is responsible for managing the modularization of software components. This layer manages the versioning, and dependencies of bundles. The module layer also is responsible for the loading of classes and the isolation of the bundles. The isolation of the bundles prevents conflicts between different versions of the same class or library. In order to be able to execute bundles independent from the hardware and operating system, there is the *Execution Environment Layer*. Its primary job is to provide the necessary resources and capabilities required for executing OSGi-based applications and bundles. Finally, there is the optional *Security Layer* that provides infrastructure for the delivery and management of applications that require applications in tightly regulated environments.

Unfortunately, JAVA is not a real-time capable programming language. By default, JAVA does not allow deterministic execution. Exemplary reasons for this are the class loading mechanism, the garbage collection, and the limited access of the application to the scheduling mechanisms of the operating system through the JAVA virtual machine. JAVA classes are initialized during runtime when an application first uses the class, causing undefined jitter at runtime. Also, the garbage collection can cause undefined delays because the algorithm pauses all application threads when executed to avoid interferences. Through special extensions, it is nevertheless possible to make JAVA real-time capable. The Real-Time Specification for JAVA (RTSJ) defines these extensions. [61] There are also approaches for the OSGi specification in combination with the RTSJ deploying dynamically reconfigurable real-time JAVA applications [180]. Since the other implementations are realized in C or C++, the dynamic services shall also be implemented in C++. Unfortunately, C++ has no direct support for OSGi. However, it is possible to apply similar concepts and principles in C++ to realize dynamic modular C++ applications. A possible C++ OSGi implementation is the CppMicroServices API [36].

### 8.1.2 OPC UA Programs

In order to realize stateful and long-running services in OPC UA, *OPC UA Programs* were defined. In the OPC UA specification part 10 [138], it is specified how programs can be described and exchanged in the OPC UA infrastructure. This specification includes a definition of *Program Nodes*, a representing OPC UA node that contains the program's structure, associated resources, and a current execution state. Each OPC UA Program has a finite state machine running on an OPC UA server representing the actual program. This state machine allows clients to initiate or monitor the execution process and receive results from the process in a standardized manner. Figure 8.3 shows the OPC UA Programs finite state machine. The specification defines four main states (HALTED, READY, RUNNING and SUSPENDED) that must be implemented by every OPC UA Program instance. It is also possible to extend the four states with additional substates for a higher level of detail. The transitions between the states are also defined by nine basic transitions that can either be triggered by a client or by the program itself. For triggering a state change, there are the five predefined program control methods: Start (Causes transition ②), Suspend (Causes transition ⑤), Resume (Causes transition ⑥), Halt (Causes transition ③, ⑦ or ⑨) and Reset (Causes transition ①). For the output





**Figure 8.3.** Standard OPC UA Program finite state machine with four default states and nine default transitions. (Adapted from [138])

of data generated by the program there is the possibility to output intermediate results or final results when the program comes to an end. In general, OPC UA Programs allow discovering and executing stateful and long-running programs generically. [138]

## 8.2 Services: Reusable Software Components

In addition to the goal of integrating and configuring resources at runtime, an important part of Plug & Produce is to integrate the newly added devices on the software side as well. Since the software must also be able to adapt quickly to changing parameters, changing processes, changing system setups, etc., it is also necessary to design the software integration to be as flexible and modular as possible. *Service-Oriented Architectures (SOAs)* are particularly well suited for this purpose. In the context of SOA, a service is a self-contained, loosely coupled unit of functionality with standardized interfaces that can be executed and composed to fulfill specific requirements. Services shall also have a well-defined service contract and the property that they are discoverable, reusable, interoperable, interchangeable, and durable. [121, pp. 39] All these characteristics are particularly well suited for the use of Plug & Produce systems.

The abstraction or generalization of skills is essential for the creation of services in the context of Plug & Play and especially for the reusability of these software components. It is, therefore, necessary to examine how skills can be reduced to specific details through abstraction and which similarities of skills can be identified through generalization. For example, considering the skill to grip a component. There are many classes of grippers that can implement the process of gripping. There are, for example, parallel grippers, suction grippers, pincer grippers, or magnetic grippers. There are different gripping principles like form closure, where the gripper conforms the contours of the object to

ensure a secure hold, or force closure, where the gripper applies a specific gripping force to hold the object. Also, the drive mechanism of the gripper can be different. There are electric, electromagnetic, pneumatic, or hydraulic grippers. The actuation also differs significantly between the different types of grippers. For example, some grippers can only open and close; there are position-controlled grippers and force-controlled grippers. Due to the large number of variations, it is therefore necessary to abstract the skill to grip and to specify the additional details only if necessary. Therefore, in many cases, it is sufficient to have a service that maps the gripping process, and depending on the type of gripper or gripping process, only the implementation of the service changes without changing the interface. This also makes it easy to combine these services, reuse existing processes that access the basic interfaces, and use them for different process combinations.

By implementing the skills as services, the following advantages are obtained for Plug & Produce systems: Through modularization and combination of services, reusable software components can be obtained. Looking again at the example of the pick and place skill, if the skill is implemented as a service, which in turn consists of the services grip, release, and move to position, it is possible to achieve that this skill is independent of the type of gripper and independent of the movement unit. This means that this service can be performed with a linear axis on which a parallel gripper is mounted, as well as on a 6-axis robot on which a vacuum gripper is mounted, without changing the service. The only thing that changes is what underlying services are called. By breaking down a complex system into small, self-contained units, it becomes easier to develop, test, and maintain such a system. This software architecture not only has the advantage that a system can be reused, it also ensures interchangeability. For example, it is possible to simply replace a parallel gripper with a vacuum gripper if the component to be moved allows it. Furthermore, easy adaptation to changing parts is possible. For example, if moving a bigger part can only be handled with a different gripper, it is not necessary to adapt the process implementation but only to use a different service of a different gripper. Since services can be nested arbitrarily, systems can also be scaled very easily, and the complexity can be expanded as desired without having to implement every Basic Skill. Another advantage of SOAs is the loose coupling of services. Because the services interact through well-defined interfaces and protocols, services can be deployed easily in distributed systems. This also allows the services to be easily redistributed in the distributed system. Services can be freely distributed, particularly in automation systems with components of very different computing power, which also increases scalability since load balancing can be operated. For example, it is possible to distribute the service pick and place anywhere in the system. For instance, everything can be done on one device or the services can be distributed across three devices, with one device performing the composed service pick and place, another device performing the service move to position, and again another performing the services grip and release. However, other distribution patterns are possible.

### 8.3 Real-Time Critical and Non-Real-Time Critical Execution

Especially when it comes to distributed systems and distributed execution, a basic distinction must be made between real-time critical and non-real-time critical executions. In particular, in distributed systems, the implementation and configuration of real-time critical applications also require non-real-time critical applications as a basic prerequisite in order to guarantee real-time execution later on. An important non-real-time critical task is the overall system coordination and configuration. Distributed systems usually consist of several independent components or subsystems that shall be able to collaborate with each other. Non-real-time critical tasks are required to initiate and coordinate communications, exchange large amounts of data and knowledge bases, and add and remove new components, new capabilities, or new tasks to the system. The initialization and parameterization of hardware and software components, for example, memory management and setting up hardware interfaces, are also important non-real-time tasks that are required to ensure real-time execution. In addition, system maintenance tasks such as logging information or errors and updating the software are not time-sensitive but make the system easier to operate. These tasks contribute to the overall reliability and coordination of the system and indirectly support the real-time critical execution, which depends on the correct functioning of the entire system.

Non-Real-Time Tasks

Apart from initialization, configuration, and parameterization, there are also skills of resources that are not deterministic or are only deterministic under fixed defined conditions. Mostly, these are skills that require several real-time cycles for execution, so-called *long-term skills*. For example, the skill of the robot move to a position can have different execution times depending on the start and end position, and it also depends on the parameterization of the skill, which maximum velocity or which maximum acceleration is allowed. Also, for some skills, no hard real-time is needed. For example, when grabbing while standing still, it is not relevant whether the gripper needs 1 ms or 100 ms for the execution. However, if several skills are combined into more complex skills, suddenly, the execution time or reaction time can play an important role. If, for example, a component is to be gripped or placed down during the movement, exact timing is required. Since it is not known at the design time how the skills will be combined later on, it is necessary that the individual skills are real-time capable. For executions that cannot be executed in one real-time cycle ( $\gg 1$  ms), it shall be possible to start the execution in one cycle, to terminate the execution in one cycle, and if necessary, to have the possibility to receive information on the execution in each cycle.

Long-term Skills

In addition to the long-term skills, there are also skills that can be executed in a fraction of a real-time cycle ( $\ll 1$  ms). These skills are called *real-time skills*. An example of real-time skills is the reading of sensor data, the calculation of positions, or the writing of control data to an actuator. Because these real-time skills require only fractions of a real-time cycle, combinations of several real-time skills can be executed in one real-time cycle. Of course, it is important to ensure that the worst-case execution times of the individual real-time skills are smaller in total than the time of one real-time cycle. The combination allows, for instance, a sensor-controlled movement of an actuator. Since,

Real-Time Skill

in one real-time cycle, the sensor values can be read, a control value can be generated based on the sensor data, and the output can be passed on to the actuator. Also, it is possible to form a long-term skill from a combination of numerous real-time skills. The sensor-guided movement can be considered as a long-term skill that runs over several real-time cycles, which is composed of the three real-time skills that are executed every real-time cycle.

Abstraction  
of Real-Time

Another essential consideration in real-time execution is the abstraction of real-time. Abstracting real-time executions involves creating a high-level representation or model of a real-time system that captures its essential characteristics without getting into the low-level details. The abstraction is solved by two techniques in this thesis: The first one is the usage of a layered architecture, where the lower layers handle the real-time requirements, and the higher-level layers focus on the system's functionality. Therefore, the system's primary user can utilize combinations of modules from the lower layers to implement his system functionality as desired without having to worry much about the real-time and associated limitations. The lower layers also take care of the execution that cannot occur in the real-time context, such as managing the memory, configuring hardware and communication interfaces, and much more. The second main technique for abstraction is a model-based approach for abstraction. For instance, function block diagrams such as those defined in IEC 61499 [214] are suitable for modeling distributed systems with modular applications. Each function block can be used to abstract the real-time execution, and several of these function blocks can be combined into more complex structures. The basic idea of these function blocks was used in this work, and further concepts were developed based on it.

## 8.4 The Plug & Produce Service Architecture

In order to get a flexible and adaptable Plug & Produce system, a software architecture is presented that can readily respond to changing use cases, changing hardware, and dynamic network topologies. As mentioned, service-oriented architectures are particularly well-suited as a basis for such customizable systems. The Plug & Produce service architecture must meet the following requirements:

- The software components (services) must be modular, capable of being added, and replaced dynamically at runtime. For this, it is important that the services have uniform interfaces. This makes it possible to add and exchange resources with the corresponding software at runtime. It is also possible to respond to changing tasks by adapting or replacing the services.
- All services must be able to be executed in real-time, and even combinations of several services must be real-time executable. All services have a defined lifecycle with different states, allowing non-real-time capable tasks (e.g., hardware initialization) to be decoupled from the real-time execution (e.g., sensor-guided motion).
- Real-time abstraction can be achieved by nesting real-time capable services in more complex structures that are also real-time capable. Only when creating new

services or modifying existing ones it is necessary to pay attention to real-time capability.

- Services can be executed on a single control device, for example, a single industrial computer, but they can also be distributed arbitrarily in the distributed system on a wide range of control devices. Here, too, the real-time must not suffer from the distribution. Therefore, the interfaces of the services can be easily expanded from local real-time interfaces to real-time-capable network interfaces.
- It shall be possible to execute both cyclic and acyclic executions in real-time in the service architecture. Process sensor values would be an example of cyclic execution since the sensor values are read and processed cyclically. An example of an acyclic execution would be the start of a movement or the opening of a gripper.
- Distributed processes synchronization shall be possible, meaning that distributed services can synchronize their states in real-time.

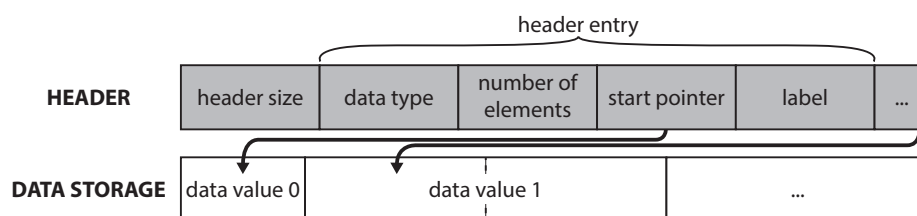
The following sections explain how these requirements can be implemented to realize a complete robot control system. It is explained how Basic Skills can be implemented and linked to form more complex structures.

#### 8.4.1 A Uniform Data Representation: DataContainer

An important aspect of a service architecture is a uniform data representation. Since the explicit data types that are later to be used for execution are not known during the design phase of the service architecture, and real-time programming languages such as C++ must know the data types at compile time, data representation is a challenge. To enable adding, exchanging, and removing modular software components at runtime, a data structure is needed that has constant access times and can represent a combination of primitive data types, as well as arrays of primitive data types.

The solution is the DataContainer data structure. Each DataContainer object consists of a header to locate the individual data objects, a storage area of dynamic length to store the data objects, and functions for accessing the data and taking care of the memory management in the system. Access to the data is only possible via the functions in order to keep the data consistent. Figure 8.4 shows the structure of a DataContainer. The header contains the number of data objects the DataContainer holds (header size) and

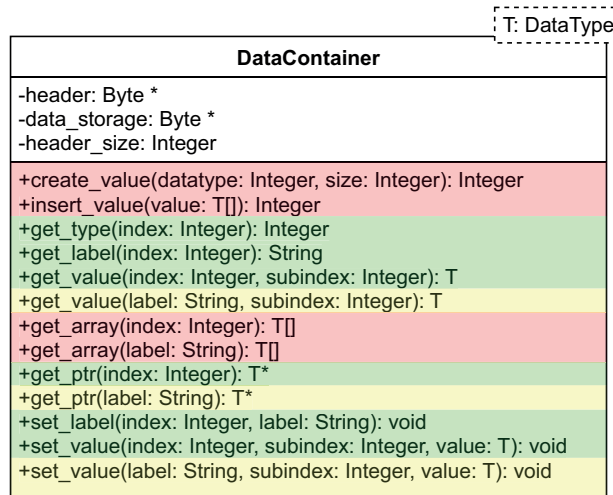
DataContainer



**Figure 8.4.** Structure of the DataContainer: Schematic structure of the header and the data storage using the example of a one-element data value and a two-element array.

entries for each data object. A data object entry has a data type, the number of elements, a pointer to the first memory location of the data object, and a label. Primitive data types have one element, and arrays have the array length as the number of elements. For example, the data value 0 represents a primitive data object with one element. Arrays of any length can be mapped, as in the example with data value 1, where the start pointer points to the first element of the array, and the number of elements defines the array size.

Figure 8.5 shows the class diagram of the DataContainer with the operations for interacting with the DataContainer. A distinction is made between non-real-time capable operations (highlighted in red), real-time capable operations with higher latencies (highlighted in yellow), and real-time capable operations with low latencies (highlighted in green). Due to the dynamic size of the containers, memory must be allocated for the header and the data objects. Since memory allocations are not deterministic, care must be taken when using DataContainers to ensure that they are created, extended, or copied (non-real-time capable operations) in the initialization phase and not in the real-time critical phase of the system. If this restriction is observed, the system remains deterministic in its execution. Care can also be taken to avoid real-time operations with higher latencies. For example, identifications via indexes are faster than having to perform a text comparison for every query, as it is the case with data access via labels. All these operations ensure type safety. If possible, compatible types are cast during execution. Since the types are not instantiated until runtime, incorrect types are reported as runtime errors.



**Figure 8.5.** Class diagram of the class DataContainer. Red operations are non-real-time capable. Yellow operations are real-time executable but need more execution time. Green operations are real-time capable and have low latency.

### 8.4.2 Modular Software Components: Real-Time Services

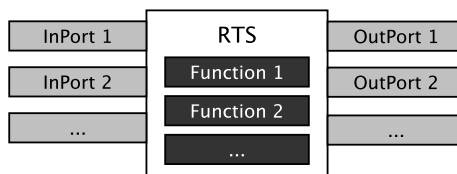
Each Plug & Produce real-time system consists of several modular software components that, in combination, take over the control of a robotic system. These modular software components must be designed so that they can be added, changed, or removed at runtime. These modular software components are called *Real-Time Services (RTSs)*.

An RTS is a modular software component that has data inputs (InPorts), processes the provided data, and then writes the results to data outputs (OutPorts). The processing can be represented as a function  $rts$  that maps the inputs ( $IP^n$ ) to the outputs ( $OP^m$ ), see Equation 8.1.

$$rts : IP^n \rightarrow OP^m, (ip_1, \dots, ip_n) \mapsto (op_1, \dots, op_m) \text{ with } n, m \in \mathbb{N}_0$$

**Equation 8.1.**  $rts$  function of an RTS

By this definition, it is possible that RTSs consist of both In- and OutPorts, only InPorts, only OutPorts, or to have no ports at all. The ports are especially suitable for mapping cyclic data connections between the modular software components. Since there are also components in automation systems that work event-based, an additional approach for acyclic calls was developed. Each RTS can provide a set of functions which are available in the complete RTS Network. The functions can either adapt the  $rts$  function of the RTS or can be executed independently. Adjustments to the  $rts$  function will be taken into account in the next execution cycle of the RTS Network in order to avoid inconsistencies. Figure 8.6 shows a single RTS with InPorts on the left and OutPorts on the right. Functions are displayed inside the RTS.



**Figure 8.6.** A single Real-Time Service (RTS) with InPorts, OutPorts and Functions

To implement a control system for robots, several RTSs are combined into an *Real-Time Service Network (RTS Network)*. Therefore, the OutPorts of an RTS are connected to the InPorts of other RTSs. An OutPort can be connected to several InPorts, whereas an InPort can only be assigned to one OutPort. As a further restriction, no cyclic dependencies are allowed due to circularity in the network.

RTS Networks can be encapsulated, which means that RTS Networks can be seen as RTS again. The RTS Network representing RTS has the set of all not yet connected InPorts of

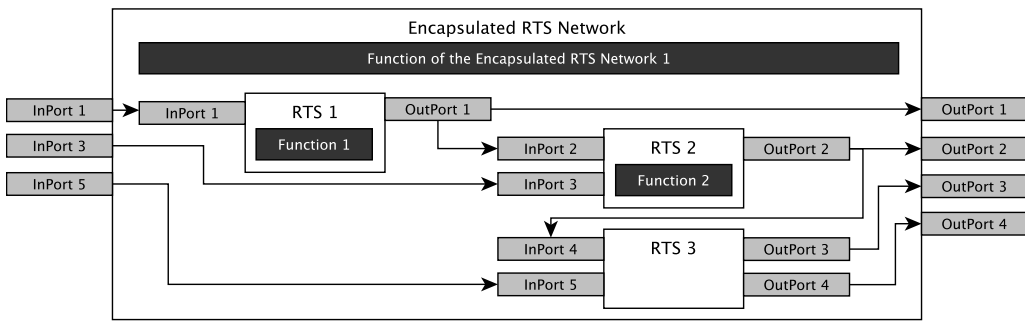
Real-Time  
Services (RTSs)

Functions

RTS Networks

Encapsulated  
RTS Networks

all RTS in the RTS Network as InPorts. Since OutPorts can be connected multiple times, all OutPorts of all RTS in the RTS Network are used as OutPorts of the encapsulated RTS. The encapsulation makes it easier to reuse complex RTS Networks for other purposes. All internal functions of the sub-RTSs are passed on to the encapsulated RTS Network. It is also possible to add functions to an encapsulated RTS Network. These added functions can use the internal functions of the sub-RTSs. Figure 8.7 shows an encapsulated RTS Network with tree sub-RTS. All four OutPorts are passed on to the surrounding structure. Since the InPorts 2 and 4 are used internally, only the InPorts 1, 3, and 5 are passed on to the surrounding structure. Both the internal functions (Function 1 and Function 2) and the functions of the encapsulated RTS Network (Function of the Encapsulated RTS Network 1) are visible to the outside.



**Figure 8.7.** Encapsulated RTS Network with an RTS Network consisting of three RTSs inside

Execution of  
RTS Networks

After describing the structure of the RTSs and the RTS Networks, the execution and execution times are considered. RTS Networks are executed cyclically with a specified cycle time. The entire RTS Network is executed in each cycle. In one cycle, all RTSs in the RTS Network have to read the InPorts, process the *rts* function, and write the results of the *rts* function to the OutPorts. Due to dependencies of the individual RTSs by the connections of the ports, the execution of the RTSs can not be parallelized in general. The RTSs are executed in paralyzed sequences that take the dependencies of the ports into account.

To guarantee real-time in the system, the Worst-Case Execution Time (WCET) must be analyzed. In a worst-case scenario, no RTSs in the RTS Network can be executed in parallel, making it a pure sequential execution. The WCET of an RTS Network is calculated by adding the WCETs of each RTS plus the worst-case transmission time for the exchange of data over the data ports between two RTSs, see Equation 8.2. However, this is an overestimation since most RTS Networks are at least partially parallelizable. Nevertheless, when implementing a system, care must be taken to ensure that the WCET of the RTS Network is less than the cycle time of the RTS Network.



$$WCET_{RTS\ Network} = \sum^{RTSs} (WCET_{RTS} + WCET_{Transmission})$$

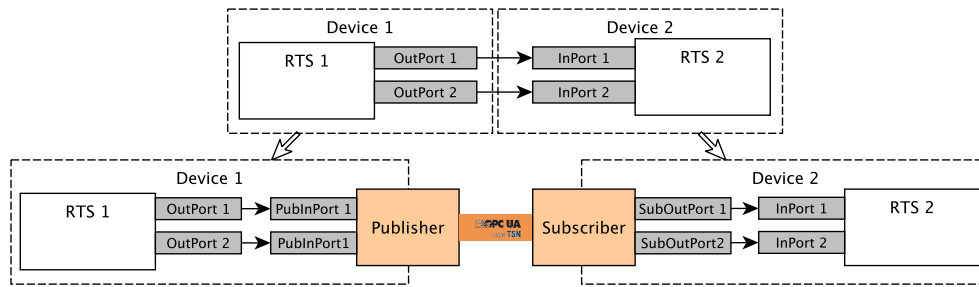
Equation 8.2. WCET of an RTS Network

### 8.4.3 Distribution of Real-Time Services

If RTSs are executed on one device, both the cyclic and the acyclic connection are easy to implement since all RTS have a common memory area. Due to the shared memory area, the latencies for data exchange via ports, and function calls are almost negligible ( $\ll 1$  microsecond). Latency increases quickly when considering distributed systems where services are distributed across multiple devices ( $< 1$  millisecond). In distributed systems, no shared memory exists, and the data between RTSs must be exchanged via the network. Many of the network protocols are not real-time capable, especially with respect to determinism and hard real-time bounds. The distributed data transmission is carried out via OPC UA over TSN to ensure data transmission in real-time. OPC UA over TSN establishes a cyclic communication with a fixed time interval. The WCET of the transmission via OPC UA over TSN is the defined cycle time of the transmission plus the time for the actual network transmission. Due to the reservation of time slots, which TSN requires, there can be no delays caused by other network traffic.

In the further course, a distinction is made between *internal* and *external connections*. An *internal connection* is a connection between RTSs that run on the same device and can use the shared memory. An *external connection* is a connection between RTSs that runs on multiple devices and has to communicate via the network. To make the interface between the ports usable for internal and external connections, the data ports are implemented by DataContainers. For an internal connection, the OutPort contains a DataContainer object. If a connection from an InPort to this OutPort is established, a pointer is created in the InPort, which points to the DataContainer object of the OutPort. Thus, the transmission time for an internal connection is the time of a pointer access. With this realization, connecting several InPorts with one OutPort is possible. For the external connections, OPC UA over TSN is used for the network transport. The facade pattern was used to hide the complexity of the configuration of a real-time connection via OPC UA over TSN. The facade uses the DataContainers of the OutPorts that need to be transferred to another device. Due to the stored data in the header of the DataContainer, most of the required data for configuration of the transmission are already available. What is also required is the cycle time of the transmission and a free time slot for the TSN transmission. The facade also covers the missing data. By default, a fixed cycle time is used for all transmissions. The approach presented in Chapter 7 is used to establish the OPC UA over TSN connection with the reservation of the time slots. An OPC UA publisher is created for each OutPort to be transmitted. For this purpose, the facade writes the data of the DataContainer into the OPC UA information model and keeps the data synchronized. Then a publisher is created for the data in the information model. Any device that now has an InPort that wants to connect to the

Internal and  
External  
Connections



**Figure 8.8.** Realization of external connections. One RTS Network is distributed to two devices.

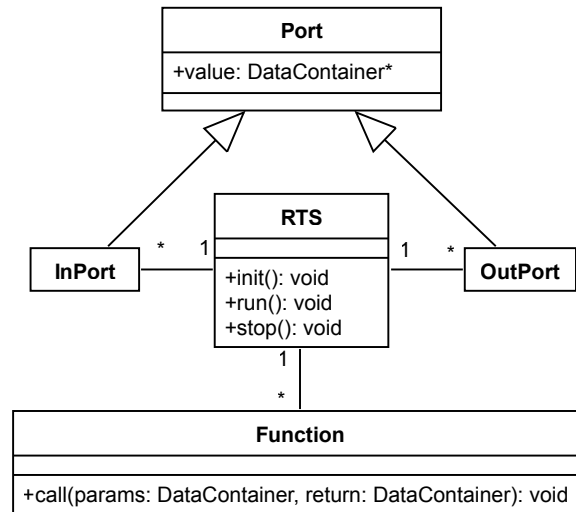
provided OutPort creates an OPC UA subscriber. The subscriber receives and writes the data into the local OPC UA information model. From there, a DataContainer object is created, which can be accessed by the InPorts. Since several subscribers can subscribe to one publisher, mapping one OutPort to several InPorts with external connections is possible. Figure 8.8 shows how a simple RTS Network can be distributed to two devices.

#### Distributed Function Calls

In order to also create a uniform definition for functions for distributed execution, functions are also implemented with DataContainers. A function has two DataContainers, one for the parameters of the function and one for the return values. Also methods can be implemented in the same way as functions. Only the return value DataContainer is empty. So every function has a call function with the following definition:

```
call(params:DataContainer, returns:DataContainer):void
```

If a function is called on the same device, it is a local function call of the call function. However, if a function is executed on another device, the function call and the transfer of the parameters and results must be done via the network, e.g., via OPC UA over TSN. Because OPC UA over TSN does not support acyclic communication in real-time, the acyclic communication is implemented by cyclic communication. Therefore, likewise, the facade is used. The facade synchronizes the DataContainer for the function parameters of the function caller to the OPC UA Information model. Additionally, a flag for the actual function call is added to the information model. A publisher is created for this information. At the function provider, a subscriber is created that writes the parameters and the call flag to the information model. A DataContainer object is created out of the parameters in the information model. If the call flag is active, the function is called with the parameter DataContainer. In the function provider, a publisher is created, which contains the result DataContainer and a flag, which indicates if the function was executed successfully. The function caller in turn creates a subscriber that writes the function results to a DataContainer object. Using the two flags, it is possible to call functions and then receive information about successful execution. The worst-case execution time can be determined as follows: A maximum of one TSN cycle plus the time for the actual network transmission is required to call the function via OPC UA



**Figure 8.9.** Class diagram of Real-Time Services (RTSs)

over TSN. The function is then executed, which corresponds to the local execution time of the function. Afterward, the results are transferred back, which requires a maximum of one TSN cycle plus the time for the actual network transmission. Thus, functions with a deterministic execution time can also be called in a distributed manner with a defined maximum execution time.

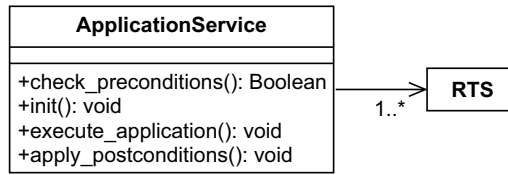
All RTSs are implemented as services implemented according to the OSGi specification [201]. Due to the real-time requirements, the services are not implemented with JAVA but with a C++ implementation. The CppMicroServices API [36] provides an OSGi-like C++ dynamic module system to create services. Although the OSGi specifications were initially designed for the JAVA virtual machine, the majority of its core principles are independent of a specific programming language and can be effectively applied in a C++ setting. The API includes functions for handling services as well as a services registry for locating and querying services. The OSGi specification also enables the deployment of dynamic services that can be installed during runtime on a deployed system. Within the context of OSGi, all RTSs are bundles that implement the uniform interface **RTS**, which is illustrated in Figure 8.9. Each RTS follows a three-step life cycle consisting of initialization, execution, and deinitialization phases. The initialization phase is triggered by the `init()` function and primarily handles tasks such as creating ports, functions, and allocating memory for the RTS, which is not necessarily deterministic. During initialization also all `DataContainers` are initialized to avoid memory allocations during runtime. The same applies to non-deterministic initializations of resources, for example, a handshake mechanism for starting the communication with a gripper or robot. These initialization sequences can take place without impacting the real-time execution. During the execution phase of the RTS, which is real-time critical, in each execution step triggered by the `run()` function, a precise sequence of actions is performed. These actions involve reading the `InPorts`, executing the intrinsic logic

of the software component (execution of the *rts* function), and writing the results to the OutPorts. The deinitialization phase, triggered by the `stop()` function, is used to remove the RTS from the system when it is no longer required. The deinitialization is also not deterministic. In the deinitialization phase, the entire memory management, e.g., the release of memory areas, is handled.

Due to the C++ implementation, there is a restriction concerning the bundle definition: Bundles must implement an interface which must be known at compile time of the system. C++ is a statically typed language, meaning that values have to be attached to types at compile time. There are also no other build-in concepts in the C++ standard, such as reflection in JAVA [123]. Reflection allows programs to modify their structure, including classes, attributes, and function at runtime. Even the newly added C++ technical specification ISO/IEC TS 23619 [86] only enhances the capabilities of C++ by introducing features that allow for the examination of program elements like variables, enumerations, classes, and their members. Still, no modification at runtime is possible. This means that all data types must be known at compile time, which makes it challenging to integrate new services that shall be added during runtime but are not known at compile time. This problem is solved by using `DataContainers` as compile time known object for the data of ports and functions. The `DataContainer` data structure provides a uniform way of representing data within the RTSs by being used for both the InPorts and OutPorts, as well as for defining the functions called on the RTSs. This allows the flexibility of not needing to know the data types during the design phase and enables the ability to load them dynamically at runtime. To elaborate, each OutPort includes a `DataContainer` object that represents the data value of the port. Conversely, an InPort only contains a pointer that directs to a `DataContainer` object of an OutPort. This approach eliminates the need to copy data objects when connecting two ports, saving time and reducing execution time. When it comes to functions callable on an RTS, they are designed with a call function that contains a `DataContainer` object for both the function parameters and return values. By hiding the parameter types and return types through the `DataContainers`, all functions have a consistent representation already determined during the compilation stage.

#### 8.4.4 Execution of Applications

In order to be able to switch between several applications and create complex sequences of skills, a further concept for the execution of RTSs and RTS Network is introduced, so-called *Application Services (AS)*. ASs are independent services that can be executed arbitrarily. An example for an AS would be the execution of the task move the cube by a robot from position  $(X_S, Y_S, Z_S)$  to position  $(X_G, Y_G, Z_G)$ . AS are also modular services that can be loaded at runtime, and additionally have preconditions and postconditions for the execution. To successfully execute an AS, first, all its preconditions must be fulfilled. After the successful execution of the application, there is a postcondition which is passed on to the system. If all preconditions are fulfilled, it is possible to execute the application. If the application is executed successfully, a postcondition is returned to the system. These conditions contain adjustments to the system made by the execution



**Figure 8.10.** Class diagram of the class `ApplicationService`

of the application. In the example shown, this means that for the movement of the cube, the following precondition must apply:

- The cube must lie on the position  $(X_S, Y_S, Z_S)$ .
- There must be resource combination with a robot that can move a cube (e.g., a robot with a gripper with matching gripper jaws).
- The position  $(X_S, Y_S, Z_S)$  and  $(X_G, Y_G, Z_G)$  must be reachable by the robot
- The gripper is open and contains no elements, and
- The target position  $(X_G, Y_G, Z_G)$  must be free.

If all these conditions are met, the cube can be moved. After execution, the state of the system is adjusted by the postcondition. Therefore, the following postconditions apply:

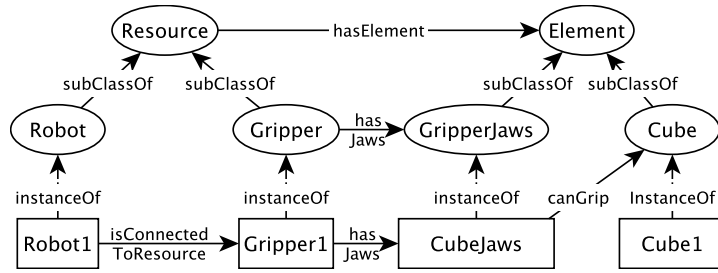
- The position  $(X_S, Y_S, Z_S)$  is free
- The cube lies on the position  $(X_G, Y_G, Z_G)$  and
- The gripper does not contain any elements.

ASs therefore have an influence on the system and its environment through the execution. After the postconditions are applied, other AS can be executed, which, on the contrary, had as preconditions that are now fulfilled by the new system state.

Figure 8.10 shows the class diagram of the AS. Each AS has four basic functions. With the function `check_preconditions()` the existing preconditions can be checked, if all preconditions are fulfilled true is returned. If the preconditions are fulfilled, the `init()` function can be called, this is intended for the initialization of all RTSs involved since the initialization cannot be executed in real-time. The AS coordinates the initialization of all RTSs. Only after all RTSs are initialized the respective application can be executed via the `execute_application()` function. After the successful execution of the application, the postconditions can be applied, and adjust the new system state. This is done by the function `apply_postconditions()`.

In order to have a uniform format to formulate preconditions and postconditions, the conditions are implemented on the knowledge base of the ontology. The checking of preconditions can be solved with the help of ASK SPARQL queries. These queries return a boolean statement as a result. The knowledge base of the ontology contains information about the individual resources, how the resources are connected to each other, and what skills the resources have. In addition, information about other elements, such as components or products, can be described. To illustrate such knowledge ontology,

Pre- and Post-  
Conditions of ASs



**Figure 8.11.** Cube example ontology with a resource configuration that can grip cubes. Classes are represented as ellipses: Instances are represented as rectangles: Relations between classes are shown as arrows:  $\rightarrow$  Instantgements are represented as dashed arrows:  $--\rightarrow$

```

1 ASK{
2 ...
3 ?robot isA Robot.
4 ?gripper isA Gripper.
5 ?robot isConnectedToResource ?gripper
6 ?gripper hasJaws ?jaws.
7 ?jaws canGrip Cube.
8 ...
9 }

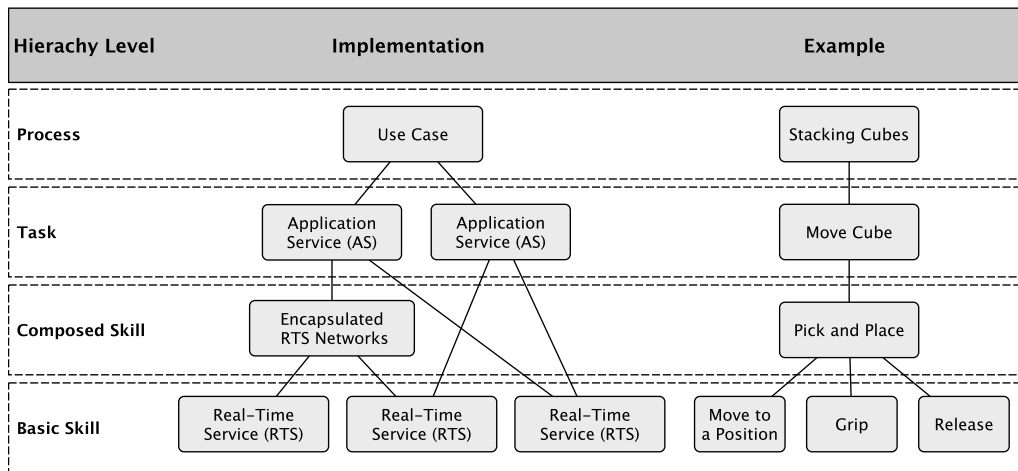
```

} There must be resource combination with a robot that can move a cube (e.g., a robot with a gripper, with matching gripper jaws)

**Listing 8.1.** ASK SPARQL query for the second precondition of the cube example

an ontology is made for the cube example. Figure 8.11 shows a part of an ontology describing a resource configuration and the description of a cube. The system contains a robot with a gripper with gripper jaws that can grip cubes. If the precondition 2 ("There must be resource combination with a robot that can move a cube (e.g., a robot with a gripper, with matching gripper jaws)") from the cube example is to be checked, the query shown in Listing 8.1 can be defined. The query checks if there is a robot connected to a gripper with gripper jaws that can hold a cube. For the given ontology, the query would return True. Postconditions are defined as change commands of the ontology. This means that new edges can be added or removed, or properties can be added, deleted, or changed. In the example of the cubes, for example, the position property of the cubes would change after move the cube task is executed.

The ASs can be used to implement complex processes that implement specific use cases. Therefore, the implementation of the process is called *Use Case*. Use cases can implement processes in two ways: The first way is that the user specifies the execution order of the ASs to form a process. The second way is that the user specifies which ASs belong to a process, and the system orchestrates itself based on the preconditions and postconditions of the ASs.



**Figure 8.12.** Overview of all hierarchy levels for the execution of processes in the RealCaPP architecture

Figure 8.12 shows the complete hierarchy for executing processes in the RealCaPP architecture. The figure shows the decomposition of a process into its individual execution components. An entire process is represented by a Use Case. A Process can be composed of several Tasks. For example, if the goal of the overlying process is to stack a tower of cubes. This process can be divided into different tasks, which are implemented by Application Services. Tasks can be, for instance, to get a cube out of a magazine or to move a cube accordingly. Tasks in turn consist of skill calls, where skills can be either Basic Skills or Composed Skills consisting of several Basic Skills or other Composed Skills. Basic Skills are implemented as RTSs and Composed Skills as Encapsulated RTSs. For example, the task of moving a cube can be implemented by the Composed Skill pick and place, which in turn is built from the Basic Skills move to a position, grip and release.

In the context of SOA, where applications are composed of multiple loosely coupled services, orchestration plays a crucial role in defining the workflow or sequence of steps that need to be executed to accomplish a particular task or process. Due to the self-orchestration of the system, the user has the possibility to adapt the system or add process steps without having to worry about the exact order of execution. If there is a possible execution order, it can be found automatically. For the first version of the orchestration of AS, a simple customized depth-first search algorithm [110, p. 36] was used. This can be replaced subsequently also by more complex orchestration or planning algorithms. For example, the concepts of Nägele [130] can be used.

First of all, it must be defined which AS are to be used, and there must be a state description of the system in the form of an ontology, as presented in Chapter 5 and Chapter 6. Initially, a snapshot of the ontology is created. This allows to restore this snapshot of the ontology later on. Afterward, all preconditions of the ASs are checked one after the other. This is done by checking the ASK SPARQL query of the ASs. If all preconditions of an AS are fulfilled, the postconditions are applied to the ontology and

Application  
Orchestration

a new snapshot of the ontology is created. Afterward, the whole routine is continued on the still existing ASs. In case no preconditions can be fulfilled, the last applied postcondition is discarded by resetting the ontology to the last snapshot, and another possible AS is chosen. This is executed until either a valid solution with all ASs is found or it is determined that no solution is possible. Finally, the execution order is saved so that it can be reused for later executions without having to perform a new orchestration. The pseudo-code for the described execution using a depth-first search to find an appropriate execution order is given in Algorithm 8.1.



**Algorithmus 8.1** : Depth-First Search for Finding Execution Order**Data** : Set *ASs* of Application Services**Result** : Possible execution order for the services**Global** : Global ontology *onto*

```

1 Function FindExecutionOrder(ASs):
2   visited ← an array of size |ASs| initialized with False;
3   stack ← an empty stack;
4   for as in ASs do
5     if not visited[as] and as.check_preconditions() then
6       | DFS(ASs, as, visited, stack) ; // Start DFS planning
7     end
8   end
9   // Check if no solution found
10  if |stack| <> |ASs| then
11    | Print "No Solution Found!";
12    return null;
13  end
14  return stack ; // Return execution order
Data : Set ASs of ASs, current AS as, array of visited AS visited, stack for
        solution stack
14 Function DFS(ASs, as, visited, stack):
15  visited[as] ← True;
16  snapshot ← onto.create_snapshot();
17  as.apply_postconditions();
18  found ← False;
19  for as1 in ASs do
20    if not visited[as1] and as1.check_preconditions() then
21      | DFS(ASs, as1, visited, stack); // Recursive call of the DFS
22      | found ← True;
23    end
24  end
25  if not found then
26    if all elements of visited = True then
27      | stack.push(as); // Solution found with all ASs
28    else
29      | // Wrong path -> Resetting system state
30      | visited[as] ← False;
31      | onto.revert_snapshot(snapshot);
32    end
33  else
34    | stack.push(as); // Add AS to solution
35  end

```

### 8.4.5 Semantic Description of RTSs and RTS Networks

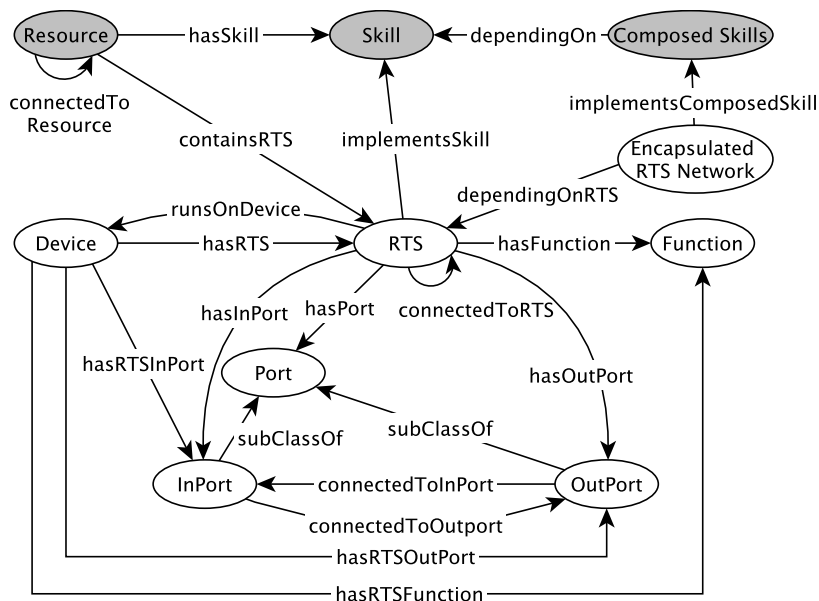
In addition to the implementation of the service architecture, a description of the services and the interrelationships is essential. Also, the description of the RTSs and the RTS Networks is added to the knowledge base of the ontology. The description contains the basic description of the RTS, e.g., information about the ports and the functions. The ontology also allows to identify, for example, which ports are connected to each other and which RTS is running on which computation device. Figure 8.13 shows the ontology for describing the RTS system. Besides the description of the RTS itself, the ontology also shows the relationship between skills and the RTSs. The relation `implementsSkill` describes which skill can be mapped by which RTS. RTS can also have a connection to resources, so it is possible to find out which RTS belongs to a resource by the relation `containsRTS`.

Service Registry  
for RTSs

Due to the detailed description of the RTS, the ontology can also be ideally used as a Service Registry for RTSs, since all relevant information can be stored and queried here. Therefore, it is possible to query via SPARQL queries which RTSs exist, which functions are available, and where the RTSs and functions can be found in the distributed system.

Determination  
of the RTS  
Execution Order

Moreover, graph algorithms can also be easily applied to the graph structure of the ontology. Since the connection of the ports creates a dependency between the RTSs it is essential in which order the *rts* functions of the RTSs are executed. For the



**Figure 8.13.** Excerpt from the ontology describing RTSs and RTS Networks. Classes are represented as ellipses: Relations between classes are shown as arrows:  $\rightarrow$  Instantancements are represented as dashed arrows:  $--\rightarrow$  Already mentioned concepts are grayed out.

---

```

1 SELECT ?rts2 (COUNT(?mid) as ?level) → Counting all dependencies to other RTSs
2 WHERE {
3   {
4     ?rts1 isA RTS.
5     ?rts1 connectedToRTS* ?mid.
6     ?mid connectedToRTS+ ?rts2.
7   }
8 UNION
9   {
10    ?rts2 isA RTS.
11    ?mid isA RTS.
12    FILTER(?rts2 = ?mid)
13  }
14 }
15 GROUP BY ?rts2
16 ORDER BY ?level

```

---

**Listing 8.2.** SPARQL query for determining the execution order. The RTSs are output in an ordered sequence.

**hasOutPort** ◦ **connectedToInPort** ◦ **inverse(hasInPort)** → **connectedToRTS**

**Equation 8.3.** Property chain for the inference of the relation **connectedToRTS**.

determination of the execution order, dependency graphs are created, and the execution order is derived from these graphs. Listing 8.2 shows the SPARQL query, which allows to retrieve an execution order from the description of the RTSs and the RTS Networks. The query consists of two parts. First, all dependencies between the RTS are searched for by chain query (see ①). This is done via the **connectedToRTS** relation. This relation can be inferred by the property chain shown in Equation 8.3. The **connectedToRTS** relation indicates which RTS depends on which RTS. Since it can happen that some RTSs do not have any dependencies, all RTSs are added once in the second part of the query (see ②). These values are now grouped according to the RTS, and the length of the dependency graphs is counted. These calculated values are afterward sorted by the `level` in ascending order, which results in the execution order of the RTSs.

#### 8.4.6 Synchronized Distributed Control Processes

Especially when a distributed system is used, there are a lot of executions in an automation system that have to run distributed on several components but have to be synchronized in some way. Particularly in the case of sensor-guided movements in robotics, for example, it is necessary that the sensor system and the robot start operating synchronously and also exchange data reliably with each other afterward. If a central

system is used, for example, a control system that is connected to both the sensor and the robot, it is easy to specify a sequence through the program. In addition, the system has a common understanding of time and also a common memory. If this structure is broken down, it must be ensured that there is a common definition of time and how sequences can be mapped in distributed networks. As already shown in previous work (see Eymüller et al. [51]), in the context of distributed real-time critical control systems, four main requirements must be fulfilled:

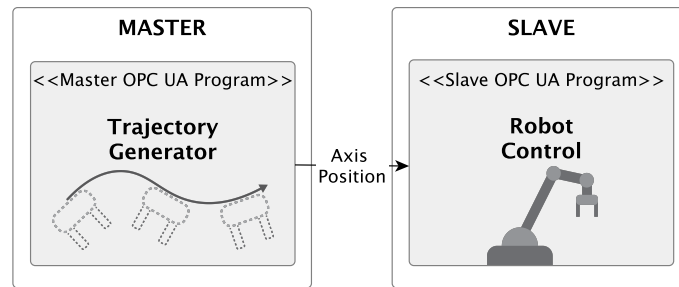
- Synchronization of the clocks of several distributed components
- Allowing real-time communication between these components
- Synchronization of distributed control processes across multiple components without losing real-time
- Enable exchange of information between the distributed control processes in real-time

A combination of OPC UA Programs and OPC UA Pub-Sub over TSN communication is used to meet the mentioned requirements. By leveraging OPC UA Pub-Sub over TSN, the solution enables clock synchronization among multiple components and facilitates real-time communication between these nodes as shown in detail in Chapter 7. Additionally, the inclusion of OPC UA Programs allows for the execution of continuous processes. The challenge is how to achieve distributed, continuously running processes that can synchronize and exchange real-time data and process information in real-time. By addressing this challenge, the proposed approach enables the implementation of distributed synchronized industrial control software.

As shown in the previous section. It is possible to distribute RTS Networks across multiple devices by using external connections. What is missing is a way to synchronize the distributed running processes. For example, so that the split RTS Network can be started on both execution nodes simultaneously.

First, a way must be found to synchronize the states of distributed OPC UA Programs. Therefore, the standard state machine of the OPC UA Program specification was revised by adding a handshake mechanism to synchronize multiple distributed OPC UA Programs with each other. In this context, synchronization refers to the process of exchanging and aligning the states of all distributed OPC UA Programs in order to achieve a unified state. It is distinguished between two types of distributed OPC UA Programs: the Master OPC UA Program and the Slave OPC UA Programs. The *Master OPC UA Program's* main task is to check whether all Slave OPC UA Programs are synchronized and ready to start the actual program. Therefore, it is crucial for the distributed OPC UA Programs to be able to exchange their status with each other. The *Slave OPC UA Programs* receive instructions from the Master and respond with state changes, which are then communicated back to the Master.

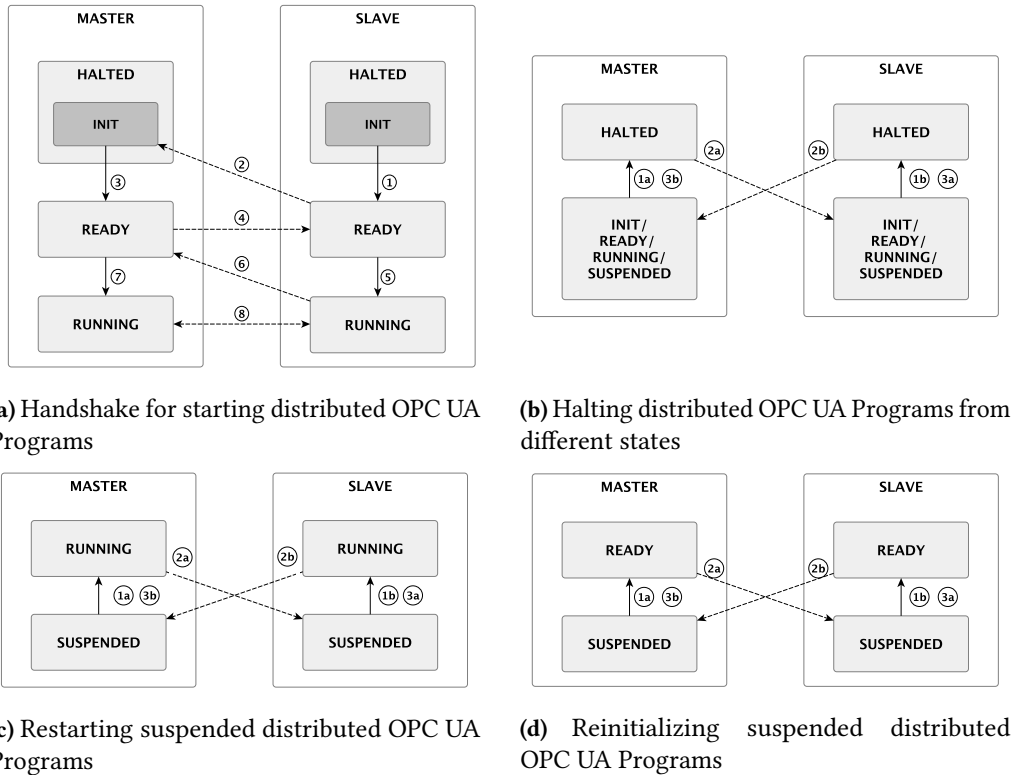
For illustration purposes, this is explained using an example in the robotics domain with one Master and one Slave depicted in Figure 8.14. Suppose a user wants to generate a trajectory on one component (Trajectory Generator) and execute this trajectory on another component connected to a robot (Robot Control). The role of the trajectory generator is to create a trajectory and supply the robot control with target positions for



**Figure 8.14.** Application example for distributed OPC UA Programs with one Master and one Slave: Distributed execution of a trajectory on a robot arm (See Eymüller et al. [51])

the robot axes at consistent time intervals. The robot control requires desired positions for each axis within a specific time frame. It converts these desired axis positions into control instructions, which are subsequently transmitted to the robot. For this purpose, it is also important that both executions on the distributed components start simultaneously, can be paused if necessary, and can be resumed. Moreover, the data flow must take place in real-time while the application is running. In order to synchronize the distributed state machines, it was necessary to add states. However, due to the strict specification of OPC UA Programs, no additional states should be added to the state machine. Instead a sub-state was appended to the existing state machine. The sub-state INIT was added to the HALTED state to ensure that all involved distributed OPC UA Programs are properly initialized and ready for the upcoming state changes before they change into the state READY.

Figure 8.15 shows the extended distributed state machine for distributed OPC UA Programs. State transitions are represented by solid arrows ( $\rightarrow$ ) and notification over the OPC UA Pub-Sub over TSN communication channel by dashed arrows ( $- - \rightarrow$ ). In the following, the Master OPC UA Program is called Master, and the Slave OPC UA Programs are called Slave. First, it is shown how distributed OPC UA Programs can be synchronized by a handshake mechanism and how they can be started after successful synchronization, illustrated by Figure 8.15a. Initially, it is necessary for the Master to determine the number of participating Slaves. The synchronization process starts with the initialization of the Slave. The Master waits in the INIT state till all Slaves are initialized and switched into the state READY. With the state switch from HALTED to READY, each Slave notifies the Master. After the Master has a notification of all participating Slaves the Master also switches to the READY state and notifies all Slaves that the Master is ready for the execution of the distributed program. On receipt of the notification from the Master, the Slaves switch to the RUNNING state and notify the Master afterward. After receiving all state change notifications, the Master also changes its state again. When all participating distributed OPC UA Programs are finally in the state running, process data can be exchanged between the distributed OPC UA Programs in real-time using the



**Figure 8.15.** Distributed OPC UA Programs finite state machine. Changes of state ( $\longrightarrow$ ) are communicated via the OPC UA Pub-Sub communication channel ( $\dashrightarrow$ ) between Master and Slaves. ① to ⑦ specify the order of the transitions and notifications. ⑧ represents the exchange of process data between the Master and Slaves. (See Eymüller et al. [51])

OPC UA Pub-Sub over TSN communication channel. For the example shown, it means that the axis positions are transmitted between the trajectory generator (Master) and the robot control (Slave) from this point on. With the mechanism shown, it is also possible to halt programs (see Figure 8.15b), to suspend programs and to restart (see Figure 8.15c) and reinitialize (see Figure 8.15d) suspended programs. The Master is the only one that can start a distributed OPC UA Program. In the other cases, it is also possible that the status transition is done by a Slave. The only thing to note here is that if the Master does not initiate the state change, then, of course, not only the Master must be informed, but all other Slaves as well. In this way, it is also possible to stop the trajectory in the example and continue it at a later point in time. For the implementation of distributed OPC UA Programs, the current status of each component is provided via a publisher that represents the current state as an integer value.

Distributed State Changes in Real-Time

The adaptation and distribution of the state machine shown indicates that it is possible to start distributed processes on different components, but what still has to be shown is

that the state transitions of the distributed state machine also occur in real-time. The reaction time for a status change of a distributed OPC UA Program can be calculated as follows:

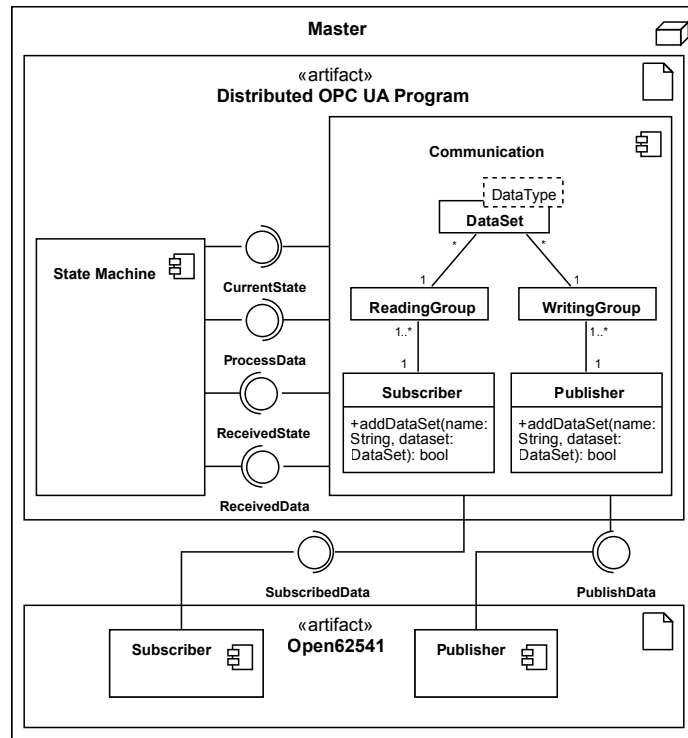
$$t_{\text{distributed state change}} = t_{\text{local state change}} + t_{\text{notification}} + t_{\text{local state change}}$$

The time required for a distributed state change consists of the time needed to change the status on the first device plus the time needed to notify other devices over the status transition. After receiving the notification, the status of the other devices must also be adjusted. Since OPC UA Pub-Sub over TSN is used as a communication channel, which is deterministic and has fixed cycle times, the notification is received within a fixed time barrier. In the worst case, this is the cycle time of the OPC UA Pub-Sub over TSN transmission plus the processing time of the message. The worst case occurs exactly when a state transition happens during the sending of the cyclic data, then a complete TSN cycle must be waited. The local status changes also take place within fixed time barriers. This results in a fixed time barrier for distributed state changes. For example, if the transmission cycle time is set to 250  $\mu\text{s}$ , a message can be processed within 10  $\mu\text{s}$ , and an internal state is changed in under 1  $\mu\text{s}$ , the maximal distributed state change time has an upper bound of 262  $\mu\text{s}$ . Assuming a conventional control cycle of 4 ms, up to 15 distributed transitions can still be performed.

The distributed OPC UA Programs can be used with and without the RealCaPP service architecture. In case the distributed OPC UA Programs are to be executed without the service architecture, there are components that encapsulate the real-time transmission via OPC UA. Figure 8.16 depicts the component diagram of a Master OPC UA Program. However, the Slave OPC UA Program has an identical structure. A distributed OPC UA Program consists of the State Machine and the Communication component. The State Machine describes the partial state machine with interfaces for communicating with other distributed OPC UA Programs. For the current state of the state machine and process data to be exchanged with other distributed OPC UA Programs, the State Machine component provides the interfaces `CurrentState` and `ProcessData`. In order to receive process data and the state changes of the other participating distributed OPC UA Programs, the interfaces `ProcessData` and `ReceivedState` of the Communication component are consumed. The Communication component encapsulates the OPC UA Pub-Sub communication, which makes it easy to create new publishers and subscribers in order to communicate with other devices. To ensure the interchangeability of the OPC UA stack, the interfaces `PublishData` and `SubscribedData` were added to the Distributed OPC UA Program artifact. For example, the open62541 OPC UA stack can easily be replaced by another OPC UA stack.

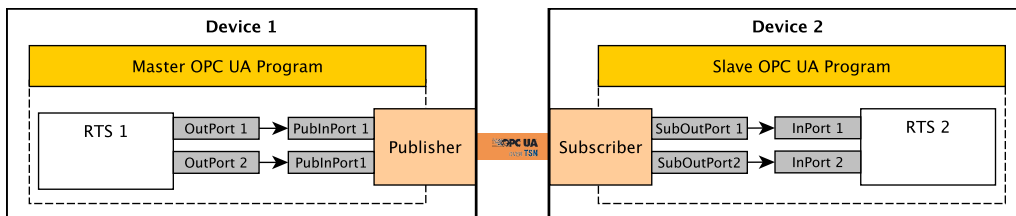
The distributed OPC UA Programs can also be used in combination with the RealCaPP service architecture. The distributed OPC UA Program is used to initialize the respective RTSs and then to execute them simultaneously. Figure 8.17 shows an example of an distributed RTS Network. The first step is to define which device is the Master OPC UA Program. The other participating devices are automatically Slave OPC UA Programs. In the example the first device (Device 1) is selected as Master. Consequently, Device 2 becomes a Slave. As mentioned before, RTSs have a three-step life cycle: INIT, RUN,

Distributed  
OPC UA Programs  
meets RTS  
Execution



**Figure 8.16.** Component diagram of a distributed Master OPC UA Program. The structure of the Slave is identical (See Eymüller et al. [51])

and STOP. This three-phase life cycle is now controlled via the distributed OPC UA Program state machines. In the INIT substate of the HALTED state of the distributed OPC UA Programs the `init()` methods of all RTSs are called. In addition, the execution order of the associated RTSs is determined as shown before. The Slave switches to the READY state after all RTSs are initialized and an execution order for the device is found. Thereafter, the Slaves informs the Master of the state change. The Master also waits until the execution order is determined, all RTSs are initialized and all Slaves are in the



**Figure 8.17.** Example for executing distributed RTS Networks with distributed OPC UA Programs



READY state. Then the Master switches to the READY state. After all distributed OPC UA Programs are in the ready state, the synchronized execution can start. For this, the Master informs the Slaves that the execution must be started. After that, all distributed OPC UA Programs change into the RUNNING state. In the running state, the RTSs are now executed by calling the `run()` methods in the previously calculated execution order. Also the transmission of the external port connections via OPC UA over TSN is started. If a distributed OPC UA Program is halted the `stop()` functions of all RTSs are called. To suspend the execution only the executions of the `run()` methods is stopped. After continuing, the execution sequence is resumed.

It was thus possible to show that distributed RTS networks can also be used for synchronously running applications.

## 8.5 Related Work

Several modular software architectures use data flow ports for the communication and information exchange between software components. Data flow ports have the advantage of loose coupling and enable an easy scalability of systems. Furthermore, data flow ports abstract the communication between components, allowing them to be located on different machines or nodes within a distributed system. Components can exchange data through data flow ports regardless of their physical location.

Tools like the graphical programming environment Simulink [3] already show the power of modular software components that can be combined into complex systems by connecting the ports of the modules. Simulink can be used, for example, to write controllers for robots, simulate systems, or develop vehicle assistance systems. In addition, the complexity of the modules can be reduced by nesting them. Since Simulink itself is only a programming environment, it is not intended to distribute the programs or make them interchangeable during runtime.

There are also function block diagrams for programming Programmable Logic Controllers (PLCs) defined in the IEC 61131-3 [82]. The execution logic is encapsulated in function blocks that have inputs and outputs. These inputs can be interconnected to combine multiple function blocks into one control program. The IEC 61499 standard [214] extends the IEC 61131-3 standard and defines how distributed control systems can be built by using these function blocks. The IEC 61499 also supports the distribution of control systems across multiple devices in a network. Connections between inputs and outputs that run across device boundaries are implemented via fieldbuses. There are also approaches to perform a reconfiguration with the IEC 61499 standard at runtime [163]. The standard does not define that the distributed execution must be done in real-time, so not all implementations are real-time capable [162].

Moreover, there are numerous software architectures for the implementation of modular distributed robotics applications. OpenRTM [14] stands for Open Robot Technology Middleware and is a software platform for component-based robotic system development. As with the concept presented in this thesis, the goal is the reusability of software components for robotics applications. The component-based architecture allows to create robot control systems by connecting and combining modular software components.

These software components are called RT-Components (Robot Technology Components) that encapsulate a specific functionality like a sensor, control component, or actuator. These components can be connected to each other via data ports and can exchange data. Moreover, RT-Components can contain service ports that implement a query-based data exchange approach, similar to RTS functions. If RT-Components are executed in a common real-time thread, executions in real-time are possible. However, no concept is provided for executing RT-Components in distributed systems with real-time guarantees. Moreover, there is no distinction between local and external communication, which is particularly bad for real-time capability. Later on, in 2012, the structure of the RT-Components was even defined as an official specification by the Object Management Group (OMG) for modular robotic software components [133].

Another framework that deals with the execution of robotic control systems is the Open Robot Control Software (OROCOS) [33, 34]. OROCOS also presents a modular framework for robotic control tasks. The focus of OROCOS is on the real-time execution of such control applications. But even with OROCOS, only the real-time capability is considered when executing on a single device and not distributed. In addition to libraries for the basic control of robots and robotic systems called Kinematics and Dynamics Library (KDL) [144], the Real-Time Toolkit (RTT) [145] is available for building and managing highly configurable and interactive real-time capable software components. Each component has a task context which contains the logic of the component. For communication with other task contexts data flow ports can be used to exchange data between components on one device with hard real-time guarantees. Components can be distributed across multiple devices using CORBA (Common Object Request Broker Architecture). However, the transmission via CORBA does not meet any real-time constraints [154]. CORBA is a defined standard that makes it possible for software components, written in multiple languages, to be distributed on multiple computers [131]. Even though there are real-time extensions for CORBA in combination with real-time networks [91, 109], there are no applications in combination with OROCOS.

Finkemeyer et al. [58, 59] have developed the Middleware for Robotic and Process Control Applications (MiRPA) that allows communication between local and distributed software modules with very small worst-case latencies ( $\ll 1$  ms). The middleware supports both client-server communication and publish-subscribe communication between software components. In this work, however, the focus is on the middleware. There is no mention of how the modular software components are built. Qnet is used for communication between distributed devices; standard ethernet protocols could not guarantee real-time. For the communication between distributed devices, a proprietary protocol for distributed interprocess communication called QNet [174] is used, which is only available in the real-time operating system QNX [175], with standard ethernet protocols no real-time can be guaranteed.

Another project that presents a component-based robot software platform is called OPRoS (Open Platform for Robotic Services) [87]. OPRoS uses modular software components that can have service, data, or event ports. The service ports allow other components to call methods of the other components. Data ports are used to exchange data. They can be processed either periodically or event-based. Events can be processed

via the event ports, which can trigger corresponding executions in the components. Data and events are implemented non-blocking, while service ports can block. For distribution to multiple devices, one of the following communication channels can be used: Socket messages over TCP, UPnP (Universal Plug and Play), or CORBA. The basic version of OPROS promises soft real-time. An extension for real-time applications has also been presented [88], but here, too, only periods with over 1 ms where possible.

The Robot Operating System (ROS) [176] also has concepts for the execution of distributed software components. In ROS, these modular software components are called Nodes. These nodes can communicate with each other by publishing and subscribing messages to data topics. For acyclic communication, nodes can also provide services, which are similar to remote procedure calls. Since messages are transmitted locally and distributed via TCP messages, there are no real-time guarantees for the data exchange between the nodes. ROS2 [116] now uses the Data Distribution Service (DDS) [150] for the communication between nodes because it already includes important features like UDP transport, distributed discovery, and security features. For ROS2, there is the possibility to use different DDS implementations. Many of the DDS implementations do not provide real-time guarantees. For example, the Eclipse iceoryx [48] DDS implementation uses shared memory to exchange information on one device. Unfortunately this does not work for distributed systems. A real-time capable DDS implementation is the Connex DDS professional from Real-Time Innovations (RTI) [178] that implements the OMG DDS for Real-Time Systems standard [132] and also uses shared memory for local transmissions and has consistent microsecond-order low latencies over standard ethernet. Therefore, a comparison between the RealCaPP implementation and a ROS2 implementation with the RTI DDS was made in Chapter 11.

One project that has been developed at our institute is the SoftRobot project [15, 71, 203]. The goal of the project was to enable complex, real-time critical robot task in JAVA. Therefore, two separate architectural tiers were designed: Applications can be developed and can be executed on top of the JAVA-based Robotics API, while real-time critical tasks are executed in the Robot Control Core (RCC). The RCC is based on the OROCOS framework. The RCC consists of calculation modules called real-time primitives that can be interconnected by dataflow ports. Graphs of these primitives are called RPI graphs (Real-Time Primitives Interface Graphs). These real-time primitives and RPI networks were used as the basis for implementing the RTS and RTS Networks. In addition, there were concepts for a service-oriented implementation of robot applications [72]. These concepts were developed using OSGi in the non-real-time JAVA part of the architecture. Therefore, in this thesis, an attempt was made to combine these two concepts of real-time executable services. Even if services were distributed over several devices, this was implemented in the JAVA part, and then a single RCC instance was used for real-time execution. The RCC does not provide for distribution.

There is also related work for the use of OPC UA Programs for the implementation of control processes. Dorofeev and Zoitl [45] demonstrated how an automation system can represent a manufacturing skill using an OPC UA Program. They developed a universal interface that enables a higher level of control to effectively coordinate the available skills. Additionally, Profanter et al. [168] utilized OPC UA Programs to model the skills

of industrial robots, while Kaspar et al. [93] adopted a similar approach by modeling PLC function blocks with OPC UA Programs. In addition to OPC UA Programs, the Robot Operating System (ROS) provides an alternative for executing long-running control processes. The ROS Action Protocol, provided by the ROS ActionLib [183], enables the execution of extended services through topic-based and publish/subscribe communication. In this context, ROS Actionlib provides a straightforward API for requesting goals on the client side and executing them on the server side using function calls and callbacks. However, these approaches mentioned above lack the capability for real-time communication and execution, limiting their applicability in comprehensive industrial communication scenarios. Moreover, all these approaches do not show synchronization between distributed processes.

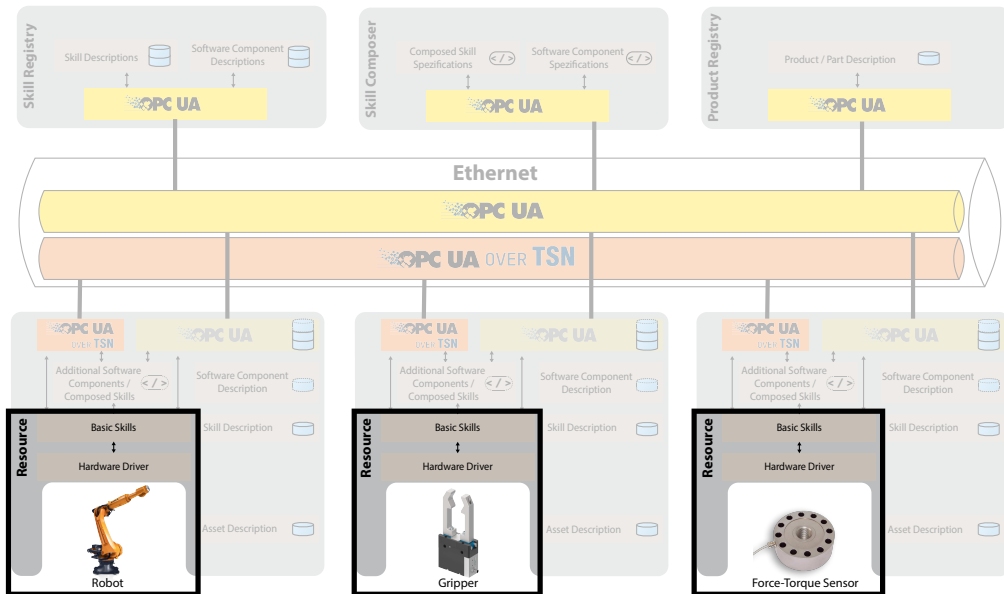
**Summary.** This chapter demonstrates the implementation of the Basic Skills for the individual resources. Abstract implementations are presented for different classes of resources. The use of uniform interfaces allows an effortless replacement of resources by other resources with matching skills.

# 9

## Implementation of Resources with the RealCaPP Service Architecture

<b>9.1 Industrial Robot Resources</b> . . . . .	<b>121</b>
<b>9.2 Sensor Resources</b> . . . . .	<b>125</b>
9.2.1 Force-Torque Sensor . . . . .	125
9.2.2 Digital Input Modules . . . . .	126
<b>9.3 Actuator Resources</b> . . . . .	<b>127</b>
9.3.1 Gripper Resources . . . . .	127
9.3.2 Screwer Resources . . . . .	129
9.3.3 Automatic Tool Changer . . . . .	132
9.3.4 Digital Output Modules . . . . .	133

So far, it has only been explained at a very high level of abstraction how to implement Basic Skills and Composed Skills in the RealCaPP service architecture. In order to ensure the reusability and interchangeability of skills, uniform implementations of the skills must be defined. If, for example, a parallel gripper in a robotic system is to be replaced by a vacuum gripper, it should be possible to change the gripper mechanically and, without making any changes to the code, to continue with the defined process. To achieve this, resources of the same type must also provide the same interfaces or skills. Therefore, it is attempted to define a uniform implementation for each type of resource. To find the commonalities of resources, they are divided into classes of resources. For these classes, an attempt is made to work out the basic skills that each resource of a particular class requires. Since these basic functions are not sufficient for some applications, there is the option of defining specializations of these basic resources. There are additional functions or interaction options besides the basic resource functionality for these specializations. For example, if there is a status lamp in a robotic system that indicates whether a system is active. The basic class of this lamp can be switched on and off via functions. A specialization would be a lamp that can not only be switched off and on but also



**Figure 9.1.** System architecture for a RealCaPP environment focusing on the the hardware drivers and the Basic Skills (cf. Figure 4.5)

provides a function to adjust the light color. Such specializations can be solved very well in software engineering by class inheritance.

The following tries to define these basic implementations for the different resources that occur in a robotic system. The main component of robotic systems is the robot itself. In Section 9.1, the common features of industrial robots are elaborated, and a unified implementation for the class of industrial robots is presented. Subsequently, in Section 9.2, it will be seen how sensors can be mapped in the service architecture. Implementations are defined for certain sensor types, like force-torque sensors (see Section 9.2.1) and digital input modules (see Section 9.2.2). Besides the sensors, implementations of actuators are defined in Section 9.3. Actuators can be tools for robots, such as grippers (see Section 9.3.1) or screwdrivers (see Section 9.3.2). To be able to change tools automatically in the sense of Plug & Produce, in Section 9.3.3 defines how tool changers can be addressed. Finally, the implementation of digital output modules is presented in Section 9.3.4.

Figure 9.1 shows the complete RealCaPP architecture, as already displayed in Figure 4.5, focusing on the points mentioned in this chapter. In this chapter, the focus is mainly on the hardware drivers of the components and the general implementation of Basic Skills for the hardware components.

## 9.1 Industrial Robot Resources

In order to define a uniform interface for the control of all industrial robots, an attempt is made to find the commonalities between the different robots. This involves looking at what types of movement there are, what parameters are relevant, and which data a robot provides. Robots can move in two motion spaces: Movements can be defined either in joint space, where each joint of the robot defines a dimension and the position of a robot is defined by the joint position of each axis, or in Cartesian space, where the position and orientation of the robot is specified by the coordinates of the Tool Center Point (TCP) in a defined three-dimensional Cartesian coordinate system. The TCP of a robot refers to the specific point on the robot's end effector, for example, the tool or device attached to the robot, that is used as the reference point for positioning and orientation calculations of the robot [85].

Robot Movements

Movements in the joint space are called *point-to-point (PTP) movements*. During PTP movements, the current joint position is tried to be converted into a given goal joint position as quickly as possible. Each robot axis is moved individually to the respective target position at a predefined velocity and acceleration. Influenced by the mechanical structure of the robot and the point to be approached, the movement results in a non-uniform, curved path of the TCP. With a PTP movement, the target point of the robot is reached in the fastest way. This does not have to be the shortest Cartesian path of the TCP.

PTP Movement

In Cartesian space, the movement of the robot is defined by its position and orientation in a three-dimensional Cartesian coordinate system. Movements can be defined as continuous paths defined by points. These continuous paths can be straight lines or curves. In Cartesian space, a motion is described by a mathematical function. This function describes the motion path and specifies how the TCP moves with respect to the robot base coordinate system in relation to time. A classical Cartesian movement is a *linear (LIN) movement*. When moving linear, a straight line is placed between the start and the end point in Cartesian space, and the TCP follows this straight line. This type of movement is mainly used as an approach movement of objects since the orientation of the tool is strictly predetermined. Another Cartesian type of motion is a defined circular path. Here, a circular arc is laid through three points in space, which are traversed by the TCP of the robot. This type of movement is called *circular (CIRC) movement*. Last but not least, movements in Cartesian space can be represented by splines. Here, a spline curve is laid through predefined points, and these are traversed. This is called *SPLINE movement*.

LIN Movement

CIRC Movement

SPLINE Movement

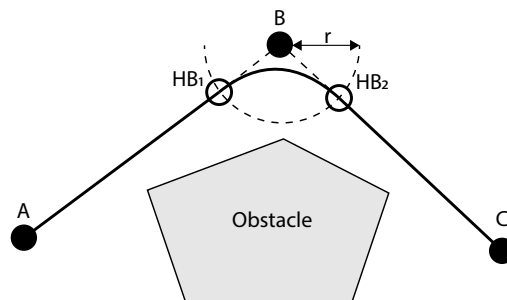
A more complex robot path can now be executed by linking these shown motions. The problem here is that the end points of the individual movements must always be reached exactly, and the start and end velocity for the movements are always defined as zero. As a result, at the end of each movement, the robot must be fully decelerated and then accelerated again for the subsequent movement. If the transition point between movements does not have to be reached exactly, there is the possibility of motion blending. *Motion blending* or also called *approximate positioning*, combines two motions by inserting a continuously differentiable curve between the motions. This means that

Motion Blending

the velocity does not have to be reduced during the transition between the movements, and two movements merge into one. Figure 9.2 shows an exemplary motion blending of two linear motions. The first linear motion is defined by points A and B. The second linear motion is defined by the points B and C. Without blending the robot movement would have to stop in point B and then start again with the second movement. For blending the movements, two help points ( $HB_1$  and  $HB_2$ ) are defined by the blend radius  $r$  around point B. A continuously differentiable curve, for instance, a parabolic blend [4, pp. 175], is then laid through the two defined help points, which unites the two movements with each other. This allows the velocity to be maintained during movement.

All these classic movements can be executed acyclically, e.g., one or more points are given, and then a corresponding path is planned and executed. For blended motions, it is necessary that the initial motion is known and the following motion is known because this is needed for the calculation of the blending curve.

In addition to the classic movement types, the possibility of adapting the position and orientation of the robot based on sensor values is also required. Besides the current position, it is also possible to adjust the programmed robot motions by sensor input. This is often referred to as *path* or *position correction*. Here, too, a distinction can be made between adjustments in joint space and adjustments in Cartesian space. In joint space, corrections can be described by offsets of the individual joint position. Cartesian corrections are usually given as a vector in three-dimensional space. In both cases, in joint and Cartesian space, the correction can be done on either absolute or relative values. In relative correction, the correction between the current value and the new target value is specified in each correction step. With the absolute correction, the correction always depends on the start value at which the adjustment starts. For path and position correction, a cyclic transfer of the correction values is required. These correction values are then immediately applied to the current position of the robot. For the path and position correction, the correction values must be entered in time for each cycle. Therefore, the correction data must be available in real-time.



**Figure 9.2.** Blending motion of two linear motions from A to B and B to C with the blend radius  $r$

Path and Position  
Correction

Robot Data  
and Parameter



uniform description of robots with their information and parameters has already been developed as OPC UA companion specification for robotics [137]. A OPC UA companion specification is a set of rules and guidelines that defines how OPC UA is used in specific domains and which information is necessary. From this companion specification, it can be derived which data are essential for a robot. An essential information for the robot is the current position and velocity. The current position and velocity of the robot can be displayed in the joint space with a list of the individual joint positions and velocities. In the Cartesian space, the current position of the robot is represented by the position and orientation of the robot's TCP in the three-dimensional space. The current Cartesian velocity is expressed by a vector velocity and a rotational velocity. In most use cases, this information is also expected in defined cycles and should, therefore, also be made available in real-time. There are also numerous parameters that shall be set in an industrial robot in order to influence the movements. First, the maximum velocity and the maximum acceleration of the robot can be influenced. Here, too, a distinction is made between the maximum velocity and acceleration of the TCP, which represents the Cartesian velocity and acceleration, and the maximum velocity and acceleration of the individual joints. Another parameter that influences the robot motion is the blend radius when blending motions. Other important parameters are the selection of reference coordinate systems and tool dimensions. To avoid having to express all positions in space in the base coordinate system of the robot, it is possible to define reference coordinate systems and use these for movements. These reference coordinate systems are usually used for the position reference to the corresponding processing stations of the robot. This gives the possibility if the position of the processing station changes, that not all points in the processing station have to be changed, but only the reference coordinate system has to be changed, and all points relevant to the processing station are expressed in this reference coordinate system. A similar concept is used for the description of the TCP with different tools. Depending on the tool used, the TCP of the robot changes. The robot must be able to set an offset for each tool from the robot flange to the TCP of the tool. This allows the exchange of tools of the same type without adjusting the interaction points. If, for example, a longer gripper is used, the grip point does not have to be changed, but only the offset of the new tool must be adjusted accordingly. All these parameters are set acyclically and influence all subsequent movements. An important aspect of robots is kinematics. In order to transfer positions from the axis space into the Cartesian space and vice versa, the direct and inverse kinematics calculation for the individual robot is required. With the help of these calculations, it can also be checked whether given positions can be reached with the respective robot.

On the basis of the interaction possibilities shown with an industrial robot, a uniform representation of the robot in the form of an RTS is developed. Through the standardization and abstraction of the Basic Skills of an industrial robot, it is possible to simply replace one robot with another without having to change the application. Figure 9.3 shows the abstract RTS implementation of an industrial robot. For the RTS representation, the UML syntax in class diagrams was used for the functions and ports. All data that the robot cyclically requires and provides were defined as ports of the RTS. For example, the current Cartesian position of the robot is represented by the

Industrial  
Robot RTS

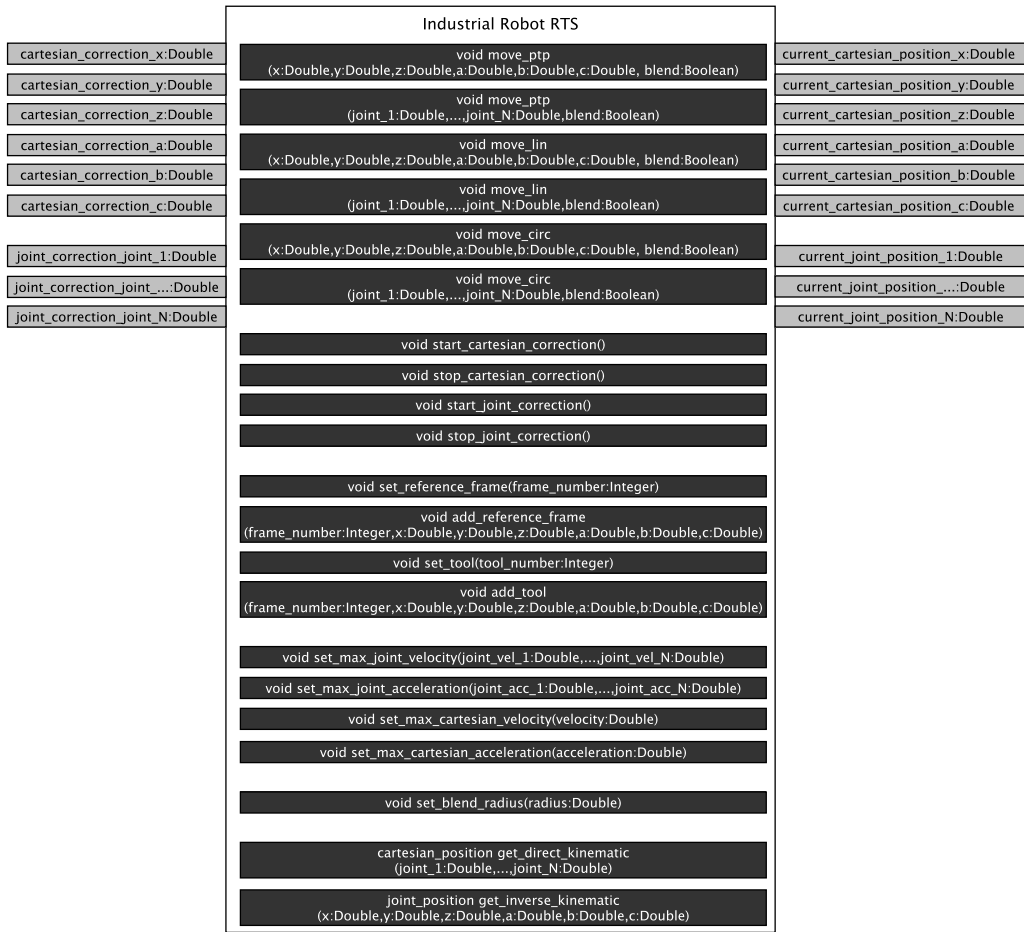


Figure 9.3. Abstract RTS of an industrial robot

OutPorts `current_cartesian_position_....`. All acyclic data were implemented as functions of the RTS. The setting of the maximum Cartesian velocity is represented, for instance, by the function `void set_max_cartesian_velocity(velocity:Double)`. In addition, there are also interactions with the robot based on both cyclic and acyclic interactions. An example of this is the position and path correction. For this purpose, there is both an acyclic function for starting and ending the correction (see functions `void start_cartesian_correction()`, `void start_joint_correction()`, `stop_cartesian_correction()` and `stop_joint_correction()`) and a cyclic interface for the corresponding correction values (see ports `Cartesian_correction_....` and `joint_correction_joint_....`). Of course, this abstract RTS can be extended by further functions and ports, but due to the reusability, care should be taken that each robot can implement these Basic Skills.

## 9.2 Sensor Resources

In addition to the robot itself, sensors are an important component in robotic systems. Sensors in a robotic system are used to gather information about the robot's environment, the state of the robot, or the state of a running process. This information is then used to sense the environment and react to changes in the environment. There are numerous types of sensors that can be used for robotic systems: There are vision sensors like cameras and depth sensors that are used to recognize objects and the surroundings of the robot. Range sensors, like ultrasonic sensors or laser sensors, can be used to provide distance measurements. Proximity sensors, such as capacitive or inductive sensors, are used to sense objects without physical contact and are often used for safety purposes. For the measurement of the robot's acceleration or orientation, inertial sensors, for instance, gyroscopes or accelerometers, can be used. Also, forces and torques can be measured by force-torque sensors or tactile sensors. In addition to the sensors mentioned above, there are numerous other sensor systems that can be used for robotic applications. What all these sensors have in common is that they provide sensor data, and in most cases, it is necessary that these sensor values are read out and processed as quickly as possible. Especially when sensors are used to influence the movement of the robot, real-time data transmission and real-time processing are necessary. With most sensors, the sensor data is provided cyclically. In addition, many sensors offer the possibility to set parameters or provide functions to calibrate the sensor. Therefore, most sensors have RTS out ports for the respective sensor data and offer RTS functions to set parameters or calibrate the sensor. In the following, two implementations of sensors are presented, which are used in the later case study.

### 9.2.1 Force-Torque Sensor

In the context of robotics, a force-torque sensor is a type of sensor used to measure forces and torques exerted by a robot. For this purpose, the forces and torques are measured in the different directions of movement. Primarily, force-torque sensors are used, which can measure all six Degrees of Freedom (DoF). This means that there are three Degrees of Freedom for the forces and three Degrees of Freedom for the torques. The force-torque sensors are usually placed between the robot and the tool to measure

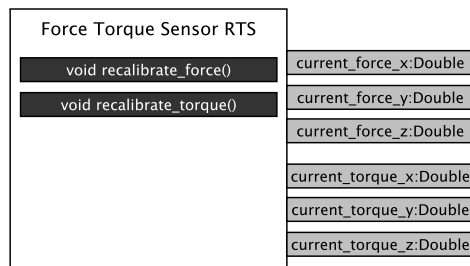


Figure 9.4. Abstract RTS of a force-torque sensor

the forces and torques acting on a tool or respectively, the forces and torques a tool exerts on a part. This allows the robot to detect contact with objects, to apply controlled forces and moments during assembly or finishing tasks, or to measure the weight of an object attached to the tool. Such sensors also can be used to detect abnormal conditions. For example, excessive forces or torques can be recognized, which would damage the robot or components. Overall, through the integration of force moment sensors, a robot acquires more sensitivity, precision, and safety when performing tasks that involve physical interactions or contact. The force and torque values are provided cyclically and have only a very short validity due to the high velocities of the robots. Since these are often used for the control of robot movements, these data are real-time critical. Force-torque sensors offer the possibility to recalibrate or zero the force and torque values as required. The recalibration is used so that only the resulting force is output and, for example, the weight forces of the tools can be neglected. Figure 9.4 shows the abstract RTS implementation of a six-axis force-torque sensor. OutPorts are used for the three force values (see OutPorts `current_force_...`) as well as for the three moment values (see OutPorts `current_torque_...`). Furthermore, the RTS provides two functions: The `recalibrate_force()` function can be used to zero all force values, while the torque values can be recalibrated by the function `recalibrate_torque()`.

### 9.2.2 Digital Input Modules

Another concept that is very similar to sensors or can also be used for sensors is digital input modules. Digital input modules are devices that are used to interface robotic systems with external digital signals. These modules are responsible for converting digital signals from sensors, switches, or other devices into a format that the robotic system can understand and process. Typically, digital input modules consist of a set of input channels that can process electrical signals representing the state of a device or sensor. These signals are in the form of logical high (1) or logical low (0), representing the presence or absence of an electrical signal. Often, a 24 V direct current (DC) voltage is used to display such signals, where 0 V represents the logical 0 and 24 V the logical 1. However, there are also other modules that realize it via other voltages or different currents. For the robotic system, the inputs are then interpreted as binary signals and can be used for various applications. For instance, binary sensors like photoelectric sensors, inductive sensors, or passive electrical components like start and stop buttons can be integrated into robotic systems by connecting them to digital input modules.

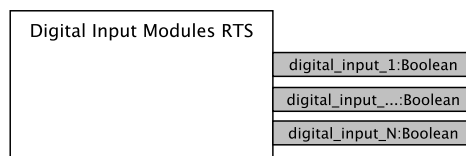


Figure 9.5. Abstract RTS of a digital input module

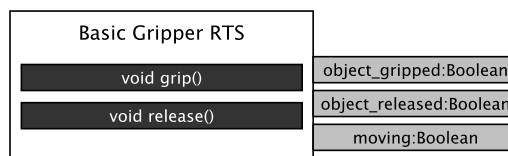
Since it is difficult for computer systems to display continuous processes, the inputs are read at cyclic intervals and provided as cyclic information. This data is also highly real-time critical since, for example, an end stop could be installed on the digital input module, which must immediately bring the robot to a stop when triggered. Figure 9.5 shows the abstract implementation of a digital input module with N inputs. Each input of the module is represented by a boolean OutPort.

## 9.3 Actuator Resources

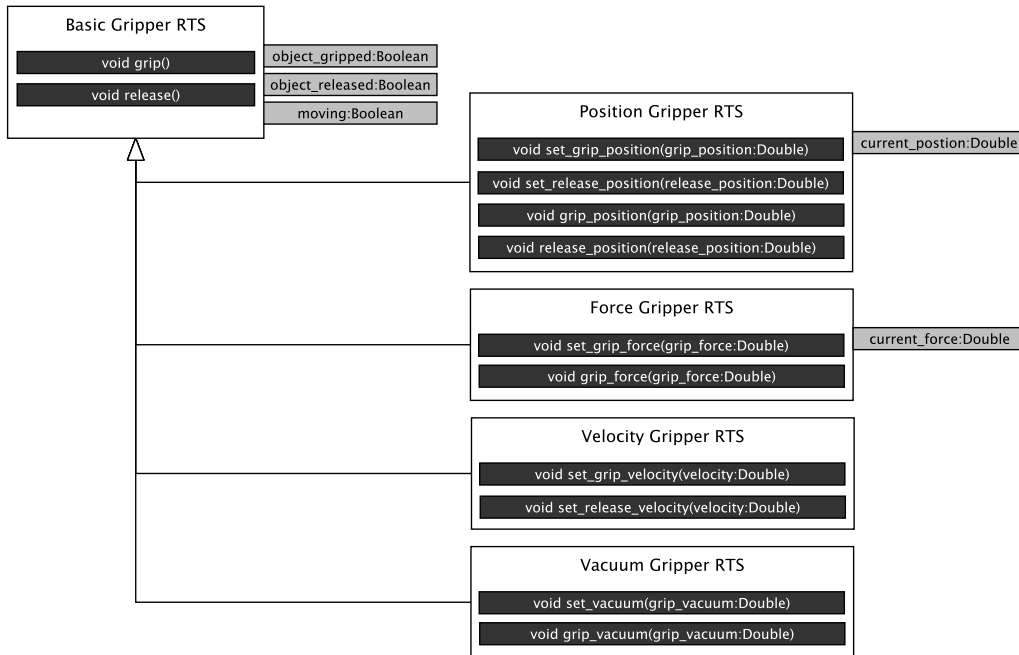
Now that it is possible to communicate with the robot and observe the environment with the help of sensors, it is necessary to interact with the environment through other active components. Actuators are components or devices responsible for generating physical motion or causing mechanical changes in a robotic system. Actuators can be, for example, tools of the robot, active clamping devices, linear axes, conveyor belts, and much more. As with the sensors, it is difficult to find a uniform control interface due to the diversity of actuators. Therefore, different types of actuators are considered here.

### 9.3.1 Gripper Resources

A frequently used type of actuator in robotic systems is the gripper. One reason for this is the variable usability of grippers. Grippers can be used in robotic systems to move objects, assemble objects, hold objects, insert objects into each other, or interact with objects. Since not only the tasks but also the objects vary greatly, there are numerous types of grippers. The grippers in robotic systems can differ based on their design, type of gripping, e.g., mechanical or suction, the mode of operation of the gripper, e.g., electric or pneumatic, and of course in the type of control. As already mentioned in Section 8.2, where the focus was on the reusability of software components, a common control basis for all these grippers is to be found. What all grippers have in common is the activation and deactivation of the gripper, meaning the gripping of objects and the releasing of objects. In addition to these fundamental skills, many grippers can determine whether an object has been successfully grasped or released. From these requirements, a basic gripper, as shown in Figure 9.6, was derived. Opening and closing can be time-critical, depending on the application, but is called acyclically. It is therefore implemented by the functions: `grip()` for closing the gripper and `release()` for opening the gripper. The state of whether an object has been gripped or released is transmitted cyclically by the boolean OutPorts `object_gripped` and `object_released`. Also a state for the



**Figure 9.6.** Abstract RTS of a basic gripper



**Figure 9.7.** Specialization of gripper controls by inheritance of abstract basic gripper RTS

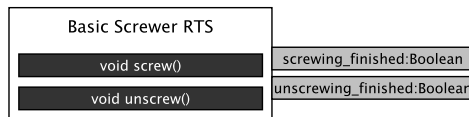
movement of the gripper is transmitted cyclically by the boolean OutPort `moving`. Since, especially with mechanical grippers, gripping is realized by a movement, a longer time can elapse between triggering the gripping and the successful gripping of the object. In order to integrate further control concepts, it is possible to inherit this abstract basic gripper RTS. This means there are specializations in the control, but it is also possible to fall back on the Basic Skills. This makes it possible to exchange several grippers that can grip a certain object, even though they may be different specialized grippers. Figure 9.7 shows the specializations of the basic gripper RTS. Since the RTS are modeled in an object-oriented way, it is possible to use inheritance. When inheriting RTSs, both the functions and the ports are passed on to the inheriting RTS. For the representation of the inheritance, the UML syntax was also used here. Multiple inheritances are also possible. A gripper can have several specializations. A specialization of the basic gripper is the position-controlled gripper (see Position Gripper RTS). For grippers with a large stroke of the gripper jaws, there is often the possibility to specify positions for open and closed. This makes it possible, for example, to grip components with different widths. Therefore, there are functions for setting the position, e.g., `set_grip_position(grip_position:Double)` and functions that combine setting the position and start gripping or releasing, e.g., `grip_position(grip_position:Double)`. In addition, the current position can be queried for these grippers, which is provided via the OutPort `current_position`. A further specialization is the force-controlled gripper. With this type of gripper, a gripping force can be defined, and the gripper jaws remain

closing until the defined gripping force is reached and hold this force on the object. This can be used, for example, to grip objects whose size is unknown, to grip objects that are very delicate and can only withstand a certain force, or to realize a friction-locked gripping with defined forces. As with the position-controlled grippers, it is possible to set the gripping force using the `set_grip_force(grip_force:Double)` function. The currently applied gripping force is provided in the `OutPort` `current_force`. A specialization closely related to the position and force-controlled grippers are grippers whose velocity can be defined, see `Velocity Gripper RTS`. With this type of gripper, the velocity of the gripper jaws can be defined. For this, there is the function `set_grip_velocity(velocity:Double)` to set the closing velocity of the gripper jaws and a function to set the release velocity, see `set_release_velocity(velocity:Double)`. Most of the specializations shown are related to jaw grippers. However, there are also other types of gripping, such as vacuum grippers, in which an object is held in place by negative pressure. Here, there is usually the possibility to adjust the vacuum. The `Vacuum Gripper RTS` is a specialization of the basic gripper, where there is a function `set_vacuum(grip_vacuum:Double)` to specify the vacuum and a function to specify the vacuum and then perform the grab, see `grip_vacuum(grip_vacuum:Double)`. As already mentioned, combinations of specializations can be realized by multiple inheritance. For example, there are parallel grippers where the velocity can be adjusted, and the gripper can be controlled either by position or force. In this case, the gripper inherits from the `Position Gripper RTS`, the `Force Gripper RTS` and the `Velocity Gripper RTS`. Of course, there are other specifications of grippers not mentioned and treated here. These can easily be integrated into the architecture by further inheritances.

### 9.3.2 Screwer Resources

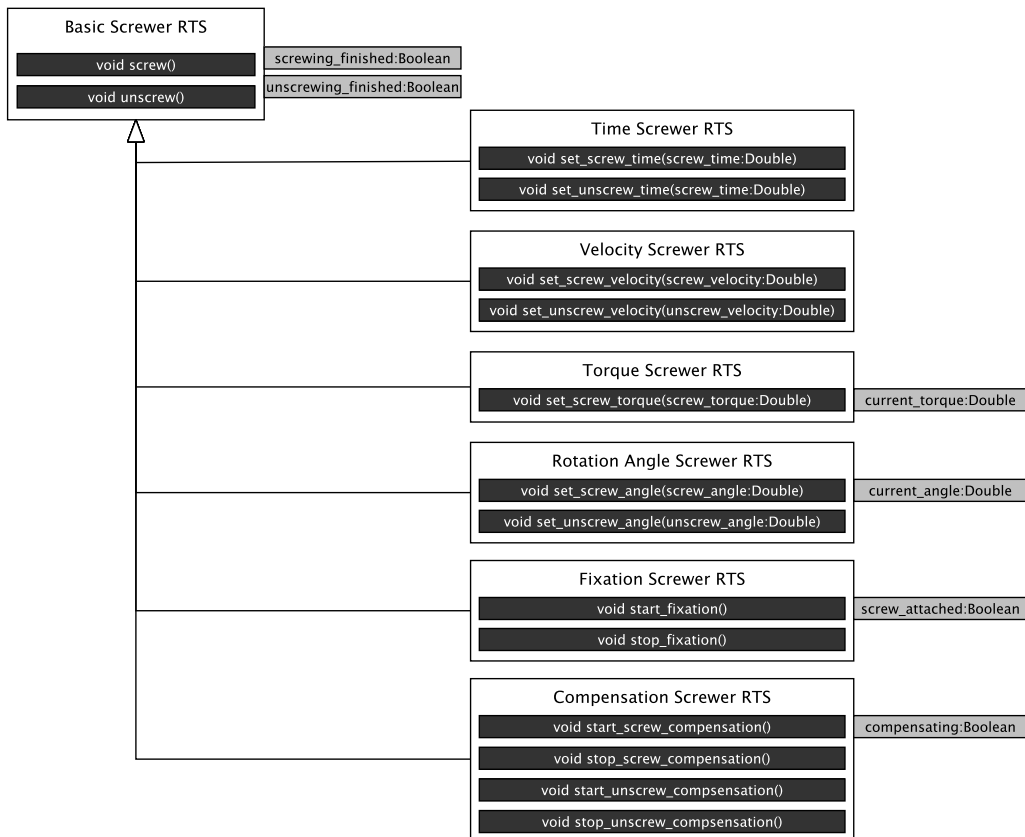
Besides handling objects, robots are also frequently used to join components. For example, components can be screwed together. In robotic systems, a screwer is an end effector designed to tighten or loosen screws and is mounted on a robot to automate the process of handling screws. Robotic screwers generally consist of a screwbit that matches one type of screw head. If different screw types are to be used for one application, it is necessary to change the screwbit. Furthermore, a screwdriver consists of a rotation mechanism, which allows to screw out or in the screws. The rotation mechanism can be electric, pneumatic, or hydraulic, depending on the design and requirements of the robotic system. Another important component of a robotic screwer is a mechanism for picking up screws and holding the screw in position during the screwing process. There are also numerous options for the handling of screws. There are screw magazines that automatically insert the screw into the screwdriver, passive systems, and active clamping or gripping mechanisms. Passive screw holders, for example, have magnets on the screw bit to hold the screw. Active screw holders retain the screws, for example, by a vacuum or special grippers on the screwbit. Depending on the screwing use case, there is also the option of attaching an angular head between the drive system and the screwbit to be able to screw in hard-to-reach places. When tightening screws, there are several parameters that are relevant to the screwing process: A screw can be tightened with a defined torque. There are possibilities to actively or passively control the torque.

In passive systems, there are mechanics installed that overrun when a defined torque is reached. However, there is also the possibility of actively measuring the torques using torque sensors and controlling them accordingly. With other screwing systems, only the screwing angle is specified, where the actuator rotates until the specified angle is reached. Nevertheless, there are also systems where only the time is specified, how long the actuator is active. This is often used either for unscrewing screws or when the screwdriver is used in combination with a passive torque system. Another distinguishing feature of screwers in robotic systems is the compensation and tracking of the screw movement. When screwing in a screw, the screw head moves with a certain velocity in the direction of the object to be screwed depending on the thread pitch of the screw and the velocity of rotation of the actuator. This movement must be compensated either by the robot or by the screwdriver itself. For compensation systems in the screwdriver itself, there are active and passive solutions. For example, mechanical spring systems can be used to compensate for the movement through a spring. For active systems, either pneumatic cylinders or electric linear axes are used. However, there are also screwers that are completely rigid, and the movement must be made exclusively by the robot. In this case, combinations with force torque sensors are often used, and the screws are screwed in in a force-controlled manner. As just shown, screwdrivers can be controlled and parameterized in different ways. Like with the grippers, a way is found via several specializations to map as many control types as possible. Each screwdriver can start a screwing process or an unscrewing process. Therefore, a basic screwdriver RTS was developed, as shown in Figure 9.8. In addition to the two functions for starting the movement (`screw()` and `unscrew()`), there are also boolean OutPorts to indicate that the screwing or unscrewing process has been completed, see `screwing_finished` and `unscrewing_finished`. For parameterizing the screwing process or adding additional features provided by some screwers, specializations were defined by inheritance, which can be combined arbitrarily by multiple inheritance. Figure 9.9 shows the specializations of the basic screwdriver. A first specialization is the setting of times through the `Time Screwdriver` RTS. This means that the time for tightening and unscrewing can be set by the methods `set_screw_time(screw_time:Double)` and `set_unscrew_time(unscrew_time:Double)`. This can also be selected in combination with other specializations as an abort criterion for an unsuccessful screwing operation. If, for example, a torque is not reached within a certain time, the screwing process is aborted. With many screwdrivers, it is possible to adjust the screwing velocity. The `Velocity Screwdriver` RTS is a specialization for this. Therefore, the velocity for turning in can be set by the function `set_screw_velocity(screw_velocity:Double)` and the



**Figure 9.8.** Abstract RTS of a basic screwdriver





**Figure 9.9.** Specialization of screwdriver controls by inheritance of abstract basic screwdriver RTS

function `set_unscrew_velocity(unscrew_velocity:Double)` makes it possible to define the screwing out velocity. For screwdrivers with adjustable torque, the specialization `Torque Screwer RTS` was defined with a function for defining the tightening torque (`set_screw_torque(screw_torque:Double)`). The double `OutPort` `current_torque` can be used to read out the current applied torque. In addition to the tightening torque, the tightening angle for screwing can be defined with the specialization `Rotation Angle Screwer RTS`. This specialization allows the definition of the angle of rotation for screwing and unscrewing with the two functions `set_screw_angle(screw_angle:Double)` and `set_unscrew_angle(unscrew_angle:Double)`. The current rotation angle is likewise provided by the `OutPort` `current_angle`. Apart from specializations for setting the screwing parameters, there is also a specialization for screw systems with active screw fixation. The `Fixation Screwer RTS` is defined, which has a function to start the fixation of a screw (`start_fixation()`) and one to release the screw (`stop_fixation()`). Moreover, with these active systems, it is generally possible to see whether a screw has been successfully picked up. The `OutPort` `screw_attached` can be used to check whether a screw has been successfully picked up and fixed to the screwbit. Fur-

thermore, there is the feature of active compensation of the screw movement. This allows to activate and stop a motion control to compensate the screw movement via linear axes or pneumatic cylinders. The functions `start_screw_compensation()` and `start_unscrew_compensation()` activate the compensation, while the functions `stop_screw_compensation()` and `stop_unscrew_compensation()` deactivate the movement compensation. The boolean OutPort `compensating` provides information on whether the screwdriver is actively compensating at the moment. Of course, numerous other specializations are also conceivable here, for which only the basic gripper RTS must be inherited again.

### 9.3.3 Automatic Tool Changer

Another actuator type is an automatic tool changer. An automatic tool changer is a mechanism that enables robots to exchange different end effectors during operation automatically. This allows the use of different tools on one robot without manual intervention and thus enables the adaptation of robots depending on the process or process step. In addition to the mechanical connection of the tool to the robot, the tool changer also ensures that the tools are supplied with power and that communication is possible between the robot and the tool. Tool changers usually consist of a robot side with a locking mechanism and media adapters to provide various types of media for the different tools, such as power, compressed air, network, and IOs. Via tool change plates it is possible to combine the respective tools with the robot side. For the required media of the tool, there are media adapters on the tool change plate to feed through the media. The locking mechanism can be realized by different types of actuators. Besides mechanical clamps, quick-release mechanisms, magnetic couplings, there are also pneumatic locks by extending bolts for the fixation of the tool. There are systems that are actively locked, for example, by extending cylinders, and there are passive tool changers that mechanically unlock when the tool rack is accessed and lock again when the tool rack is left. Figure 9.10 shows the implementation of a tool changer with active locking. The Basic Tool Changer RTS has a function for locking the tool to the tool changer (`lock_tool()`) and the function `unlock_tool()` for releasing the tool. The status of whether a tool has been successfully deposited or picked up is defined via the boolean OutPorts `tool_attached` and `tool_released`. To ensure that tools are not released while moving the robot or at an unexpected position, the InPort `over_tool_station` is used to check whether the robot is currently above a tool station. For example, an inductive sensor can be mounted on the tool changer and a counterpart

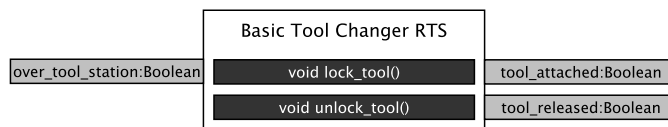
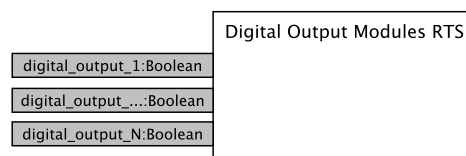


Figure 9.10. Abstract RTS of a basic tool changer

on the tool station. The OutPort of the sensor RTS would then be connected to the InPort of the tool changer RTS.

### 9.3.4 Digital Output Modules

Analog to the digital input modules, there are also digital output modules. With these modules, data values are converted into outgoing current signals. Digital output modules typically consist of a set of output channels that can be activated and deactivated individually. These digital signals can be used, for example, to control actuators, control lamps, trigger devices, or for the modulation of light signals, like photoelectric sensors. Figure 9.11 shows the implementation of a RTS for digital outputs. The output data is provided cyclically via the OutPorts `digital_output_...` and written to the outputs at defined intervals.



**Figure 9.11.** Abstract RTS of a digital output module



**Summary.** This chapter evaluates the RealCaPP architecture in two robotic cells. To demonstrate the reusability of the software components, the execution distributability, and real-time capability, four case studies were examined. These case studies range from industrial robotic processes, such as screwing aluminum profiles, to everyday tasks that can be done by a robot, like making coffee.

# 10

## Evaluation of the Case Studies for Robot-Based Automation

<b>10.1 Structure of the Robot Cells</b> . . . . .	<b>136</b>
10.1.1 A Flexible Industrial Robotic Cell: WiR Augsburg Innovation Laboratory . . . . .	136
10.1.2 Robarista Cell . . . . .	140
<b>10.2 Implementation of RealCaPP Concepts with Real Hardware Components</b> . . . . .	<b>141</b>
10.2.1 KUKA KR Industrial Robots . . . . .	142
10.2.2 Gripper of the KR90: Zimmer Group GEH6180 . . . . .	143
10.2.3 Screwer of the KR90: Stoeger SPATZ 30 . . . . .	146
10.2.4 Force-Torque Sensor of the KR90: ME-Meßsysteme K6D80 . . . . .	149
10.2.5 Grippers of the KR10 and KR6 . . . . .	149
<b>10.3 Hand Guiding of Industrial Robots</b> . . . . .	<b>151</b>
<b>10.4 Assembly of a Circuit Board Component</b> . . . . .	<b>154</b>
<b>10.5 Assembly of Aluminium Structures</b> . . . . .	<b>158</b>
<b>10.6 Robarista: A Robot Making Coffee</b> . . . . .	<b>165</b>

In order to demonstrate the results and application areas of the RealCaPP architecture, different robotics case studies were developed. These case studies consist of handling and assembly processes that are executed with different industrial robots, sensors, and other actuators. The goal is to cover all areas of Plug & Produce, from plugging in new resources, exchanging their self-descriptions and skills, setting up a real-time communication between these resources, and executing processes with real-time guarantees in the distributed robotic system. These case studies shall also cover the following challenges:

- Reusability of modular software components (services) for different case studies.
- Adding and exchanging of resources and services at runtime.

- Matchmaking between process requirements and the global system description from the self-descriptions of the resources.
- Execution of robotic processes in real-time, for example, sensor-controlled movements.
- Any distribution of modular software components in the plant is possible. It can be executed centrally as well as distributed.

To meet all these challenges, two independent robot cells were used for the implementation of a total of four case studies.

In Section 10.1, the structure of the two robot cells is described. In the following, the different case studies are presented. The hardware resources used in the robot cells are explained in detail in Section 10.2 and the respective implementations in the RealCaPP architecture are presented. The first case study deals with the sensor-controlled movement of industrial robots by force-torque sensors and how this can be used for hand-guiding industrial robots, see Section 10.3. In Section 10.4 a first assembly process is examined. In this case study a circuit board is inserted into a plastic component shell. A slightly more complex assembly process is described in Section 10.5. In this case study, aluminum groove profiles are screwed together with angle connectors to form aluminum structures. In addition to these industry-related use cases, the final case study examines how the elaborated concepts can be applied to a Robarista application, see Section 10.6. In this application, an industrial robot prepares coffee using a portafilter machine.

## 10.1 Structure of the Robot Cells

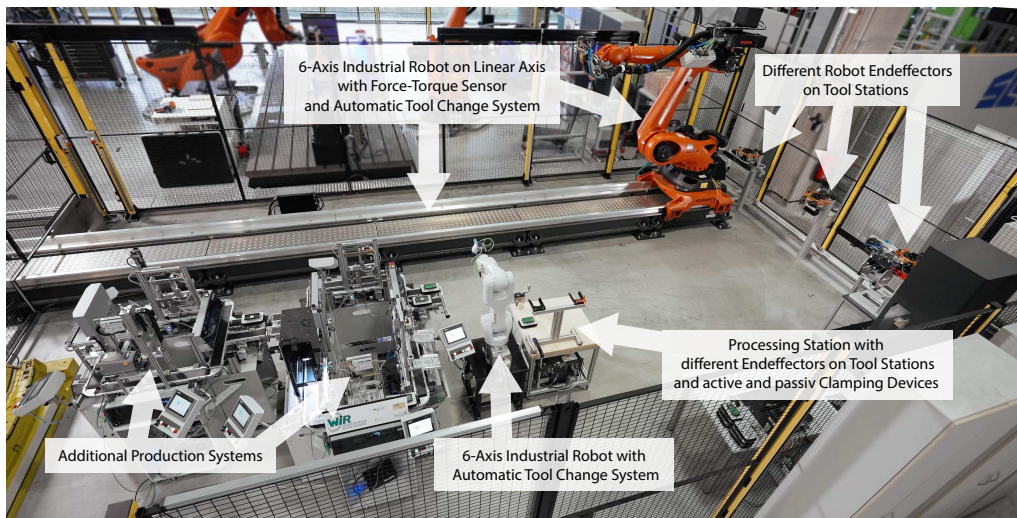
Two completely different robot cells were used to carry out the different case studies. This also allowed testing of how the concepts work in heterogeneous robotic cells. Due to the different payloads of the used robots and the different changing systems with more or less media feed-through, different designs of Plug & Produce are possible. Especially in robotic systems with small robots, it is difficult to attach powerful computing units to the end effectors. This changes the distribution of the control components (e.g., Industrial PCs (IPCs)) in the plant and thus also the possibility of distributing software components accordingly. Some concepts cannot be implemented on small robots with low payload, or only partially, due to the size and weight of the control components. Appropriate alternative approaches have been developed for this purpose. For this reason, alternatives were developed for some concepts, as will be shown later.

### 10.1.1 A Flexible Industrial Robotic Cell: WiR Augsburg Innovation Laboratory

The first robotic cell was built as part of the innovation laboratory in the WiR Augsburg project<sup>1</sup>. WiR Augsburg is a German acronym and stands for Wissenstransfer Region Augsburg (WiR Augsburg) and translates as knowledge transfer region Augsburg. In the WiR project, an innovation laboratory on the topics of "Digital Engineering and

---

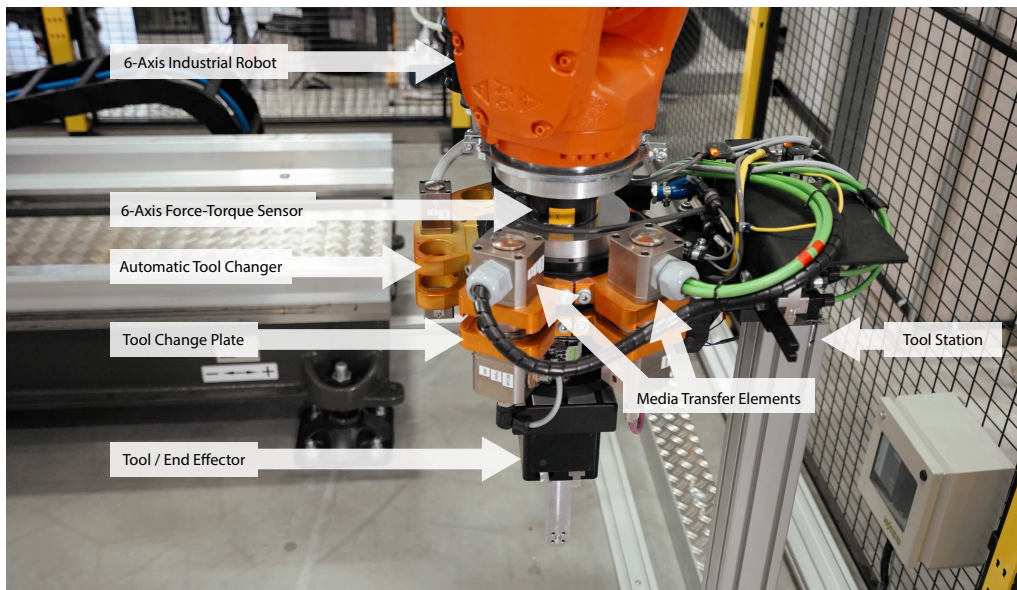
<sup>1</sup><https://www.uni-augsburg.de/en/fakultaet/fai/isse/projects/wir-augsburg/>



**Figure 10.1.** Flexible industrial robot cell of the WiR innovation laboratory with two robots, additional production systems, and processing stations

Automation" was set up. One goal of the innovation laboratory was to build up a flexible robotic cell to conduct research in the field of Industry 4.0 technologies and concepts such as Plug & Produce and big data analytics. Flexibility in this context means that different production and handling processes can be implemented in the plant by adapting the cell by changing the end effectors of the robots. In addition, there is enough space to add processing stations or additional production systems. The additional production facilities also give the opportunity to show interactions between different types of installations.

Figure 10.1 shows a picture of the flexible robot cell of the innovation lab. The main component of the system is a six-axis industrial robot KUKA KR90 R3700 prime K [108] with a payload of 90 kg on an 11 m KUKA KL4000 linear unit [105], which can cover the entire area of the cell due to its large reach of 3.7 m and the additional linear axis. In addition, there is a second six-axis industrial robot KUKA KR10 R900 [106] with a payload of 10 kg and a range of 0.9 m. This robot is fixed on a pedestal and can interact with the additional production units, perform tasks together with the KR90, or carry out handling or production tasks on one of the processing stations. In addition to the two industrial robots, the innovation laboratory has space to set up additional production systems or processing stations for the robots. An example of such an additional production system is the shown CP Lab (Cyber-Physical Lab) system of the company Festo. The CP Lab system is a modular production system in which the modular components are connected to each other by conveyor belts. With the help of component carriers, equipped with RFID chips to provide information about the components, components can be passed between the modular components. Based on the information stored on the RFID chips, the component carriers can be routed by active turnouts through the system. Through workstations above the conveyor

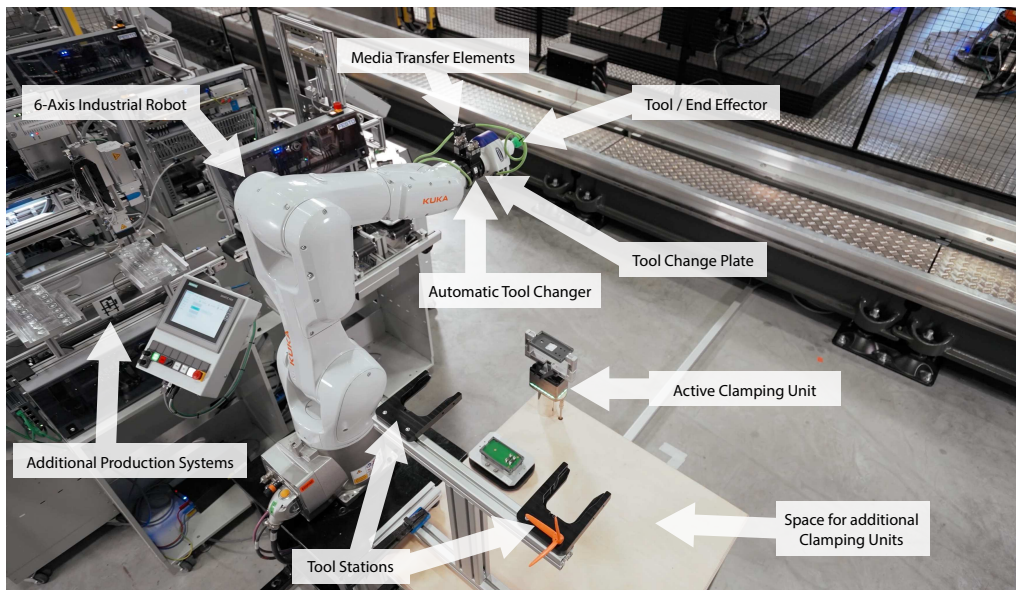


**Figure 10.2.** Structure of the complete end effector of the KUKA KR90. A six-axis force-torque sensor is attached to the robot flange, to which the automatic tool changer is attached. Different tools can be attached to the tool changer via tool plates.

belts, processes can be carried out. With the modular system, workstations can be mounted above the conveyor belt. The workstations can execute one process step each, like placing different components onto the component carriers, drilling holes into components, pressing multiple components, checking components with cameras, or labeling components. Furthermore, there are components that can output finished components and component locks that can eject components to the robotic system or reinsert components into the production system by the robots. The concept of the flexible robot cell is finalized by processing stations for the robot. In this case, processing stations are fixed structures or machining tables with passive or active clamping devices. This allows components and parts to be inserted into the clamping fixtures by the robot and to be further processed or machined accordingly by the robot. Active clamping devices are, for example, grippers attached to the processing stations. Passive holders are, for example, component holders that prevent inserted components from twisting or specify an exact position of the component. Additional parts, components, or tools can also be attached to the processing stations by these passive fixture devices.

In order to be able to execute processes with uncertainties, force-controlled processes, or to record and monitor the process forces and torques that occur during processing or handling operations, a six-axis force-torque measuring cell was attached to the flange of the KR90. The used force-torque sensor is a K6D80 [126] of the company ME-Meßsysteme that can measure forces up to 2 kN and torques up to 100 Nm. The associated measuring amplifier GSV-8AS [125] is mounted on the swing arm of the robot. Figure 10.2 shows the complete assembly between the robot flange and the tool.





**Figure 10.3.** Processing station with an additional robot, active clamping units, and the CP Lab production system

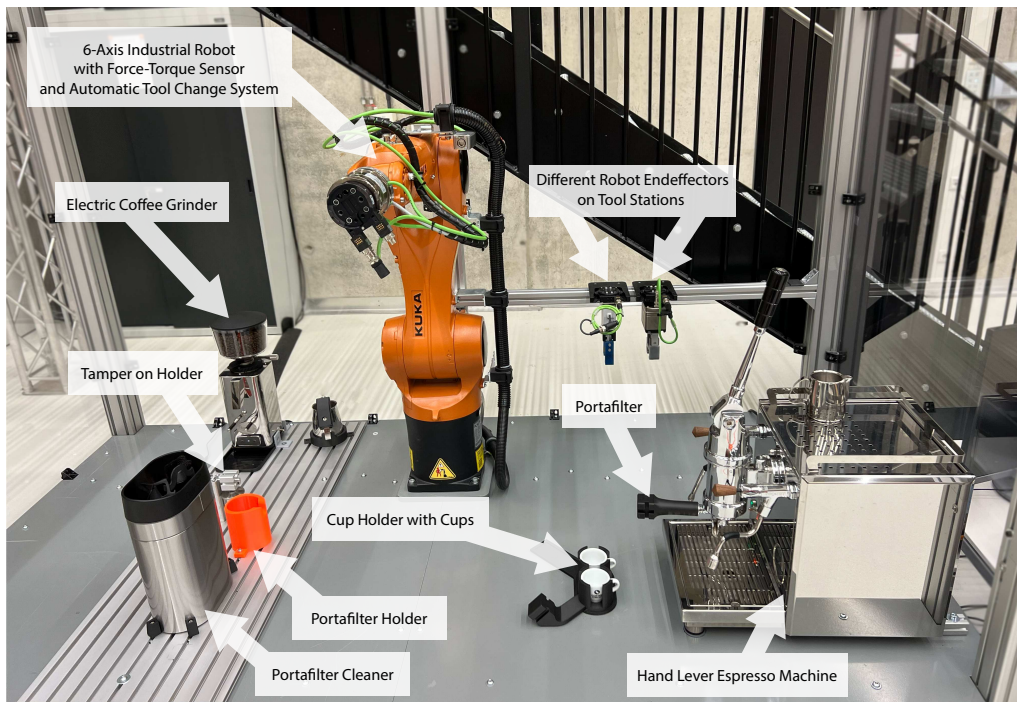
The automatic tool changer is located behind the force-torque sensor. This setup makes it possible to measure forces and torques independently of the tool since the load cell is not changed with the tool. One disadvantage, however, is that the structure between the force-torque sensor and the TCP can vary greatly, so the forces acting on the TCP must be calculated depending on the tool. For example, due to the law of levers, a force on a tool with a large lever can act as a torque on the sensor, which must be considered when using the sensor values for controlling the robot. As tool changing system the HTC 180 [196] of the company Stefan Holzer Feinmechanik was used. The automatic tool changer of the KR90 has several media feed-throughs for power, two network channels, compressed air, and other control signals to provide the different media for different tools. The tool changer can accommodate tool change plates that are equipped with media transfer elements. These tool change plates can be used to mount corresponding tools and supply them with power and data via the media feed-through system. Any tools, such as different grippers or screwdrivers, can be attached to the tool change plates.

Figure 10.3 shows the CP Lab system with the KR10 robot and a processing station. The KR10 robot is also equipped with an automatic tool changing system FWR50 [210] of the company Zimmer Group. Due to the size of the robot and the smaller tool changer, there are not as many media feed-throughs as with the KR90 tool changer. The different tools of the KR10 can therefore not be used on the KR90. The tool changer of the KR10 is only equipped with one feed-through media adapter for the ethernet network and one for IO-Link [84] components. IO-Link is a digital point-to-point interface for supplying small sensors and actuators over a five-conductor standard cable with power and data,

which is defined in the IEC 61131-9 [195] standard. For example, many grippers can be controlled via IO-Link. The processing station is equipped with tooling stations for the KR10 and an active clamping unit. The active clamping unit can be used to fix components of different sizes in order to be able to carry out processing or assembly activities on the component. The clamping unit can also be used as a transfer point to exchange components between the two robots. Therefore, it must be possible for both the KR10 and the KR90 to interact with the active clamping unit. Depending on the use case in the plant, there is space for additional clamping units on the processing station.

### 10.1.2 Robarista Cell

The second cell is a smaller robotic cell designed for the use case of making coffee with an industrial robot. The robot used is expected to take over all the tasks that a barista performs by preparing coffee. This is also the reason for the name Robarista (Robot-Barista). The goal is to use a commercially available hand lever portafilter espresso machine without modifications and an electric coffee grinder to make coffee. Figure 10.4 shows an illustration of the Robarista robot cell. The robot used is a KUKA KR6 R900 [107] industrial robot with a payload of 6 kg and a range of 0.9 m. The KR6



**Figure 10.4.** Robarista robot cell with an industrial robot and a hand lever portafilter espresso machine and the other equipment needed for the preparation of coffee with the robot

is equipped with a six-axis force-torque sensor FTM45 [19] of the company ATI to measure applied forces and torques on the tools. An automatic tool change system is attached to the torque sensor. An identical system was used here for the KR10 in the WiR innovation lab. The used espresso machine is a hand lever coffee machine Pro 800 [171], kindly provided by the company Profitec. In addition, the electrical coffee grinder ECM S-Automatik 64 [50] was also provided by Profitec. In addition, the robot cell has tooling stations with different gripper tools that are needed for the interactions with the coffee machine and the other components. In order to press the coffee powder in the portafilter with the robot, the so-called tamping process, there is a tray to lay down the portafilter. The tamper for pressing down the coffee powder also has a holder with a fixed position. Because of the difficulty of gripping the cups individually, a cup holder for two espresso cups has been designed that can be easily picked up with the gripper. The cup holder also has a fixation with a stable position in the cell. The portafilter, the tamper, and the cup holder are designed in such a way that they can all be gripped with a single gripper. Since it is difficult to tap out the portafilter with a robot, a portafilter cleaning machine was included in the cell. The cleaning device has two electrically driven brushes that can be used to remove the coffee residues from the portafilter after a coffee brewing session. Due to the cleaning components, making multiple espressi in a row is possible. All coffee components, the coffee machine, the grinder, and the portafilter cleaner, are not capable of communication. All these devices are used or triggered by passive interactions. The portafilter espresso machine is triggered to make coffee by pushing the lever. The coffee grinder is triggered by a switch, which starts the grinding process, and a timer in the grinder ends the grinding process. The active brushes of the portafilter cleaning machine are triggered by pushing on the brushes, if there is no more force on the brush, the rotation of the brushes stops.

## 10.2 Implementation of RealCaPP Concepts with Real Hardware Components

To evaluate the Plug & Produce concepts on real hardware, different end effectors and robots were made Plug & Produce capable. Therefore, AAS were developed for the various hardware components. Hardware drivers were written for the end effectors and robots, and interfaces to the Basic Skills were created. Since different distribution concepts are possible depending on the size of the end effector, the different prototypes with different distributions are explained. The prototypical implementation also revealed some problems that will be addressed. Problems encountered are, for example:

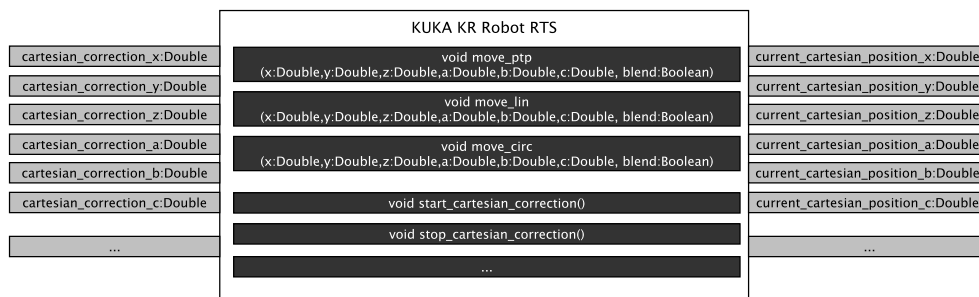
- Some associated hardware component controls cannot be attached to the end effector due to size and weight and must be attached to the robot or the robot base.
- Additional control components for making hardware RealCaPP capable are too large to be mounted on the end effector of small robots.
- Boot times of control components that are disconnected from the power supply and are only energized when added are high.

- Tools on the tool holder have no power and cannot communicate with the system.

### 10.2.1 KUKA KR Industrial Robots

All the robots used are KUKA industrial robots of the KR type and have a robot controller called KRC4 (KUKA Robot Controller 4) [102] with the same structure and the same interfaces for interacting with the robots. Classically, a proprietary robot programming language called KRL (KUKA Robot Language) [103] is used to program the KUKA KR robots. In this programming language, motion commands can be executed, calculations can be performed, additional integrated sensors can be queried, or additional actuators can be controlled, such as end effectors or linear axes. However, this programming language is not suitable for implementing complex Plug & Produce concepts, so an attempt was made to use external interfaces of the robot controller. KUKA offers two external interfaces over ethernet for this purpose. It is therefore possible to connect an additional external control component to the KUKA controller via an ethernet connection. The first ethernet interface is Ethernet KRL Interface (EKI) [101]. EKI is an add-on technology packet to exchange XML data between the robot controller and an external system via ethernet. The data is transmitted via the TCP/IP protocol. EKI is more suitable for transferring long-running and non-cyclical actions. These can be, for example, the sending of move commands or the setting of reference coordinate systems. Another interface that KUKA provides is the Robot Sensor Interface (RSI) [104]. RSI is an interface for real-time transmission of data between the robot and an external system. Therefore, it uses a real-time processing cycle of 4 ms. RSI also offers the possibility to directly influence the path of the robot by values of an external system in a 4 ms cycle. The main application of this interface is, for example, a force-controlled movement by an external force-torque sensor.

Figure 10.5 shows the implementation of the Basic Skill as an RTS for a KUKA KR robot. As already shown in Section 9.1 for the general case, the KUKA robot implements all ports and functions of the abstract industrial robot RTS. Due to the separation of the KUKA interfaces into cyclic data communication via RSI and acyclic communication

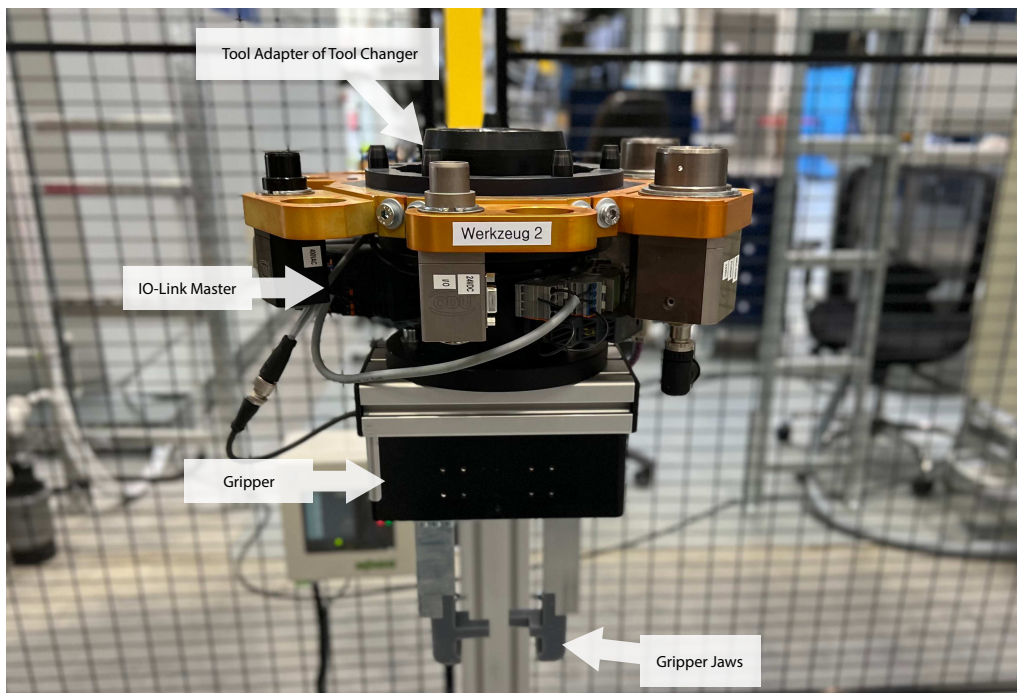


**Figure 10.5.** Abridged presentation of the Basic Skill of a KUKA KR robot as RTS implementation

via EKI, RSI was used to implement the in and out ports of the RTS and EKI was used to implement the functions of the RTS. Since RSI does not permanently supply data to the external system, the data transfer must be started acyclically accordingly. EKI commands are used to start the RSI interface. For example, the function of the RTS `start_cartesian_correction()` starts a cartesian correction of the end effector position based on the in port `cartesian_correction` values provided by RSI.

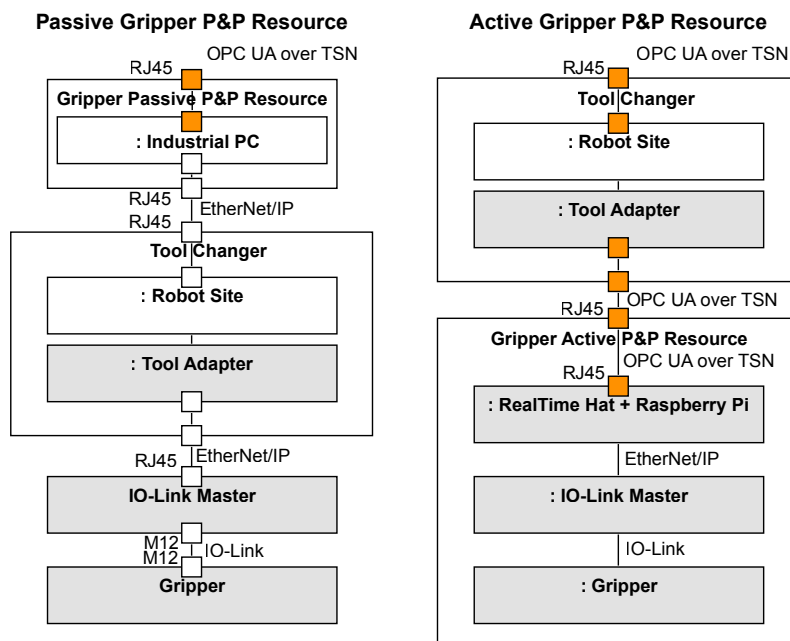
### 10.2.2 Gripper of the KR90: Zimmer Group GEH6180

The GEH6180 [211] two-jaw parallel gripper with long stroke from Zimmer Group was used as a gripper on the KR90. The gripper has a stroke per jaw of 80 mm, a gripping force between 150 N and 1800 N, and can be picked up via the KR90's automatic changing system. Figure 10.6 shows the assembly of the gripper with the tool adapter for the gripper. The gripper can be controlled via IO-Link, which is why an IO-Link master [151] from Pepperl+Fuchs is attached to the tool adapter. The IO-Link master provides the EtherNet/IP (EtherNet Industrial Protocol) fieldbus protocol [30] as the interface for controlling the gripper. The IO-Link interface can be used to set the desired positions for the open and closed states, the gripping forces, and gripping velocity. In addition, whether the gripping process has been carried out successfully and a component has been gripped is provided. Thus, the GEH6180 implements the following RealCaPP RTSs: Position Gripper RTS, Force Gripper RTS and Velocity Gripper RTS.

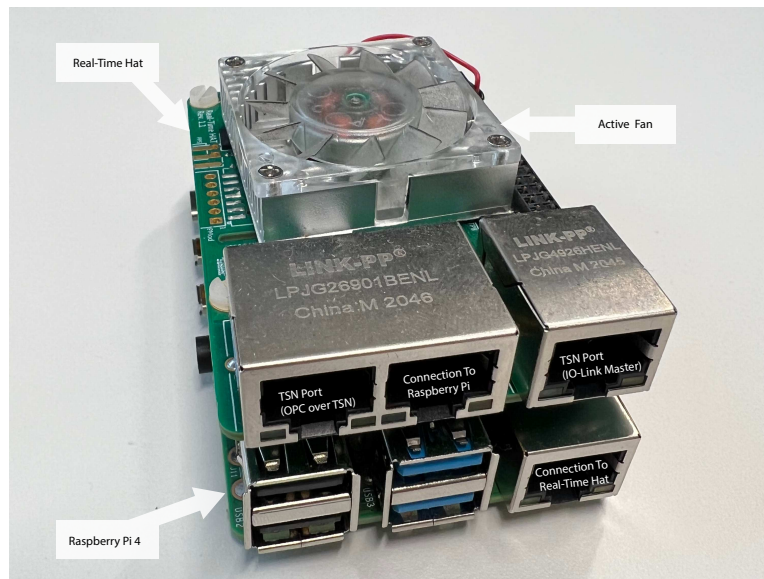


**Figure 10.6.** Structure of the GEH6180 gripper with tool adapter and IO-Link master for use on the KR90

Due to the high payload of the KR90 at 90 kg, different distribution patterns were carried out with the gripper. The gripper was developed as both, an active and passive Plug & Produce resource. Figure 10.7 shows the two UML composite structure diagrams of the passive and active realization. The realization of a passive Plug & Produce resource is shown on the left side. On the tool side (gray background), the gripper is connected to the IO-Link master via an IO-Link cable connection and provides the EtherNet/IP interface over the ethernet media feed-through of the tool-changing system. On the robot side, a passive Plug & Produce resource is realized with a hardware driver for controlling the gripper via EtherNet/IP. When the tool adapter of the gripper is connected to the robot, the passive Plug & Produce resource can communicate with the IO-Link master over the media feed-through of the tool changing system and send commands to the gripper. Only the passive Plug & Produce resource makes the gripper RealCaPP capable and provides the OPC UA over TSN interface. On the right side of Figure 10.7 the realization with an active Plug & Produce resource is shown. Here, the active Plug & Produce component encapsulates both the gripper and the IO-Link master and provides a RealCaPP capable interfaces via OPC UA and OPC UA over TSN for the gripper. For the implementation as an active Plug & Produce resource, with OPC UA and OPC UA over TSN interface, an additional control component is used. Since special network cards are required for TSN, a Raspberry Pi with a corresponding extension board was used. The Real-Time Hat of InnoRoute [79] is an expansion board for the single board



**Figure 10.7.** Composite structure diagram of the gripper for the active and passive resource implementation. Structures located on the tool side are grayed out. The RealCaPP capable OPC UA over TSN interfaces are marked orange.

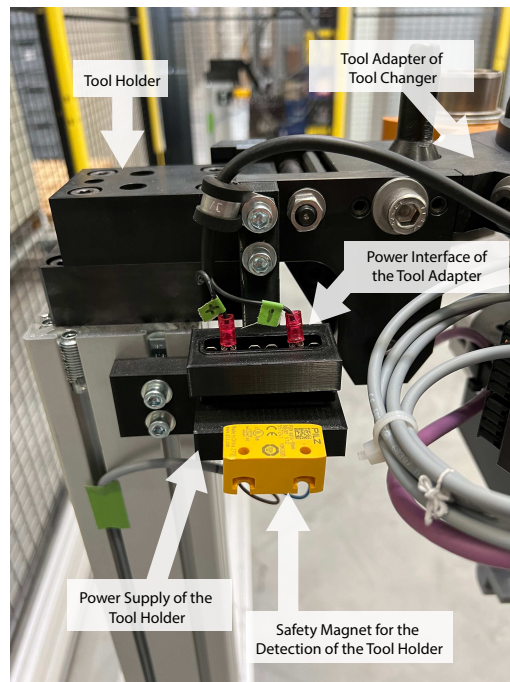


**Figure 10.8.** Raspberry Pi 4 with Real-Time Hat extension board for TSN capability on two ethernet ports

computer Raspberry Pi 4 [177] that extends the Raspberry Pi by precise network timing with hardware-time-stamping, needed for real-time protocols such as TSN. Figure 10.8 shows the used Raspberry Pi 4 with the Real-Time Hat mounted. The expansion board is connected to the Raspberry Pi via IOs and an ethernet connection. The Real-Time Hat has two TSN capable network ports available. One of these ports is used for the OPC UA and OPC UA over TSN interface. In this case, the other port is used to connect to the IO-Link master via Ethernet/IP. Thus, the combination of the gripper, the IO-Link master, and the Raspberry Pi with extension board forms an active gripper Plug & Produce resource.

As already mentioned, one problem is the boot time of control devices. A Raspberry Pi, for example, has an average boot time of over 30 seconds after switching on the power. A delay of 30 seconds after plugging in new resources is unreasonable, especially when changing tools. For this reason, a way was searched to minimize or eliminate the boot times of control components. To keep the control components running even after the tool has been set down in the tool holder, additional power supplies have been added to the tool holders. Figure 10.9 shows the realization of the tool holder power supply. Each tool holder of the KR90 is equipped with 24 V contacts. The tool adapter of the tool changing system can optionally be equipped with a power interface, which receives the power supply through spring-loaded contacts when placed in the tool holder. This means that the tool in the tool holder is also supplied with power, and control components remain running. When a tool is picked up, the tool is briefly supplied by the tool holder and the robot to prevent a voltage drop. When removing the tool, the power supply of the tool holder is interrupted. However, at this time, the

Problem with  
Boot Times



**Figure 10.9.** Structure of the tool holder power supply for the supply of the tool and its control devices

tool remains powered by the power supply of the robot. In addition, each tool holder is equipped with a safety magnet, which enables a magnetic sensor on the tool changer to detect whether the tool changer is located above a tool holder. The tool changer can only be opened and locked if it is located above a tool holder.

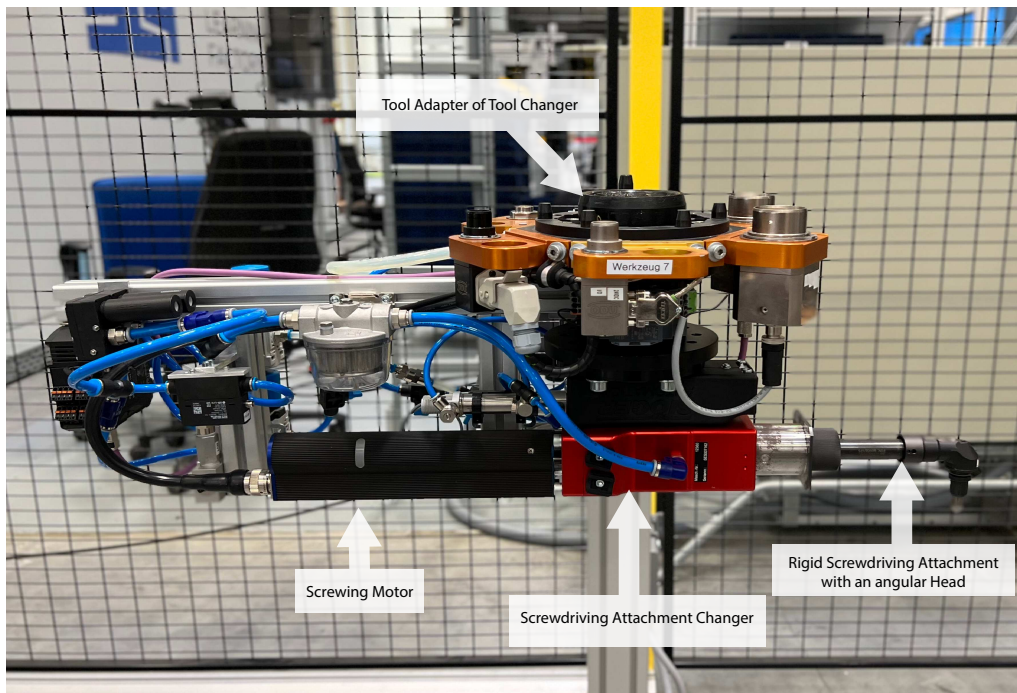
Problem with the Discoverability

One problem, especially with the active Plug & Produce realization, is the discoverability of the resource. It must be ensured that the gripper is connected to perform the self-introduction routine, but in this case, a connection is only possible when the robot picks up the tool. Therefore, at the start of the plant, either all tools must be picked up once, the self-description of the tool is added manually into the knowledge base or the components in the tool tray must use a different communication channel for the self-introduction, such as wireless networks.

### 10.2.3 Screwer of the KR90: Stoeger SPATZ 30

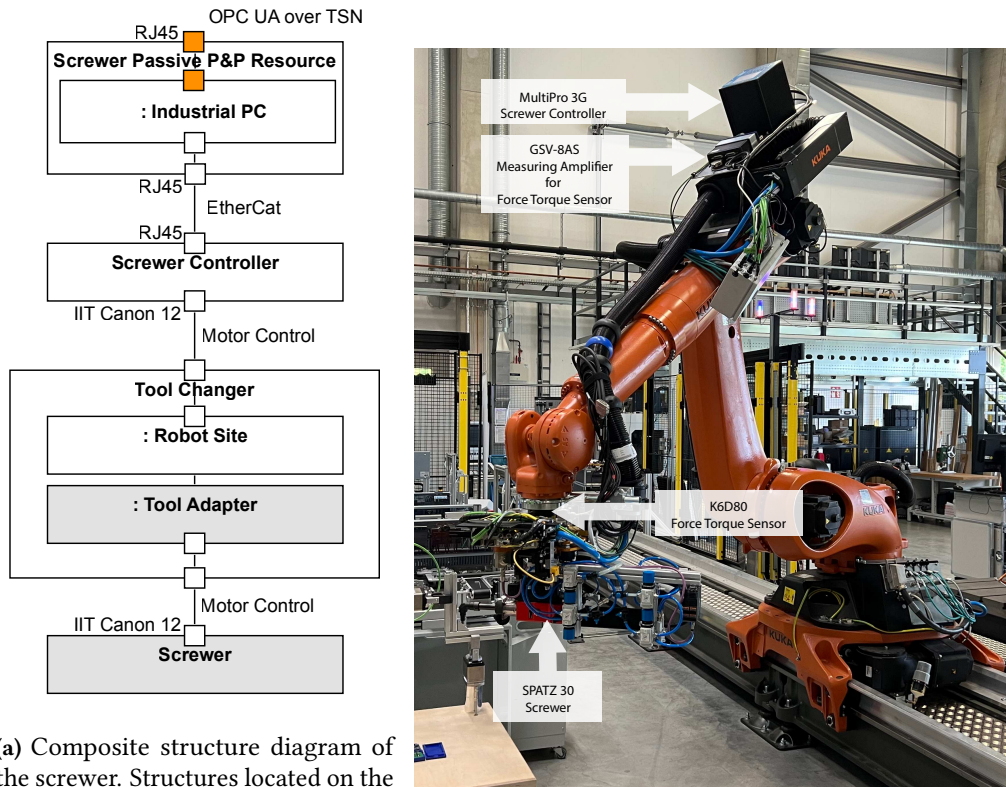
As screwdriver, the Stoeger SPATZ 30 [197] was used. The screwdriver has an integrated tool change system that allows the use of different screwdriver tools. For example, it is possible to change from a straight screw attachment with automatic screw tracking to a rigid screw attachment with an angular head. Figure 10.10 shows the screwdriver mounted to a tool adapter with inserted angular head screwdriver attachment. The blue hoses are pneumatic hoses and are used to track the screw through a pneumatic cylinder on the straight screwdriver attachment. With the straight screwdriver attachment, the





**Figure 10.10.** Structure of the SPATZ30 screwdriver with tool adapter for use on the KR90

screw is held in place by a vacuum, which is also generated pneumatically. In the angled attachment, the screw is held in place by a magnetic holder. The screwdriver is controlled via a DSM MultiPro3G [46] screwdriver control system. Due to the size of the control unit of 246 mm x 201 mm x 128 mm, it is unfortunately not possible to mount it on the tool adapter of the automatic tool changing system. The screwdriver control was mounted on the arm swing mounting plate of the robot as shown in Figure 10.11b. The controller has an EtherCat interface as a communication interface to start the tightening programs and to set possible parameters such as the direction of rotation, the tightening torque, or the tightening angle. Figure 10.11a shows the UML composite structure diagram of the screwdriver integration into the RealCaPP architecture. The screwdriver is implemented as a passive Plug & Produce resource. In this case, the screwdriver controller is permanently connected to the passive Plug & Produce resource over an EtherCAT connection and provides the RealCaPP capable interface over OPC UA over TSN. A proprietary cable and protocol for the motor control implements the connection of the screwdriver controller to the screwdriver. However, this connection can be disconnected. The screwdriver cable provides 4 phases for the screwdriver motor and eight wires for additional sensor data and monitoring functions of the screwdriver. This cable is connected to the tool side of the tool changer, and the tool plate on which the screwdriver is mounted provides media feed-throughs to pass through the wires to the screwdriver. On the tool adapter, after the wires have been fed through, the wires are fed back onto the screwdriver cable and connected to the screwdriver. If the screwdriver is disconnected from the controller,



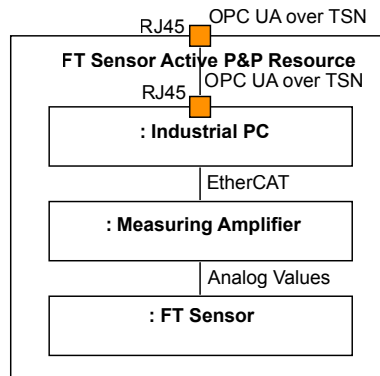
(a) Composite structure diagram of the screwdriver. Structures located on the tool side are grayed out. The Real-CaPP capable OPC UA over TSN interfaces are marked orange.

(b) Real construction of the robot with picked up screwdriver

**Figure 10.11.** Structure of the screwdriver with the integration into the RealCaPP architecture

an error is triggered in the controller, but this can be reset via the EtherCat interface when the screwdriver is available again. Thus, after picking up the screwdriver, it must be initialized. Since there are no active components on the tool side, the tool adapter does not have to be supplied with power in the tool holder.

The screwdriver does not have a screw feeder, so the screws must be picked up. For this purpose, a screw magazine was used in which the screws are arranged in a row. Since the screwdriver does not recognize whether a screw has been successfully picked up, the screw is passed through a photoelectric sensor after it has been picked up. In this way, it can be ensured that a screw has been picked up. This photoelectric sensor is connected to a digital input module and can therefore be easily integrated into the architecture.



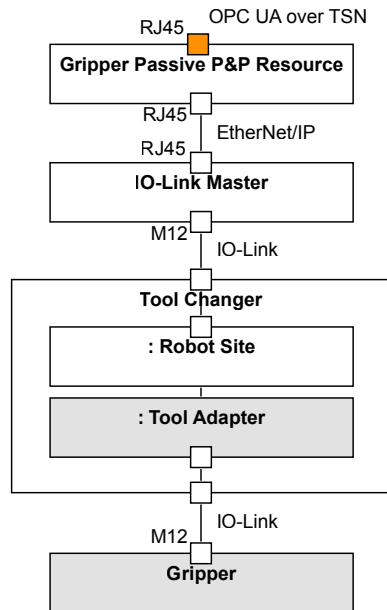
**Figure 10.12.** Composite structure diagram of the force-torque sensor of KR90. The RealCaPP capable OPC UA over TSN interfaces are marked orange.

#### 10.2.4 Force-Torque Sensor of the KR90: ME-Meßsysteme K6D80

As mentioned, the K6D80 6-axis force-torque sensor from ME-Meßsysteme is used for controlling and monitoring end-effector forces on the KR90. The sensor itself has a proprietary interface and transmits the analog force and torque measurement signals to an amplifier, where they are converted to a digital signal. Therefore, the eight-channel measuring amplifier GSV-8AS from ME-Meßsysteme was used. Due to the size of the measuring amplifier, it was mounted on the swing arm of the KR90, where the screwdriver controller is mounted (see Figure 10.11b). The amplifier provides an EtherCat interface to the digital sensor values for the three force and three torque values. An additional control component like an IPC or a Raspberry Pi with Real-Time Hat is connected to the EtherCat interface of the amplifier in order to make the sensor RealCaPP capable. Figure 10.12 shows the UML composite structure diagram for the force-torque sensor implemented as active Plug & Produce resource.

#### 10.2.5 Grippers of the KR10 and KR6

Since the KR10 in the WiR innovation lab cell and the KR6 in the Robarista cell have the same tool changer with identical media feed-throughs, they can also use the same tools. Three different grippers are available for the two robots. For the control of the grippers, IO-Link is used. Figure 10.13 shows the UML composite structure diagram for the grippers. Due to the size of the IO-Link master, it is not possible to mount the IO-Link master on the tool side of the small robots, as it is the case with the KR90. Therefore, it is only possible to integrate the grippers per passive Plug & Produce resource. The passive Plug & Produce resource is connected directly to the IO-Link Master via EtherNet/IP fieldbus. The IO-Link signal from the IO-Link master then goes to the IO-Link media feed-through of the tool changer. From there, it goes directly to the respective gripper via the tool adapter. IO-Link can also be disconnected and reconnected during runtime without any problems. Since the boot times for the IO-Link communication of the



**Figure 10.13.** Composite structure diagram of the grippers for the KR10 and KR6. Structures located on the tool side are grayed out. The RealCaPP capable OPC UA over TSN interfaces are marked orange.

grippers are under a second, the grippers do not have to be supplied with power in the tool holder.

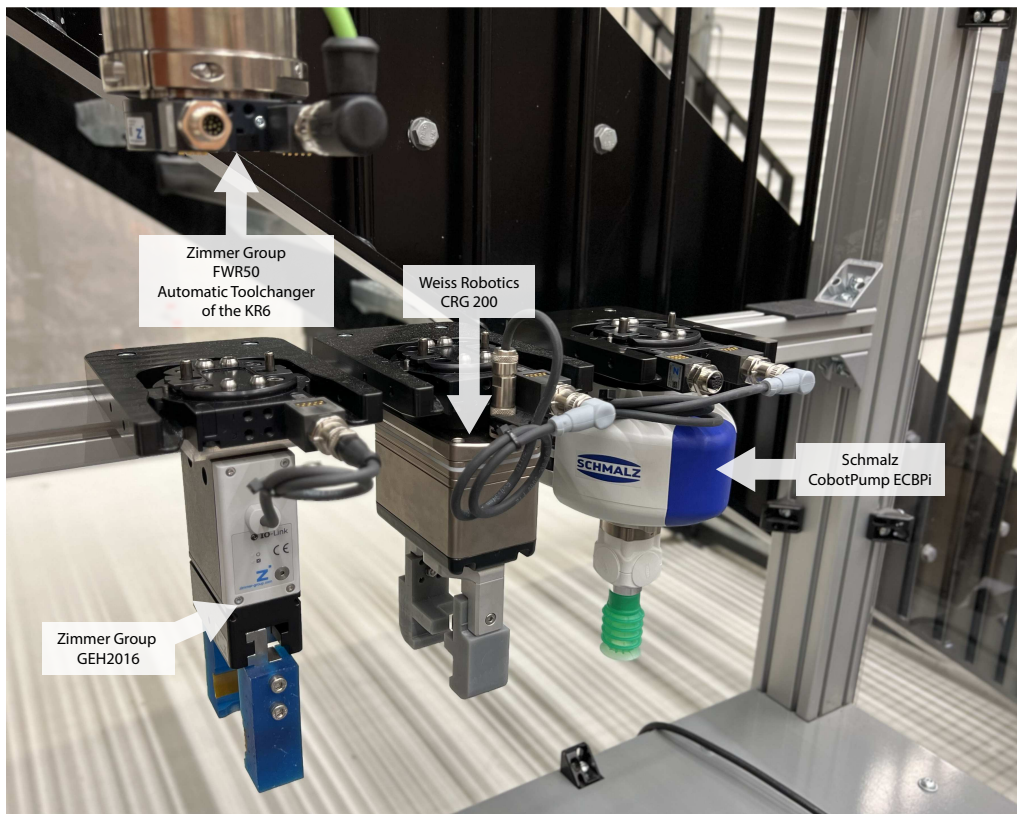
Figure 10.14 shows all three grippers with their tool change adapter in the tool holders of the Robarista cell.

### Zimmer Group GEP2016

One of the grippers used is the two-jaw parallel gripper GEP2016 [212] from Zimmer Group. The gripper has only a stroke per jaw of 16 mm, but a large adjustable gripping force between 150 N and 500 N. The position and force can be set via the IO-Link interface, and corresponding movement commands can be sent to the gripper. In addition, parameters are provided to indicate whether a grasp operation has been successfully executed and an object has been grabbed. It is possible to attach different gripper jaws to the gripper, which can be exchanged depending on the use case, but an automatic exchange of the gripper jaws is not possible.

### Weiss Robotics CRG200

The second gripper is also a two-jaw parallel gripper. The Weiss Robotics CRG200 [206] has a stroke per jaw of 42.5 mm but can only apply an adjustable gripping force between 75 N and 200 N. The IO-Link interface enables the configuration of the grip velocity, position, force and allows the gripper to receive corresponding movement instructions.



**Figure 10.14.** Grippers of the KR6 and KR10 in the Robarista cell

Here, too, it is possible to read out the success of a grip. The gripper jaws can also be changed manually on the CRG 200.

### **Schmalz CobotPump ECBPi**

The Schmalz CobotPump ECBPi [188] is a vacuum gripper with an integrated vacuum generator. The CobotPump can create a maximum vacuum of 750 mbar, has a maximum suction rate of 12 l/min, and an integrated ventilation valve for fast and precise depositing. Over the IO-Link interface, the gripper can be activated, deactivated, and open the valve for releasing parts. There are also parameters for setting the negative pressure and the air volume. The vacuum pump uses the pressure to detect whether an object is being held and can provide these values via the IO-Link interface. Several suction cups are available for the CobotPump. However, these can only be exchanged manually.

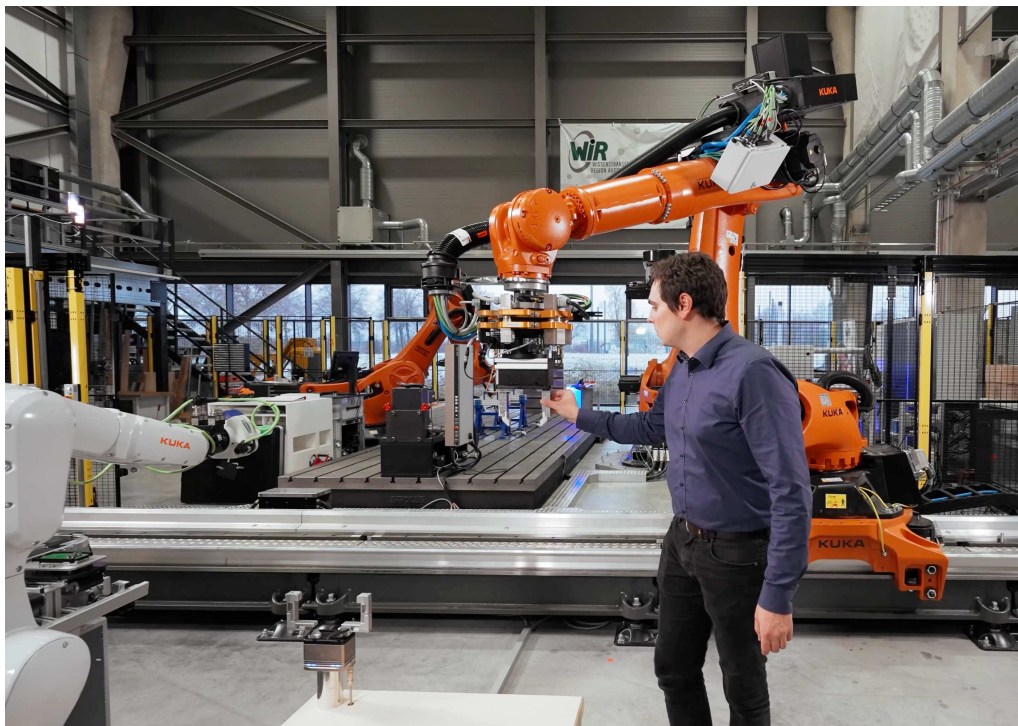
## **10.3 Hand Guiding of Industrial Robots**

Hand-guiding is a technique used in industrial robotics to guide a robot arm to a desired position physically. A special form of this is the force-controlled hand guiding. This involves using force sensing technology to guide the robot's movements based on the

force a human applies to the robot. Additional force sensors on the robot allow to measure the forces on the robot and can react accordingly. In this way, the robot can be moved by pulling or pushing the end effector. In the WiR innovation lab, the KR90's end effector is equipped with force sensors that can measure the amount of force being applied to the tool in various directions. These sensors provide real-time feedback about the forces being exerted on the end effector. As a human applies forces to the end effector, these are detected by the sensor. A control component processes the force data and translates it into movement commands for the robot. These generated commands must then be sent back to the robot, which has a movement of the robot as a result.

Figure 10.15 shows a photo where a person controls the KR90 of the WiR innovation lab by force-controlled hand guiding. This type of control requires hard real-time because the robot must be able to respond to the measured force values immediately. Delays or even gaps in the force values cause the robot to react delayed to the interactions or even cause the robot to jerk. This severely restricts the usability of the application and may even endanger humans, as they are in the direct vicinity of the robot. In the worst case, the robot moves toward the user and then no longer reacts to opposing forces. Even with a delay of only 100 ms at an end effector velocity of 0.2 m/s, the end effector would still travel a distance of 20 mm, which is a great distance with contact present.

For this case study, the deadline of the real-time system was set to 1 ms. However, since the RSI interface of the KUKA controller only accepts values at a maximum rate of 4 ms,



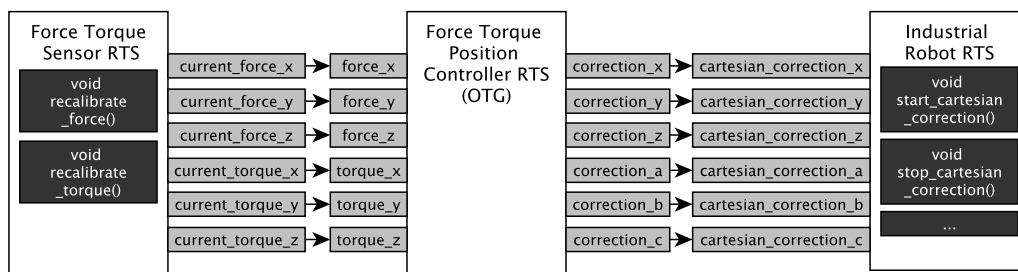
**Figure 10.15.** Force-controlled hand guiding of the KR90 in the WiR innovation lab

only every fourth value is sent to the robot controller. If the robot travels at an end effector velocity of 0.2 m/s, the end effector can only travel a distance of 0.8 mm in the 4 ms.

Figure 10.16 shows the RTS implementation of the force-torque controlled hand guiding. The Force-Torque Position Controller RTS is an abstract RTS that gets as input force values and torque values, applies a calculation routine to these inputs, and returns the results position correction values and orientation correction values as output. The calculation routine used for the RTS implementation in this case study is an Online Trajectory Generator (OTG). An OTG is a trajectory generator that continuously uses incoming data to instantaneously generate the next point of a robot trajectory, taking into account various constraints such as velocity limits, acceleration limits, or jerk limits. The OTG algorithm of a nonlinear filter described by Biagiotti and Melchiorri [25, pp. 209] was used as OTG implementation of the RTS implementation. The OTG receives the force value and torque value as input, calculates a resulting direction in which the end effector should move, and calculates a position and orientation correction, considering the constraints. The mentioned OTC algorithm considers the velocity limits, acceleration limits, and jerk limits of a robot. Particularly with regard to direct cooperation between the human and the robot, it is essential to be able to set the maximum possible end effector velocity in the application.

Due to the general description of the task with the abstract RTS implementations, it can be executed with different hardware since the hardware only has to imply the required abstract RTSs. The hardware required to execute the use case is a RealCaPP capable force-torque sensor and a RealCaPP capable industrial robot. This generalization of the task makes it possible to run the use case in the WiR innovation lab as well as in the Robarista cell without having to adapt the programming. In the WiR cell, the K6D80 is used as a force-torque sensor in combination with the KR90 industrial robot. In the Robarista cell, the FTM45 force-torque sensor is used together with the KR6 industrial robot.

Thus, it could be shown for this case study that **real-time critical application can be executed on different systems without adapting the software.**



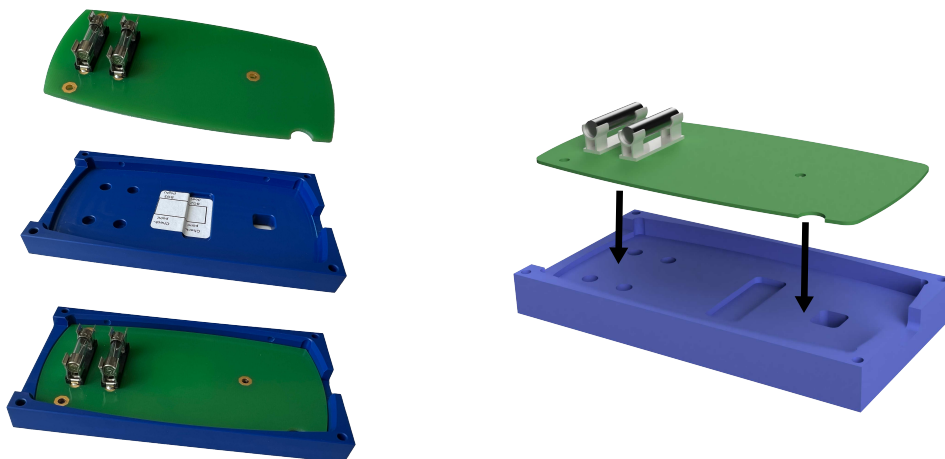
**Figure 10.16.** RTS implementation of the force-torque controlled hand guiding. Due to size and readability, the data types were omitted.

## 10.4 Assembly of a Circuit Board Component

As a second case study for the evaluation of the RealCaPP architecture, a handling process was examined in which a circuit board is inserted into a housing by a robot. A frequently used robot handling skill is pick and place, where an object is picked up and placed at a defined position. Because of the high degree of accuracy a robot has, robots can precisely place objects in a given position at high velocities. This leads to a high production throughput.

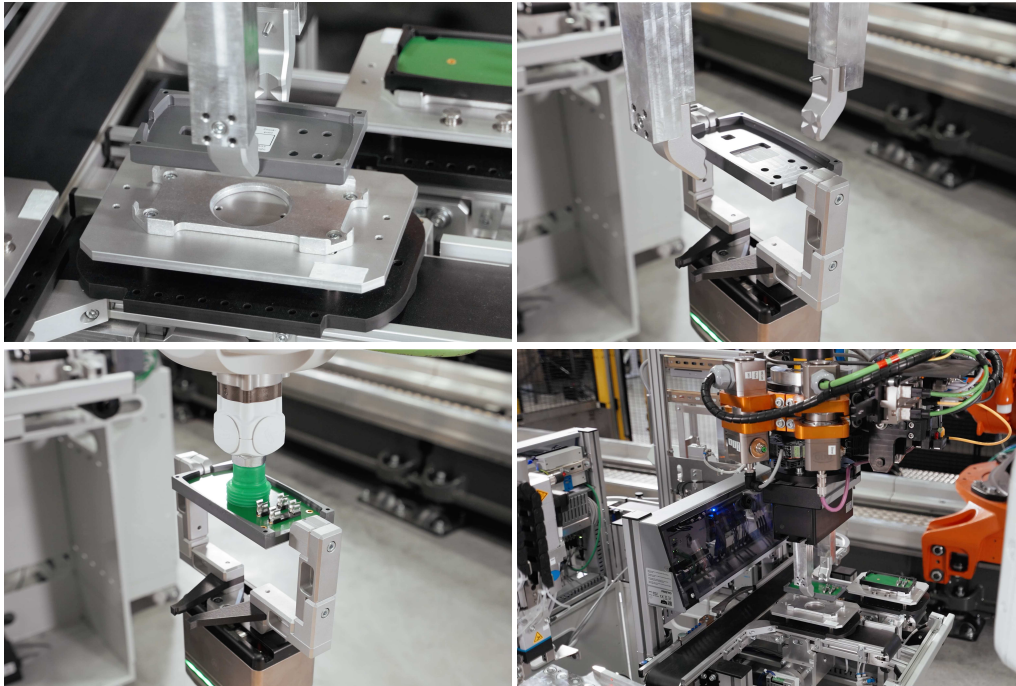
Figure 10.17 shows the component to be produced. On the left side of the figure pictures of the parts and the end product are shown, while the right side shows the 3D CAD drawing of the product to producing. The entire handling process consists of four subtasks, as shown in Figure 10.18. First of all, the housing must be picked up from a component carrier, which is provided by an additional production line via a conveyor belt (top left). Afterwards, the housing is placed in an active clamping device at the robot's processing station (top right). Then, the circuit board must be picked up from a carrier and inserted into the clamped housing (bottom left). Finally, the housing with the inserted circuit board is unclamped and placed back on the product carrier of the production line (bottom right). All of these subtasks do not require hard real-time. Nevertheless, an attempt is made to maintain the hard real-time criteria. The manufacturing process basically consists of three pick and place tasks with different parts and different positions. There is a small peculiarity in the transfer between the gripper and the clamping device (top right). In addition to the pick and place skill, it is necessary to wait for the active clamping device to close.

The pick and place skill was realized as a Composed Skill, as mentioned in several examples in previous chapters. This Composed Skill is implemented by the encapsulated



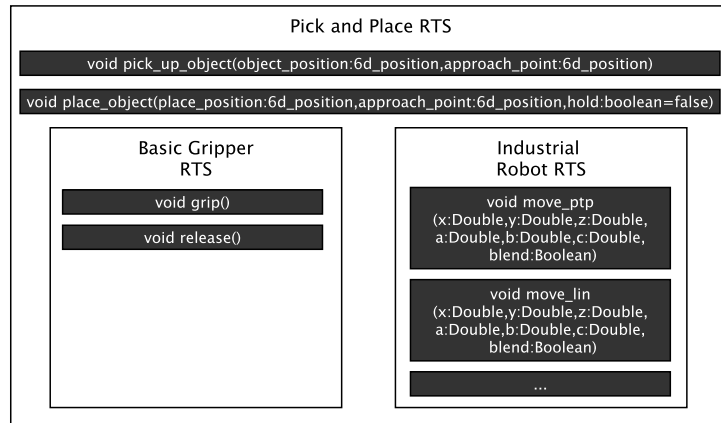
**Figure 10.17.** Circuit board component. The left side shows from top to bottom shows the circuit board, the housing, and the assembled component. The right side shows the 3D CAD drawing





**Figure 10.18.** Process steps for the assembly of the circuit board component. From left to right: Picking up the housing. Placing the housing in the active clamping device. Placing the circuit board in the housing. Depositing the fully assembled component.

Pick and Place RTS as shown in Figure 10.19. The Pick and Place RTS is dependent on the Basic Gripper RTS and the Industrial Robot RTS. The Basic Gripper RTS includes the skills for gripping and releasing an object, and the Industrial Robot RTS provides the skill of moving to a position. The Pick and Place RTS provides two functions. A function to pick up a component at a specific position with a defined approach point (see `void pick_up_object(...)`). Moreover, a function to deposit a picked up object at a defined position via a set approach point (see `void place_object(...)`). For the place function of objects, there is an additional boolean parameter `hold`. It specifies whether the object should be released after the placement or whether the object should be held in position. For example, the `hold` parameter is used for transferring the housing in the active clamping device. The robot holds the housing in position, the active clamping device is activated and then the gripper can be opened and the robot can move away. The approach points are points in space near the destination where the object should be picked up or placed. The pick-up point or the deposit point is always reached by a linear movement from the approach position point. These additional points avoid collisions when placing or picking up. For later iterations, it is conceivable that these points will be calculated and tested for collisions through appropriate simulations. The two functions of the Pick and Place RTS in turn access the functions of the sub-RTSs. For example, the `move_ptp` and `move_lin` functions of



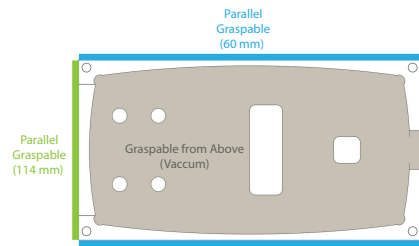
**Figure 10.19.** Abstract encapsulated RTS of the Composed Skill pick and place. The sub-RTSs were limited to the essential functions for the sake of clarity.

the Industrial Robot RTS and the release function of the Basic Gripper RTS are combined to create the function `place_object`.

Due to the abstract implementation without concrete hardware resources, the skill pick and place can be executed with several hardware configurations. So only a system configuration is needed, containing a robot implementing the Industrial Robot RTS and a gripper implementing the Basic Gripper RTS. Due to the semantic connection between the RTSs and the description of the skills in the ontology, it is possible to use the approach presented in Section 6.5 to search for suitable hardware configurations. Based on the components available in the WiR innovation lab, the following configurations result in the configuration request for the skill pick and place without further constraints:

- KR90 with long stroke parallel gripper GEH6180
- KR10 with parallel gripper GEP2016
- KR10 with parallel gripper CRG200
- KR10 with vacuum gripper CobotPump

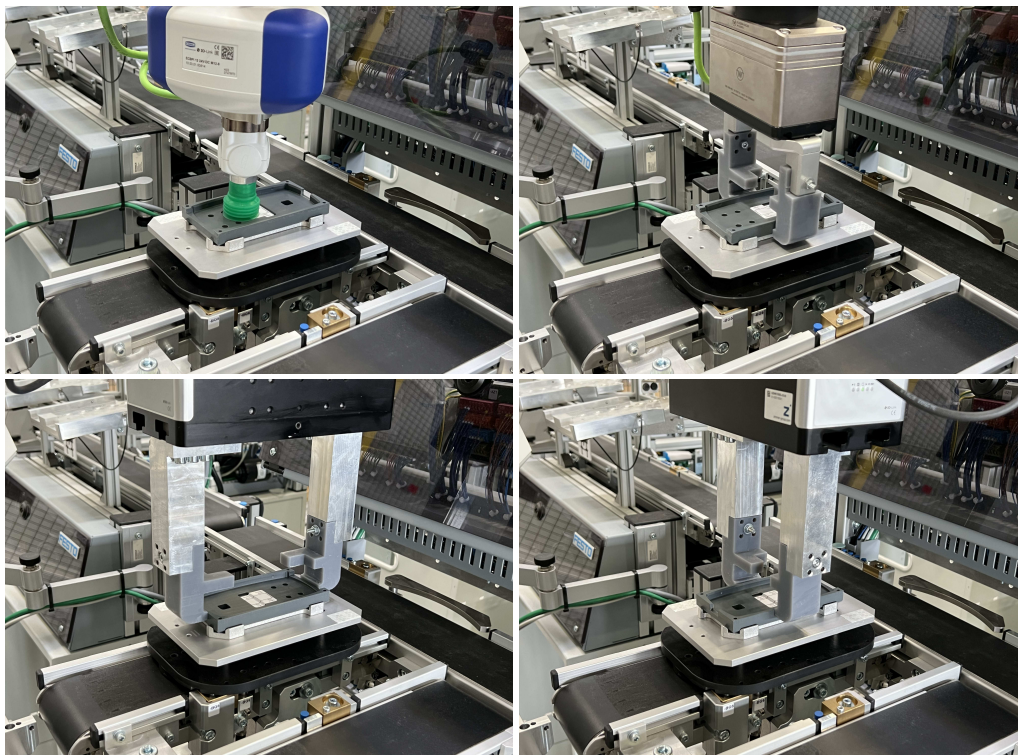
Additional constraints for the query can be defined via the description of the part and the annotation of the possible gripping positions. Figure 10.20 shows the gripping annotation for the housing. The housing part can either be gripped with a parallel gripper that can grip components with a width of 60 mm via the short side (blue sides) or with a gripper that can grip components with a width of 114 mm via the wide side (green sides). Alternatively, a vacuum gripper can be used to grip the component from above (gray area). Figure 10.21 shows the resulting pick and place system configuration for the housing part. Unfortunately, the configuration with the KR10 and the parallel gripper GEP2016 does not meet any of the requirements since components with a maximum width of 40 mm can be gripped. One possible configuration is gripping from above by the



**Figure 10.20.** Top view CAD drawing of the housing part with the annotated gripping options

KR10 with the CobotPump vacuum gripper (top left). The short side of the component can be gripped both with the CRG200 parallel gripper on the KR10 (top right) and with the GEH6180 parallel gripper on the KR90 (bottom right). Only the GEH6180 parallel gripper on the KR90 is suitable for the wide side (bottom left).

The system offers two options for the selection of configurations. Either the programmer can select a possible configuration, or the system decides automatically on a possible con-



**Figure 10.21.** Possible hardware configurations for handling the housing part

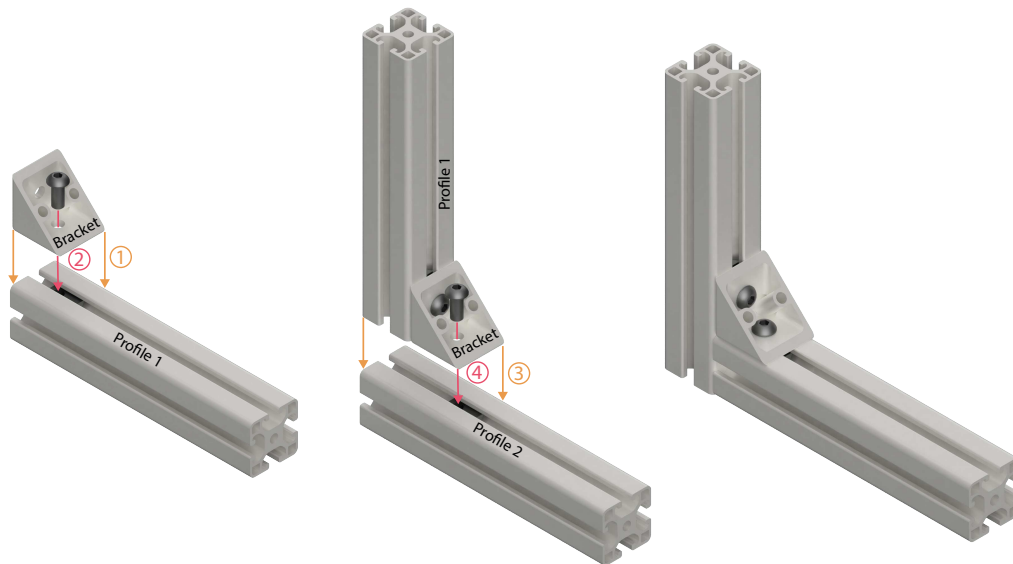
figuration. For this, it is recommended that the programmer specifies an order function in addition to the part constraints in order to find the most suitable configuration.

Thus, it could be shown for this case study that **tasks can be developed independently of the hardware, the tasks are thus easily reusable, and the RealCaPP architecture suggests possible hardware configurations for the fulfillment of a respective task.**

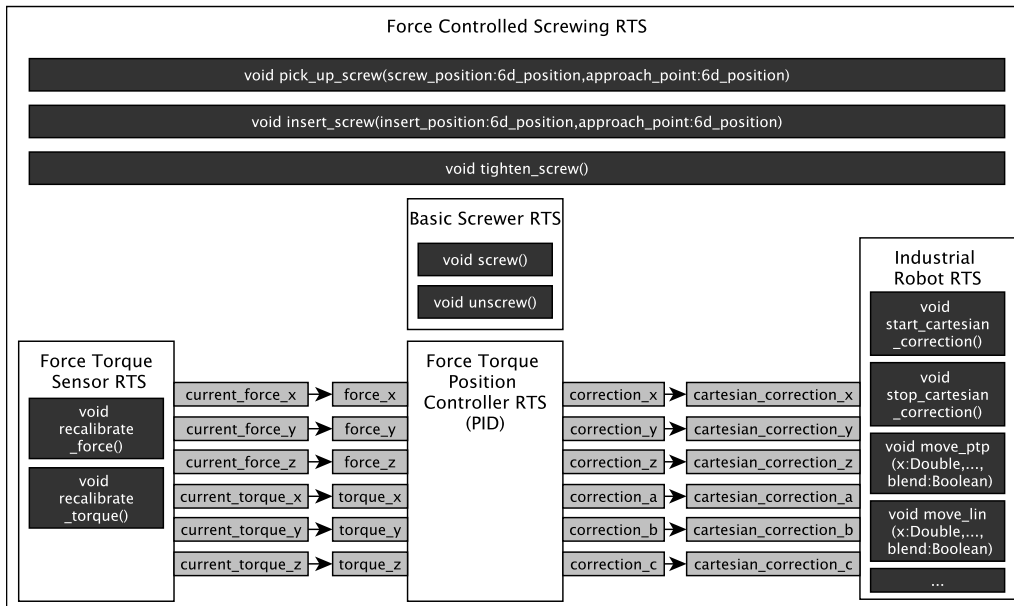
## 10.5 Assembly of Aluminium Structures

Since robotics applications do not consist exclusively of pick and place sequences, the assembly of aluminum structures was evaluated as an additional industrial case study. The component to be produced is an L-structure consisting of two aluminum profiles connected by a bracket. Figure 10.22 shows the process steps leading to the finished component. First, a bracket is placed on one of the aluminum profiles (①) and fixed with a screw (②). The finished intermediate product is then rotated and placed on the second aluminum profile (③). Both parts are then fastened with a screw (④). Through these process steps, the desired L-structure is created.

If this L-structure is to be produced in a robot cell, these individual process steps must be broken down into robot skills again. Several subtasks that can be implemented using the pick and place skill can be found here as well. For example, the placement of a bracket can be realized by a pick and place skill. In addition, a skill is needed to pick up, insert and tighten screws at a given position. For this reason, an RTS was developed to realize the skill of picking up, inserting, and tightening screws. These Composed Skills consist



**Figure 10.22.** Construction plan of the aluminum structure consisting of two aluminum profiles that are screwed together via a bracket



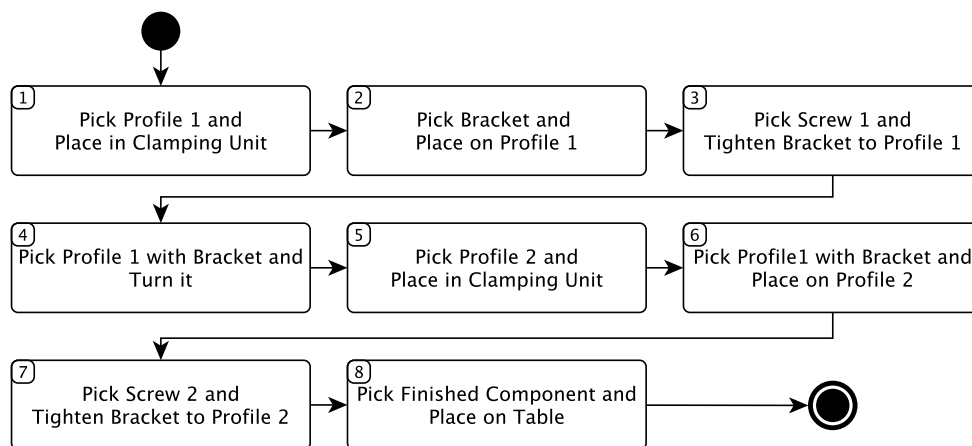
**Figure 10.23.** Abstract encapsulated RTS of the Composed Skill force controlled screwing. The sub-RTSs were limited to the essential functions for the sake of clarity.

of the Basic Skills move to position of a robot and the Basic Skills screw in and screw out of a screwdriver. The problem that remains is the tracking of the screw, either this can be solved by an automatic screw tracking system, but this is not possible with this component due to the accessibility of the screw holes. In this case study, a force-controlled screwing process is therefore implemented. This means that, depending on the forces applied to the screwdriver, it is detected whether there is contact with the screw hole, and the movement of the screw during tightening is determined by the forces measured. For this purpose, a force-torque sensor is used to measure the forces on the tooling. Figure 10.23 shows the abstract Force Controlled Screwing RTS. Force Controlled Screwing RTS is dependent on the Force-Torque Sensor RTS, the Basic Screwdriver RTS, the Force Torque Position Controller RTS and the Industrial Robot RTS. In addition, the RTS contains three functions. With the function `pick_up_screw(...)`, a screw can be picked up at a given position via a defined approach point. The second function is to position the screw at a defined position (`insert_screw(...)`), and the third function takes over the tightening of the screw (`tighten_screw()`) in a force-controlled manner. This again provides a hardware independent service description that can be used for the development of production applications. As a Force-Torque Position Controller RTS, the OTG used for hand guiding could be used again, but this is only suitable to a limited extent for contact driving and holding forces. Here, a classic Proportional-Integral-Derivative (PID) controller [181, pp. 11] is better suited, as the controller can be better parameterized for the specific use case. For screwing, a constant force is tried to be kept on the screw. In the use case shown, an attempt is made to maintain a force of 5 N on the screwhead, even if the screw moves as it is

tightened. Since the screw stops abruptly after being screwed in, the PID controller must react quickly. Therefore, the complete interaction between the screwer control, the force measurement, the position calculation, and the robot control must take place in real-time, with a hard deadline of 4 ms (minimal cycle time of the robot). Delays in any part of the force-controlled screwing process can lead to the destruction of the parts or the screwdriver. Again, a brief example is given of why real-time is necessary for this process. M8 screws with a thread pitch of 1.25 mm were used for the use case. The screwdriver is operated at a velocity of 480 rotations per minute (8 rotations per second). This results in a tool velocity of the robot of 0.01 m/s. Delays of one tenth of a second (100 ms) already lead to a movement of 1 mm. Which is a lot of way, due to the fact that there is already contact between the screwdriver and the screw.

Now that all the required skills are available, it is possible to implement the application in principle. Figure 10.24 shows the UML activity diagram for the complete production run of the aluminum structure. The activities were described from the point of view of the process developer, who only has the boundary conditions that components can only be assembled and screwed together in a clamping device and that screwing can only take place in a position where the screw points downwards. The individual activities have been numbered to make them easier to reference for the following description.

All this activities can be implemented by Application Servicess (ASs) as shown in Section 8.4.4. For this purpose, preconditions and postconditions must be defined for each AS. If a fixed sequence is used, as shown in this use case, it is possible to specify the predecessor AS as a precondition and to set the last executed AS as a postcondition after execution. In addition, it is also possible to specify which configuration is required for a corresponding activity, so that a needed toolchange is automatically performed before execution. An example for the precondition of activity [2] is shown in Listing 10.1. As a



**Figure 10.24.** Activity diagram for the processing steps for the assembly of the aluminum L-structure

---

```

1 ASK{
2   CurrentState hasStateNumber 1.           → Check current state
3
4   ?robot isA Robot.
5   ?gripper isA Gripper.
6   ?robot isConnectedToResource ?gripper   } A suitable hardware configuration must be found
7   ?gripper hasJaws ?jaws.
8   ?jaws canGrip Bracket.
9
10  ClampingUnit hasElement Profile1.       → Profile 1 must be in the clamping unit
11 }

```

---

**Listing 10.1.** SPARQL ASK query for checking whether all preconditions for the activity [2](#) (Pick Bracket and Place on Profile 1) are fulfilled

precondition, activity [1](#) must have been executed. There must be a suitable gripper hardware configuration that can grip a bracket and the profile 1 must be hold by the clamping device. Finally, the postcondition for this activity is that the CurrentState is 2 and the bracket is on profile 1.

The activities [1](#), [2](#), [5](#), [6](#) and [8](#) can be performed by simple pick and place skills. A suitable configuration for the execution of these tasks is the KR10 with the CRG200 parallel gripper. This configuration allows profiles and brackets to be picked up and placed. Another possible configuration would be the KR90 together with the GEH6180 parallel gripper. However, this configuration was not used because the KR90 is the only robot in the system that can handle the screwdriver. Therefore, the end effector would often have to be changed which requires a lot of time and can be avoided. For activities [3](#) and [7](#), which involve screwing, the only possible configuration is the KR90 with SPATZ 30 screwdriver with angle head. The parts to be tightened must be held in position, so the KR10, together with the CRG200 parallel gripper, must act as a part holder. Since the parts are already placed with the KR10 with the CRG200, the placing position can be maintained for holding the part. The placing function of the Pick and Place RTS is thus executed with hold parameter set true. If the holding is to be released, the gripper can simply be opened by the open function of the gripper.

Thus, all activities except for activity [4](#) can be executed. The problem with the activity [4](#) is that a robot can pick up and rotate the part, but the robot can only pick it up from above or from the side, which causes the screw holes to be covered by the gripper jaws. For this reason, the activity was realized by a transfer between two robots. The KR90 must be equipped with a gripper for this purpose. After changing the tool, the KR90 with the GEH6180 parallel gripper takes profile 1 with the bracket out of the clamping unit, rotates it, and holds it in front of the other robot. Afterward, the KR10 with the CRG200 parallel gripper can continue with activity [5](#). For the realization of activity [6](#), the KR10 can then grip the held part with with the CRG200 gripper. The GEH6180 of the KR90 is then opened, and the KR10 with the CRG200 can place the

part. The KR90 can then change its tool back to the screwdriver and then perform the activity [7]. Consequently, the following parallel sequence, as shown in the UML activity diagram (see Figure 10.25), results for the two robots. A cooperation area was defined to parallelize the tasks. Only one robot is allowed to move in the cooperation area simultaneously. Therefore, a mutex was introduced, which can lock the cooperation area. Therefore, all tasks of the robots that do not take place in the cooperation area can be executed independently of each other.

Figures 10.26 - 10.28 show images from the WiR innovation lab from the assembly of the aluminum structure. Figure 10.26 demonstrates the implementation of activity [6a] in which the KR10 picks the parts held by the KR90. Subsequently, the KR90 releases the component, leaves the common work area, and releases it. Figure 10.27 shows the

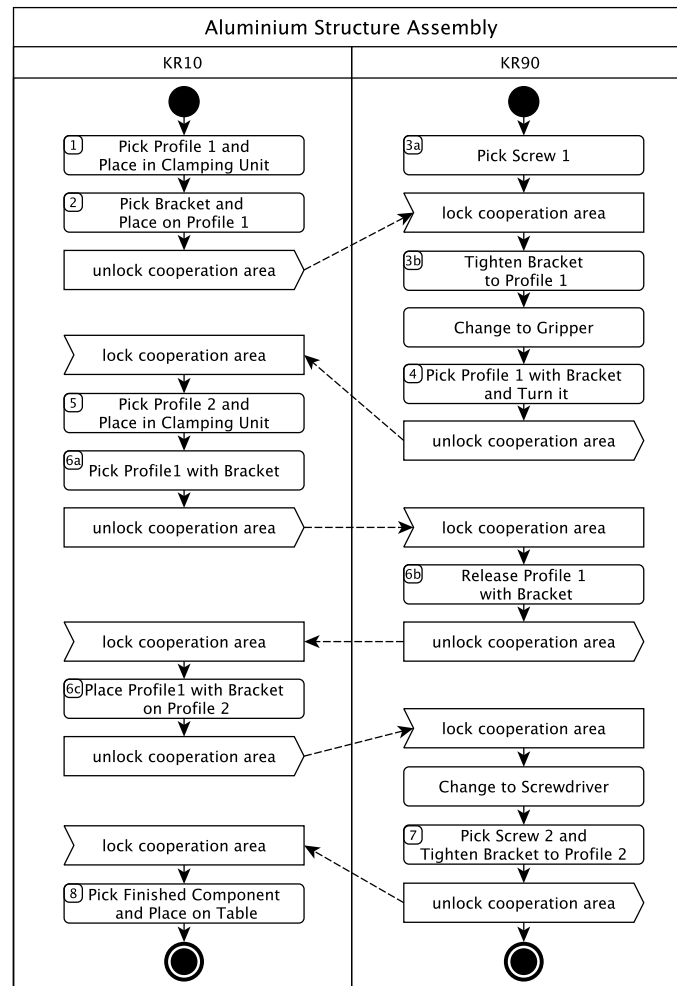


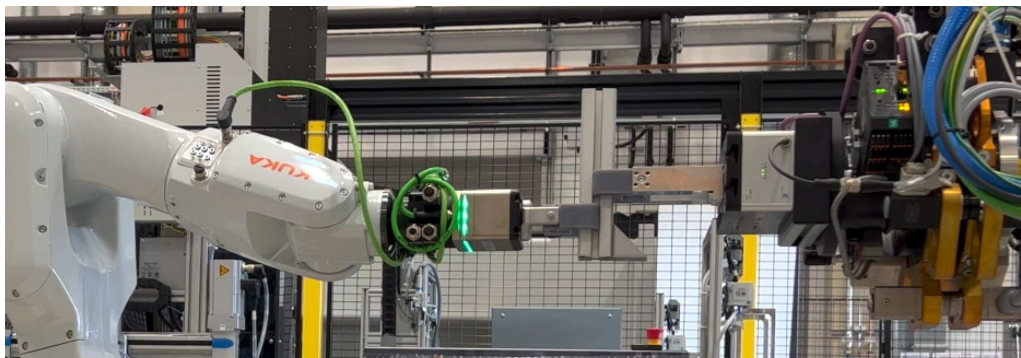
Figure 10.25. Activity diagram of the assembly process of the aluminum structure distributed between the two robot resources



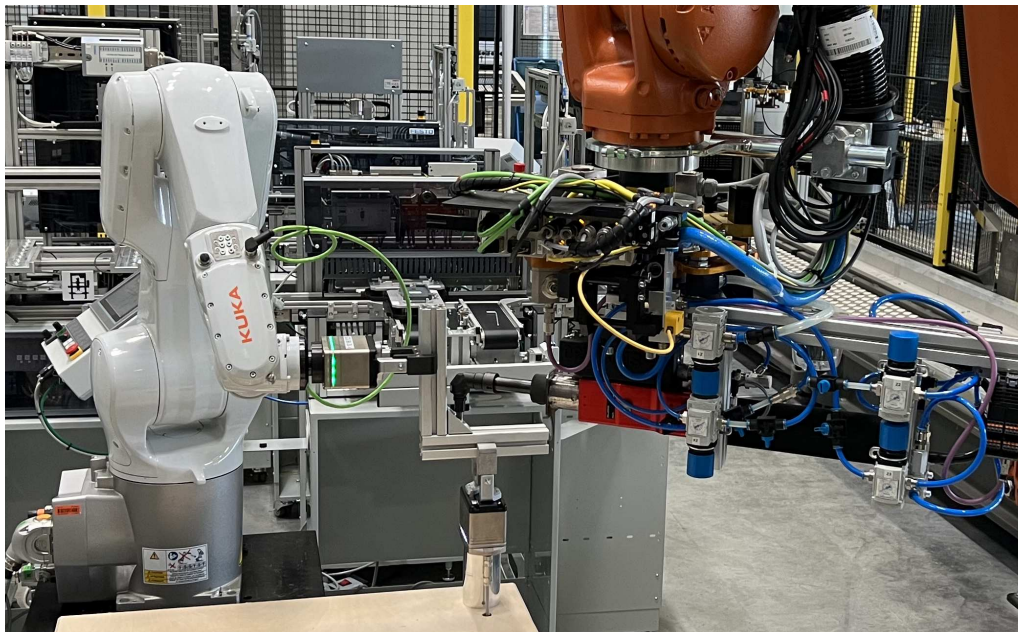
subsequent screwing of the two profiles (Activity [7](#)) and Figure 10.28 finally shows how the finished component is placed on the table (Activity [8](#)).

Another important aspect of the RealCaPP architecture is the distribution of the execution. Since the distribution of the parallel tasks of the two robots does not have to run in real-time, the real-time critical task of force-controlled screwing was used as a use case to show the distribution of real-time critical processes to different computing units. Since the process consists of three main parts, namely the measurement of sensor values, the processing of these measured values, and the writing of position values to the robot, three distributions are examined. In the first case, all skills were executed on one computer. In the second case, the sensor values were measured on one computing unit, and another computing unit calculated the position and wrote it to the robot. In the third case, each of the three parts was executed on a separate execution node. For all three distributions, the deadline of 4 ms was always kept. The exact execution times are mentioned in detail in Chapter 11.

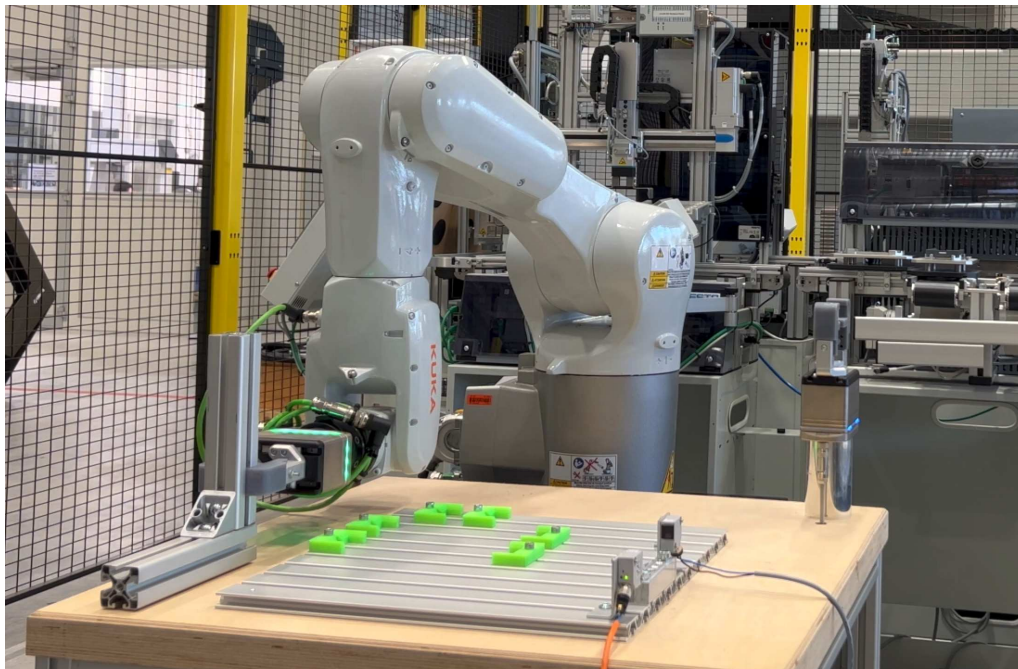
Thus, it could be shown for this case study that **dependencies between several parallel running tasks can be mapped. Robotic systems can change their skills through reconfiguration (tool changing) at runtime. The execution of real-time critical tasks can be distributed to several execution nodes without losing real-time.**



**Figure 10.26.** Overhanding of profile 1 with the bracket from the KR90 to the KR10



**Figure 10.27.** Tightening of the profile 1 with bracket onto profile 2 by the KR90, while the KR10 is holding the profile 1 with the bracket



**Figure 10.28.** KR10 placing the fully assembled aluminum structure on the table

## 10.6 Robarista: A Robot Making Coffee

In contrast to the previous case studies, the last case study focuses on a non-industrial application area. In this scenario, the process is to make coffee with a portafilter handlever espresso machine in the Robarista cell (see Section 10.1.2). An industrial robot is supposed to take over all the handling steps that a barista would typically perform. Since making coffee consists of several steps, this process is particularly well suited for evaluating the orchestration of services. In addition, the reusability of already developed services can be evaluated for a completely new use case.

First, the process of making coffee is broken down into its individual steps:

- Unclamp portafilter from the coffee machine
- Grind coffee into the portafilter with the electric coffee grinder
- Insert portafilter into tamp station
- Tamp coffee flour with the tamper
- Clamp portafilter into the coffee machine
- Place coffee cups under the outlet of the coffee machine
- Pull the lever of the coffee machine
- Serve the finished coffee

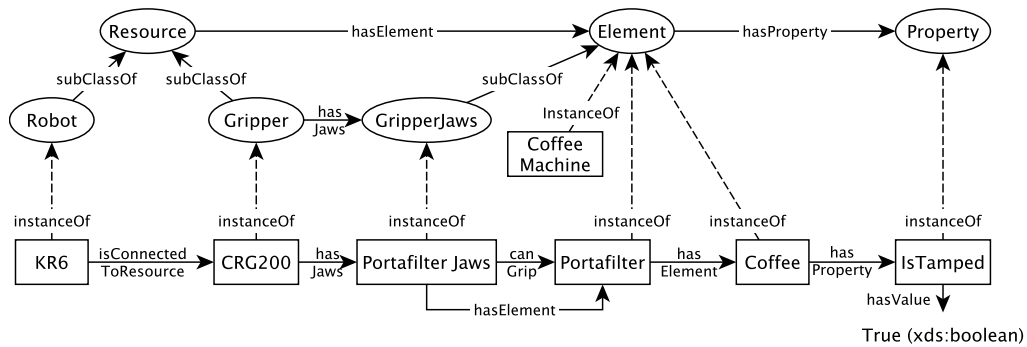
Subsequently, all these steps were implemented as application services (AS) as shown in Section 8.4.4. This means that the individual process steps were broken down into executable applications and for each application, the preconditions and postconditions were specified. In order to avoid having to list all the process steps with their preconditions and postconditions, everything is explained in the following using the task Clamp portafilter into the coffee machine as an example. Before the portafilter can be clamped into the coffee machine, the following preconditions must be fulfilled:

- There must be a robot with a gripper mounted and suitable gripper jaws to grasp the portafilter.
- The gripper must hold a portafilter filled with coffee and the coffee must be tamped.
- The coffee machine cannot have a portafilter inserted.
- The coffee cups are not placed under the outlet of the coffee machine.

If all these preconditions are met, the task can be executed. After successful execution, the postconditions are set. For the task of clamping the portafilter into the coffee machine, the following postconditions would have been set:

- The gripper does not hold a portafilter.
- The coffee machine has a portafilter inserted.

The preconditions are formulated as SPARQL ASK queries, and the postconditions are formulated as scripts that can make changes to an ontology. Figure 10.29 shows a snapshot of the ontology in which all preconditions for the execution of the insertion of



**Figure 10.29.** Snapshot of the ontology showing the system state in which all the preconditions for clamping the portafilter in the coffee machine are met. Classes are represented as ellipses: Instances are represented as rectangles: Relations between classes are shown as arrows:  $\rightarrow$  Instantancements are represented as dashed arrows:  $-->$

```

1  ASK{
2  ?robot isA Robot.
3  ?gripper isA Gripper.
4  ?robot isConnectedToResource ?gripper
5  ?gripper hasJaws ?jaws.
6  ?jaws canGrip Portafilter.
7  ?gripper hasElement Portafilter.
8
9  Portafilter hasElement Coffee.
10 Coffee hasProperty IsTamped.
11 IsTamped hasValue True.}
12
13 ?portafilter isA Portafilter.
14 OPTIONAL{CoffeeMachine hasElement ?portafilter}.
15 FILTER(!bound(?portafilter)).
16
17 ?coffeecups isA Coffeecups.
18 OPTIONAL{CoffeeMachine hasElement ?coffeecups}.
19 FILTER(!bound(?coffeecups)).
20 }

```

} There must be a robot, which has a gripper mounted, which has suitable gripper jaws to grasp the portafilter.

} The gripper must hold a portafilter filled with coffee and the coffee must be tamped.

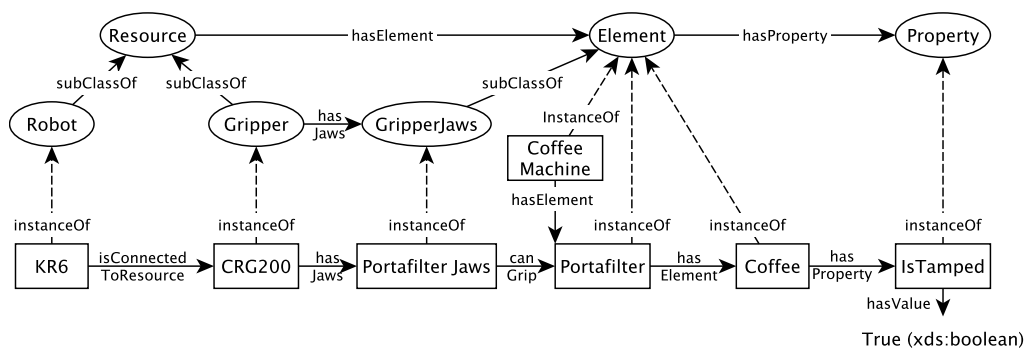
} The coffee machine cannot have a portafilter inserted.

} The coffee cups are not placed under the outlet of the coffee machine.

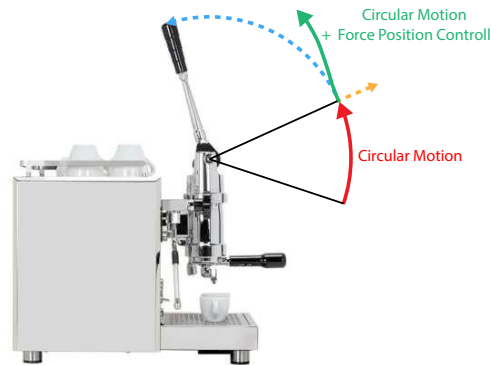
**Listing 10.2.** SPARQL ASK query for checking whether all preconditions for clamping the portafilter in the coffee machine have been met

the portafilter into the coffee machine are fulfilled, while Listing 10.2 shows the SPARQL ASK query to check if all preconditions are met. There is the KR6 robot connected to the CRG200 gripper with gripper jaws that can hold the portafilter, and the gripper holds the portafilter. Thus, the first precondition is fulfilled. The portafilter is filled with coffee, which has the property of being tamped. This fulfills the second precondition. The third precondition is fulfilled since there is no connection `hasElement` between a portafilter instance and the coffee machine. Because the coffee machine also has no connection to coffee cups, the last precondition is also fulfilled. Thus, in this case, the AS could be executed, and after successful execution, the postconditions would be applied. After doing so, the ontology would look as shown in Figure 10.30. The connection between the CRG200 gripper jaws and the portafilter is deleted, and a new connection between the coffee machine and the portafilter is added. As shown for this one task, this can be done for all further tasks, and a sequence for the process can be determined with the help of the adapted depth search shown in Section 8.4.4.

As not all tasks can be performed with the same gripper jaws and, for example, different gripper jaws are required for pulling the lever of the coffee machine than for inserting the portafilter, additional ASs have been added for placing tools and picking up tools. The placement has as a precondition that a tool is connected to the robot and as a postcondition that no tool is present on the robot. When picking up tools, the precondition is that no tool is mounted and the postcondition is that a tool with the appropriate properties is mounted. If, for example, no suitable gripper is available for the task of inserting the portafilter into the coffee machine, it is possible to ensure that the task can be carried out by placing the current tool and then picking up a tool with portafilter gripper jaws. In this way, the tool change is also taken into account in the orchestration of the overall process. This enables the system to determine and subsequently execute a corresponding execution sequence for an overall process that consists of several individual tasks. Since the ontology of the current system description is kept up to date, it



**Figure 10.30.** Snapshot of the ontology showing the system state after the portafilter has been successfully clamped into the coffee machine. Classes are represented as ellipses: Instances are represented as rectangles: Relations between classes are shown as arrows:  $\longrightarrow$  Instantiations are represented as dashed arrows:  $- - \longrightarrow$

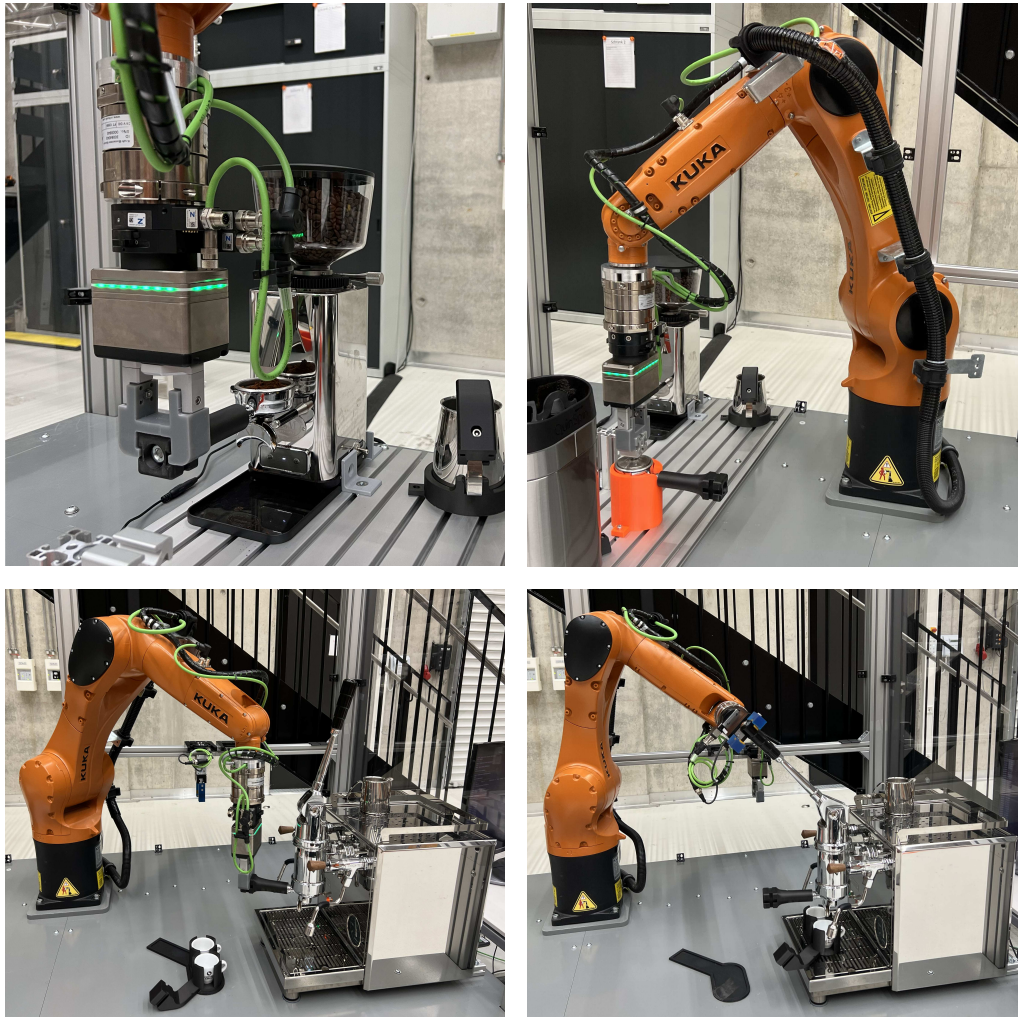


**Figure 10.31.** Schematic representation of the lever trajectory of the robot on the coffee machine. The first part of the lever movement is a pure circular movement (red path). The second part is a circular motion with a superimposed force control (green path).

is also possible to abort process executions and perform orchestrations that continue to plan on the current system state as the current precondition.

Now that the orchestration has been shown, the reusability of services will be discussed. All individual steps except Tamp coffee flour with the tamper and Pull the lever of the coffee machine can be implemented using the pick and place service. The tasks Tamp coffee flour with the tamper and Pull lever of the coffee machine are force-controlled processes. In the task Tamp coffee flour with the tamper, the tamper is first picked up and placed over the portafilter, which can again be implemented using a pick and place service. The robot then moves with the tamper onto the coffee flour until a suitable pressure is reached. This corresponds exactly to the approach movement of the screwdriver when inserting the screw into the aluminum profiles. Therefore, the Force-Torque Position Controller RTS implementation with PID controller is used.

The lever movement must also be force-controlled. For the espresso coffee brewing process, the lever must first be pressed all the way down. This movement represents a circular path and can be implemented by the `move_circ` skill of the robot. The lever must then be moved upwards again. As soon as a certain angle of the lever is reached, the coffee machine generates a resistance to the lever, at which point the lever can be released. From this point on, no more force should be applied to the lever. The lever up movement is implemented in two steps. Figure 10.31 shows schematically the resulting robot trajectory. First, a circular movement is executed up to a certain angle of the lever (red path), and then this circular movement is combined with a force position controlled movement (green path). The underlying circular motion (blue path) maintains the orientation of the gripper in relation to the lever path. The force position controller ensures that the velocity of the circular path is influenced by the forces of the lever acting on the gripper and at the same time that the robot moves away from the lever (orange arrow). For implementation, this means a `move_circ` movement is executed, which is superimposed with a force-controlled movement implemented by



**Figure 10.32.** Pictures of the execution of the Robarista showing the four process steps: Grind coffee into the portafilter with the electric coffee grinder (top left), Tamp coffee flour with the tamper (top right), Clamp portafilter into the coffee machine (bottom left) and Pull lever of the coffee machine (bottom right).

the Force-Torque Position Controller RTS with the PID controller. Thus, it could be shown that even for a completely different use case, the services could be reused. Only the parameters, such as force and torque directions, positions and orientations, and control variables must be adapted.

Figure 10.32 finally shows pictures of the execution of the Robarista application. On the top left, the process step Grind coffee into the portafilter with the electric coffee grinder is shown. The start of the grinding process is triggered by a button on the coffee grinder when the portafilter is inserted. The robot then waits a defined time until the coffee grinder is ready. The portafilter is then inserted into the tamper holder, and the robot

retrieves the tamper. The picture on the top right shows the execution of the process Tamp coffee flour with the tamper. The portafilter is then clamped into the coffee machine by the robot, as shown at the bottom left. The cupholder with the cups is then inserted beneath the portafilter outlet and the tool is changed in order to be able to pull the lever. The robot then operates the lever, as shown on the bottom right.

Thus, it could be shown for this case study that, **process flows can be orchestrated based on system descriptions and preconditions and postconditions of the individual process steps. In addition, it could be shown that the developed services could be adapted for a completely different use case only through appropriate parameterization without making adjustments to the services themselves.**



**Summary.** In addition to the feasibility studies of the case studies, the performance of the architecture has been evaluated. The main focus is on the real-time capability of the distributed execution. However, time measurements were also taken when adding new hardware or setting up real-time communication.

# 11

## Evaluation of the Real-Time Performance

<b>11.1 Performance Evaluation of Adding Software and Hardware Components at Runtime . . . . .</b>	<b>172</b>
<b>11.2 Test Setup for the Real-Time Performance Measurements . . . . .</b>	<b>173</b>
<b>11.3 Latency Evaluation of the Real-Time Communication and Execution . . . . .</b>	<b>174</b>

The measurement of real-time performance holds crucial significance for real-time systems due to its direct impact on their functionality and reliability. Real-time systems are designed to meet stringent timing constraints, where tasks must be executed within specific time windows. Consequently, accurate performance measurement enables to ascertain whether these timing constraints are consistently met. For this reason, after it has been shown how the RealCaPP architecture was used for the implementation of various real-time critical robot applications, this chapter will now go into more detail about what the individual execution times of the Real-Time Services (RTSs) are, and what influences the distribution of the services to different computing nodes has.

In Section 11.1, the measured times for adding new resources to the system are evaluated. In particular, the execution times of the self-introduction process and the establishment of real-time communication were considered. Subsequently, the execution times for real-time execution are examined. First, the evaluation setup is described, in which the execution times were measured (see Section 11.2). In Section 11.3, the execution times for a real-time critical robotic process were considered. Both local and distributed executions were evaluated, and the corresponding execution times were analyzed.

## 11.1 Performance Evaluation of Adding Software and Hardware Components at Runtime

Even though the self-introduction and the initial initialization of a newly added hardware component is not real-time critical, care has been taken to ensure that it is implemented in a performant manner. As shown in Section 5.3, the OPC UA discovery service is used to add new hardware devices to the RealCaPP system. Especially when starting systems, several hardware components become active simultaneously and want to register themselves with the system. Time measurements were performed for the registration of new hardware components, including the transfer of the own self-description. The measurements were each taken for up to five simultaneously added hardware devices. 100 measurements were performed for each outcome. Table 11.1 shows the recorded times for adding hardware components for the sequential and parallel implementation of the OPC UA LDS-ME.

Setup	Min [ms]	Max [ms]	Avg [ms]	SD [ms]
Sequential (1 device added)	300.10	300.83	300.44	0.24
Sequential (2 devices added)	500.09	500.84	300.56	0.35
Sequential (5 devices added)	1,100.12	1,100.80	1,100.46	0.55
Parallel (up to 5 devices added)	200.12	200.92	200.42	0.52

**Table 11.1.** Times for the self-introduction of resources in milliseconds. Comparison between the sequential and parallel implementation.

The values shown illustrate the advantage of parallelization in the addition and self-introduction of the hardware components. Even with five hardware devices added in the sequential case, it took over a second ( $\varnothing$  1,100.46 ms) to add the devices to the RealCaPP architecture. In contrast, the parallel implementation required only 200.42 ms on average to add the five new hardware devices and integrate their self-description into the system. What is noticeable is that the parallel implementation is already significantly more performant for the addition of a single hardware component (Sequential:  $\varnothing$  300.44 ms vs. Parallel:  $\varnothing$  200.42 ms). This is because several requests to the OPC UA information model are processed in parallel.

In addition to the time measurement for the self-introduction, times were also measured for the establishment of an OPC UA over TSN communication. In the case of a Plug & Produce mechanism, the duration of the startup phase is particularly relevant. The startup phase includes the time from starting the program until the first message is sent or received. The measured time includes all steps presented in Chapter 7 from configuring the network interfaces, time synchronization between devices, adding data topics, reserving time slots for the TSN communication to sending or receiving the first message. Table 11.2 shows the measured times for the entire RealCaPP real-time communication startup process, as already published in Eymüller et al. [52]. Measurements were carried out with the clocks of all devices already synchronized and measurements in which the synchronization time was taken into account. Again, 100 measurements per setup were performed for the measurement.

Setup	Min [s]	Max [s]	Avg [s]
Add Publisher, time sync needed	20.1	23.3	21.5
Add Publisher, time already synced	6.4	7.0	6.7
Add Subscriber, time sync needed	19.7	22.4	20.8
Add Subscriber, time already synced	4.0	4.0	4.0

**Table 11.2.** Times for initializing and starting real-time communication in seconds. Comparison of times with clocks synchronization and already existing clock synchronization.

The results show that the time until all clocks run synchronously has a decisive influence on the start times. Because the clock synchronization via PTP needs approximately 15 s. Adding a publisher to an already time-synchronized network takes an average of 6.7 s. If the clock must be synchronized beforehand, about 21.5 s are required to add a publisher. Adding a subscriber requires 4.0 s on average in the synchronized network. Including time synchronization, it takes an average of 20.8 s to add a subscriber to the system. What is also worth mentioning here is that for the transmission or reception of the first message, the clock waits until it reaches an exact second value, the nanosecond counter is zero. For this, a sleep of 3 seconds plus the missing time to the full second was used.

## 11.2 Test Setup for the Real-Time Performance Measurements

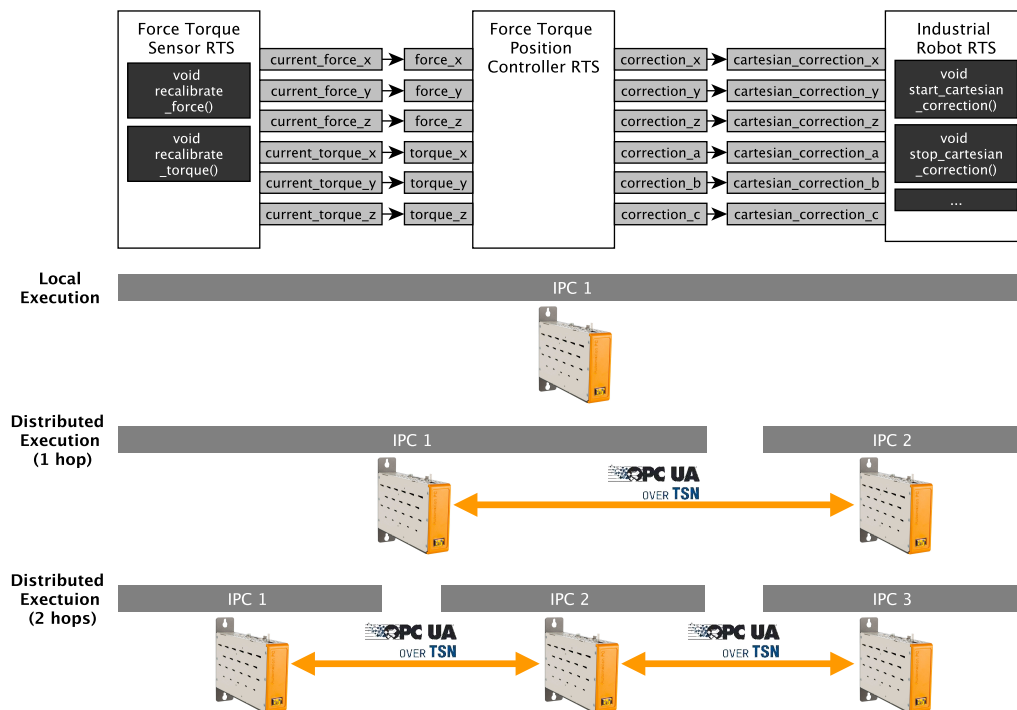
The following test setup was used for all performance measurements. Three identical IPCs of the type Automation PC 2200 [28] from B&R Automation were used. The IPCs are equipped with an Intel Atom E3940 quad-core processor running at 1.6 GHz. The IPCs have 8 GB of RAM and four integrated Intel I210 network cards that support TSN. All IPCs operate with a real-time capable Preempt-RT Linux Kernel (5.11.4-rt11) with enabled ETF (Earliest TX-time First) drivers for TSN. All IPCs are connected to a TrustNode [80] TSN capable network switch from InnoRoute via one I210 port. The four cores of the IPCs are operated in isolation mode. Therefore, a fixed assignment of the processes to a core must be made. On all IPCs, the first core is used for system operations and other running applications. The second core is responsible for sending TSN network packets, the third core is responsible for receiving and processing TSN network packets, and the fourth core runs the main process of the RealCaPP architecture and the additional software components (RTSs). For the performance measurements in C++, the Linux system function `clock_gettime(CLOCK_REALTIME)` [114] was used, which can display the current time in a resolution of one nanosecond. The `CLOCK_REALTIME` is set globally in the system by the PTP time synchronization. Therefore, measurements on different IPCs are possible. The performance of a distributed system can be measured relatively accurately. As OPC UA stack, the open source C stack `open62541` [155] was used. The advantage of this stack is that it already supports publish-subscribe communication via TSN [158].

### 11.3 Latency Evaluation of the Real-Time Communication and Execution

For the evaluation of the execution times, the force-controlled movement process was considered, as it was used for screwing or pressing the coffee flour. This process requires hard real-time and consists of three independent RTSs, which can be well distributed. This means that the process can be executed locally or distributed on several compute nodes, and it can be checked whether the real-time limits can be met. Figure 11.1 shows the individual evaluations that are carried out. The three-part process is first executed locally on one IPC, for which OPC UA over TSN communication is not yet necessary. In the second step, the sensor data is read out, and the position calculation is carried out on one IPC. The position values are then transmitted to the second IPC via OPC UA over TSN, where they are processed for the robot and sent to the robot. In the final case, all three process steps are executed on separate IPCs. On one IPC, the sensor data is read, sent via OPC UA over TSN to the second IPC, where the force data is processed. Then position correction values are sent via OPC UA over TSN to the third IPC, where the position corrections are processed and sent to the robot.

Due to industrial robots' high velocities and accelerations, control loops must operate at high frequencies between 100 Hz and 1 kHz. Therefore, the target is to achieve

Real-Time  
Deadline: 1 ms



**Figure 11.1.** Evaluation scenarios for the measurement of runtimes using the example of force-controlled robot motion. Three different setups are considered. The local execution, the execution on two IPCs and the execution on three IPCs.

a guaranteed frequency of 1 kHz (cycle time of 1 ms) for the force control system implemented with the RealCaPP service architecture. This results in a real-time deadline of 1 ms, which cannot be exceeded.

In the first step, the runtimes of the individual RTSs were measured. The execution time of an RTS includes the reading of the InPorts, the processing of the data, and the writing of the processed data to the OutPorts. In order to obtain a significant runtime statement,  $10^8$  (100,000,000) runs per RTS were performed, and the execution times for each run were measured in nanoseconds. The values were each plotted in histograms, with the horizontal axis indicating the execution time in nanoseconds and the vertical axis indicating the frequency of the measured time values. The vertical axis is provided with a logarithmic scale. Otherwise, outliers cannot be detected. For each plot, the respective minimum values, maximum values, and the average were inserted as dashed lines. In addition, for each evaluation, the numerical values for the minimum value, maximum value, average value, and standard deviation were given below the plot. The most relevant values for real-time are the Worst-Case Execution Times (WCETs), meaning the maximum execution times.

Figure 11.2 shows the execution times of the Force-Torque Sensor RTS. An average execution time of 27.17 ns was required to read and write the sensor values to the OutPorts. The low standard deviation of only 11.91 ns also shows that a large number of the measurements are close to the average value. Overall versions of the RTS, there was only one maximum outlier with 7,040 ns. Thus, a WCET of 8  $\mu$ s is assumed.

WCET Force-Torque Sensor RTS: 8  $\mu$ s

Figure 11.3 shows the execution times of the Force-Torque Position Controller RTS. Reading out the InPorts, processing the values with the PID controller, and then writing the position correction values to the OutPorts takes an approximate time of 101.87 ns. There is also a small standard deviation of only 20.39 ns for this RTS. The maximum execution time is 6,220 ns. Here, there was only one occurrence of the outlier. Thus, a WCET of 7  $\mu$ s is assumed.

WCET Force-Torque Position Controller RTS: 7  $\mu$ s

Finally, Figure 11.4 shows the execution times of the Industrial Robot RTS. This RTS has the largest average runtime of 2,190.62 ns. This time is needed, on average, to read the position correction values from the InPorts, translate them into an XML format, and then send them to the robot. Since the processing into the XML format requires multiple string operations, the execution time for this RTS is slightly higher compared to the others. Also, a higher variance of the execution times can be seen in the higher standard deviation of 103.31 ns. The maximum execution time for the measurement was 11,420 ns. Again, there was only one outlier. Thus, a WCET of 12  $\mu$ s is assumed.

WCET Industrial Robot RTS: 12  $\mu$ s

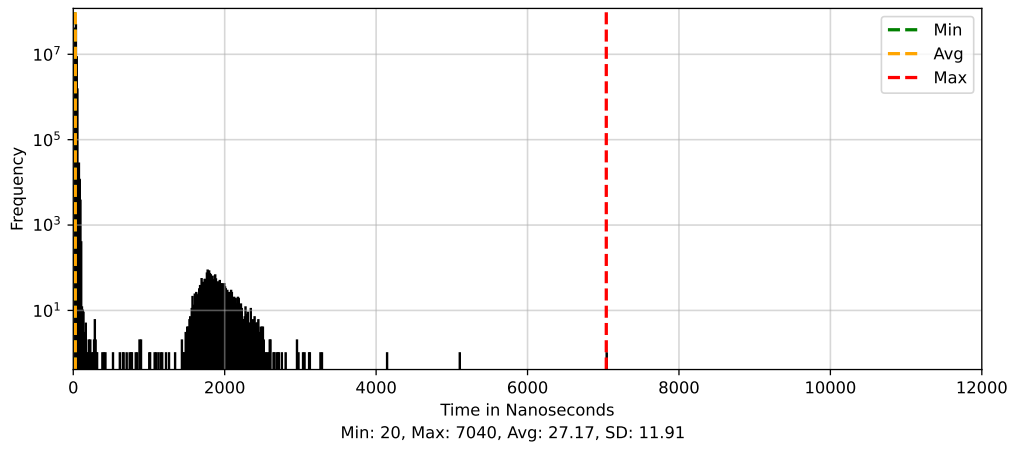


Figure 11.2. Histogram of the Force-Torque Sensor RTS execution time

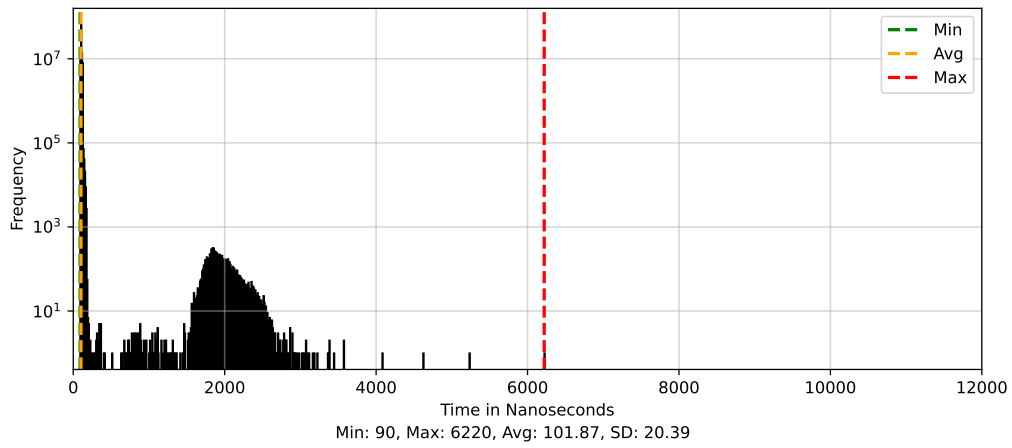


Figure 11.3. Histogram of the Force-Torque Position Controller RTS execution time

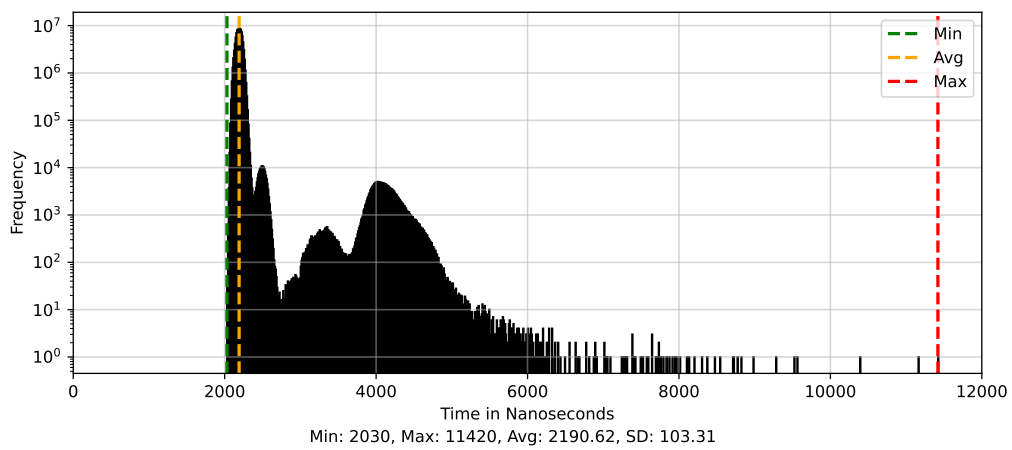
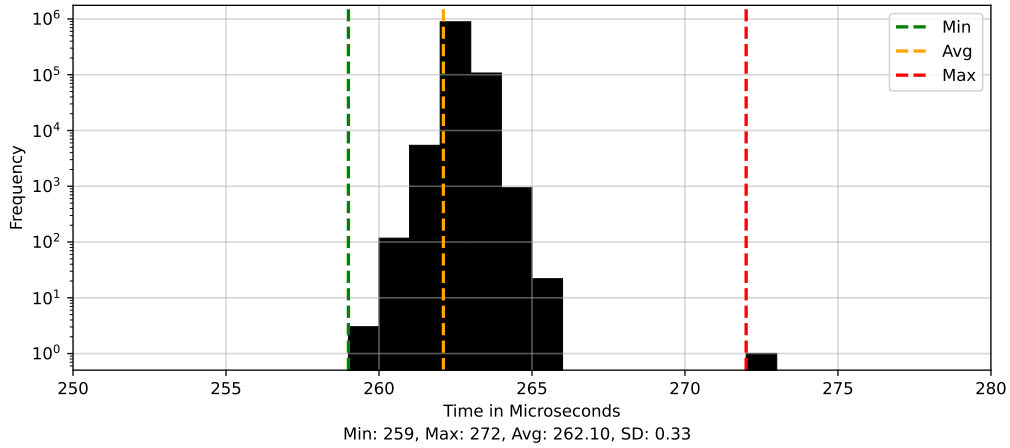


Figure 11.4. Histogram of the Industrial Robot RTS execution time



**Figure 11.5.** Histogram of the round trip time of the OPC UA over TSN network transmission

In addition to the execution times, the transfer times for a network transfer were measured. For the measurement of the transmission time of the OPC UA over TSN connection, the measurement of the Round Trip Time (RTT) is best suited. This means that a sender sends a message to another node, which loops back the message. That means the message is received from the second node and then returned to the sender. For the time measurement, a time stamp is stored when the message is sent from the first node, and when it is received again at the first node, then the difference between these timestamps is taken as the RTT. To measure the RTT,  $10^6$  (1,000,000) messages were sent and received. The OPC UA over TSN connection runs with a cycle time of  $250 \mu\text{s}$ . Since the transmission times are strongly dependent on the amount of data, 10 floating point values with a size of 8 bytes each were transmitted for the measurement in each message. Thus, 80 bytes are transmitted per message.

Figure 11.5 shows the RTT measurement as a histogram. On average, a time of  $262.10 \mu\text{s}$  was required for one transmission round. With a low standard deviation of only  $0.33 \mu\text{s}$ , there is little variation in the time values. The longest transmission round lasted  $272 \mu\text{s}$  and occurred once. The RTT can be used well as WCET for one message transfer since, in the worst case, the message has to wait for one transfer cycle before it is sent. Thus, a WCET for the transfer of one message of  $272 \mu\text{s}$  is assumed.

In the following evaluation section, the total execution times for the entire process were measured. To measure the performance, a timestamp was recorded before and after the execution of one application cycle, and the difference between the two timestamps was measured. To make certain that this assessment functioned effectively even within the distributed configuration, the initial timestamp was integrated into each RTS as a port and was thus dragged through all RTS. A total of  $10^5$  (100,000) iterations were evaluated for each configuration.

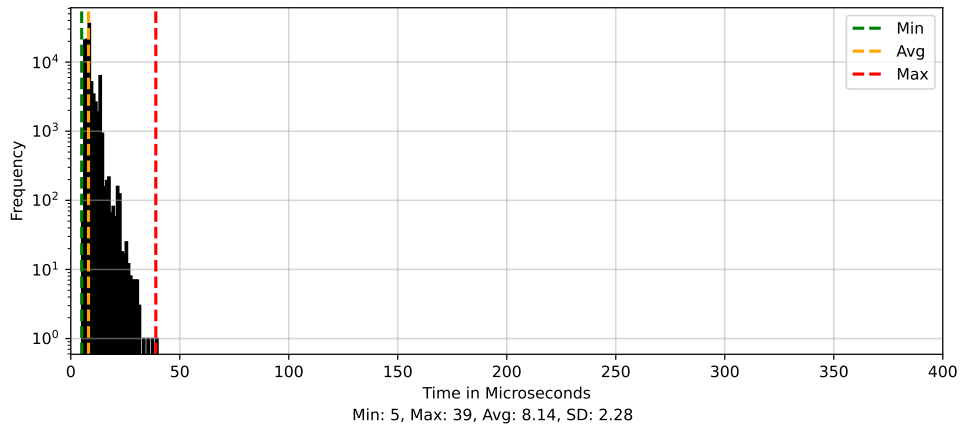
In order to compare the RealCaPP architecture against existing approaches, the same application was implemented with ROS2. ROS2 offers the most similarities to the Re-

WCET Transmis-  
sion Time:  $272 \mu\text{s}$

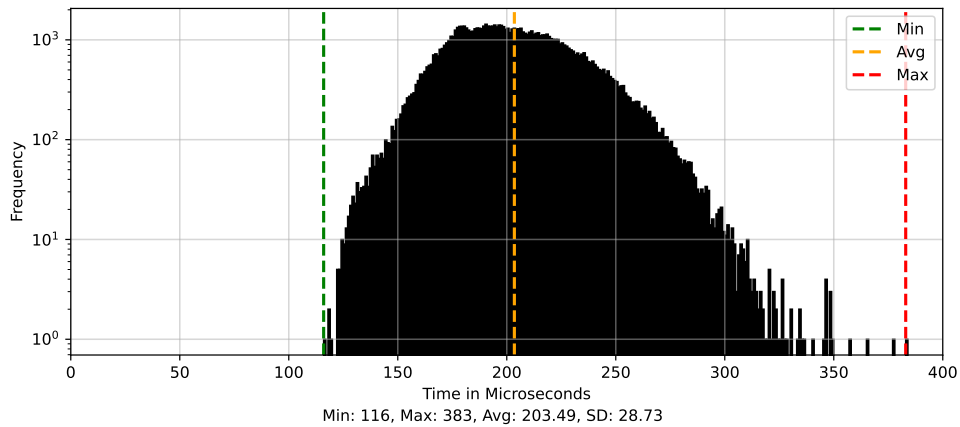
RealCaPP vs ROS2

alCaPP architecture, since ROS2 can be executed in a distributed manner on several computers. In ROS2 the software components can be modularized and communication takes place via a common middleware. In addition, ROS2 can be programmed in C++. In this scenario, the three RTSs are realized as three distinct ROS2 nodes, each containing the exact same C++ implementation for the various functions of measuring the force-torque values, calculate a position correction value out of the force-torque values, and send the position correction values to a robot. The ports of the RTSs are represented as publishers and subscribers in ROS2. Because not all Data Distribution Service (DDS) implementations in ROS2 possess real-time capabilities, the real-time DDS implementation Connex DDS Professional [178] from Real-time Innovations (RTI) was used. On non-distributed systems, the Connex DDS implementation uses shared memory to exchange data between software components. Since there is no TSN implementation for ROS2 yet, the measurements were performed in a separate network without additional network traffic. This prevents delays in the network due to other network traffic. When implementing the ROS2 nodes, the same real-time criteria were taken into account as for the RTSs. For example, memory was created before the actual real-time execution in a separate initialization phase. In the RealCaPP implementation, the transmission slots were not synchronized with the execution clocks. With synchronization, it would be possible to time RTS executions to always be just before the next transmit cycle. Since such synchronization is not possible in ROS2, it has been omitted for comparability between the two architectures.





**Figure 11.6.** Histogram of the execution times of the force-controlled robot movement local RealCaPP implementation



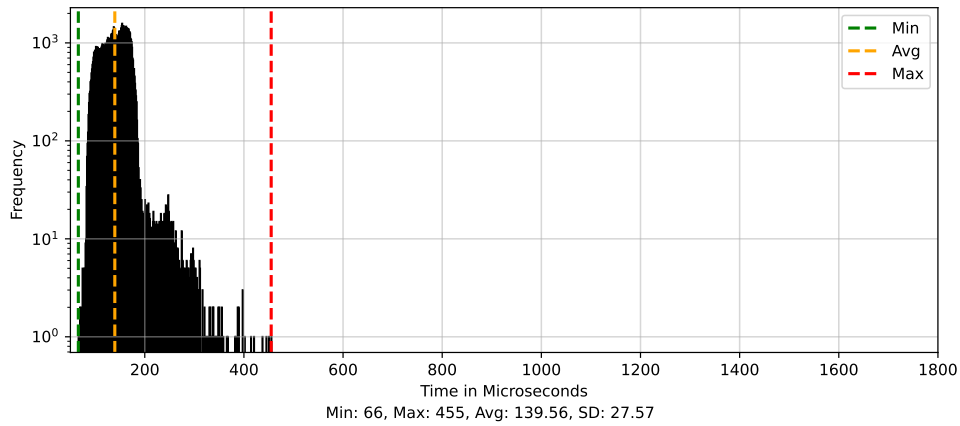
**Figure 11.7.** Histogram of the execution times of the force-controlled robot movement local ROS2 implementations

Figure 11.6 shows the execution times for the local execution of the force-controlled robot movement implemented with the RealCaPP architecture as a histogram. On average, a run time of  $8.14 \mu s$  was required to run from reading the sensor values to writing the position values to the robot. The standard deviation was  $2.28 \mu s$ , which indicates very small deviations. In the measurements, a maximum value of  $39 \mu s$  was reached once. Thus, the WCET for the local execution with the RealCaPP implementation is  $39 \mu s$ .

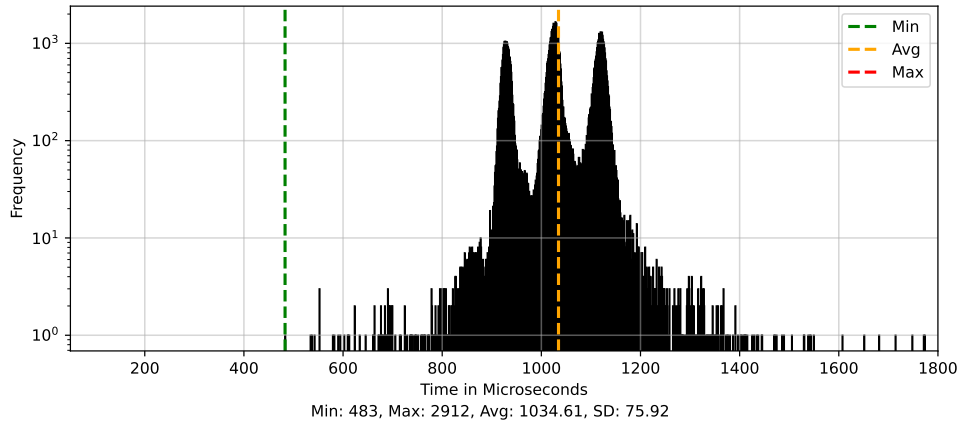
WCET RealCaPP  
Execution Time  
(local):  $39 \mu s$

Figure 11.7 shows the execution times for the same force-controlled robot movement implemented with the ROS2 architecture as a histogram. The average value here was  $203.49 \mu s$  with a standard deviation of  $28.73 \mu s$ . The significantly higher standard deviation indicates partially higher distributed execution times. The maximum execution time of  $383 \mu s$  was also clearly higher. Thus, the WCET for the local execution with the ROS2 implementation is  $383 \mu s$ .

WCET ROS2  
Execution Time  
(local):  $383 \mu s$



**Figure 11.8.** Histogram of the execution times of the force-controlled robot movement distributed RealCaPP implementation (1 hop)



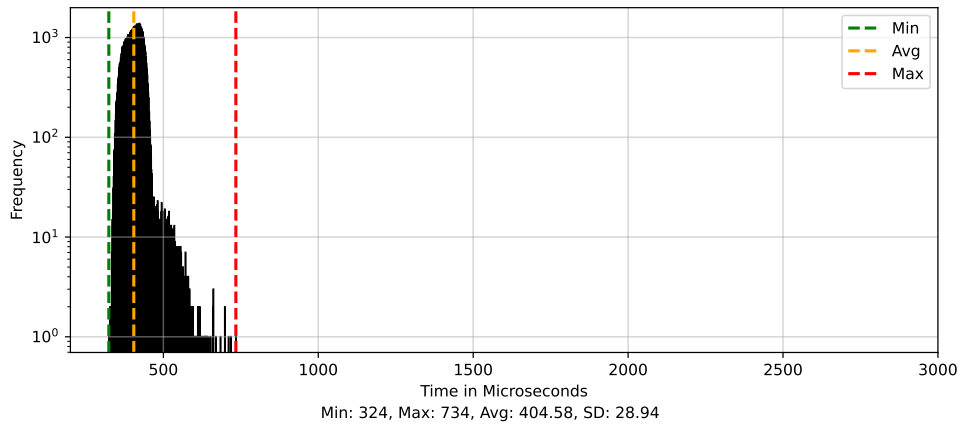
**Figure 11.9.** Histogram of the execution times of the force-controlled robot movement distributed ROS2 implementation (1 hop)

WCET RealCaPP  
Execution Time  
(distributed 1 hop):  
 $455 \mu s$

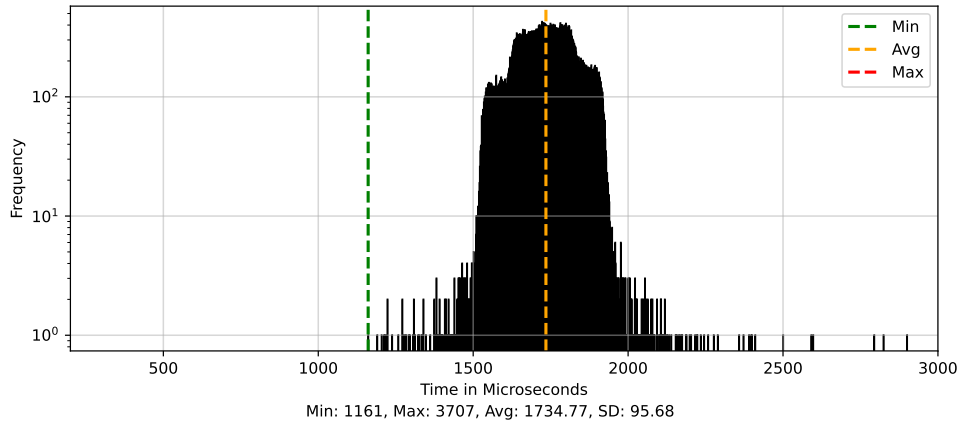
Figure 11.8 shows the execution times for the distributed execution on two computation nodes. The reading of the sensor values and the calculation of the position correction values is carried out on the first IPC, then the position values are transferred to the second IPC and pre-processed for the robot and sent to the robot. With the RealCaPP architecture on average, an execution time of  $139.56 \mu s$  was required. The standard deviation is also slightly higher with a value of  $27.57 \mu s$ . The maximum execution time was  $455 \mu s$ . Thus, the WCET for the distributed execution with one hop with the RealCaPP implementation is  $455 \mu s$ .

WCET ROS2  
Execution Time  
(distributed 1 hop):  
 $2,912 \mu s$

Figure 11.9 shows the execution times for distributed execution on two computation nodes implemented in ROS2. Overall measurements, an average execution time of  $1,034.61 \mu s$  was achieved. The standard deviation of  $75.92 \mu s$  indicates a higher jitter. The maximum execution time was  $2,912 \mu s$ . Due to readability, the significant outliers in the histogram were cut off. Thus, the WCET for the distributed execution with one hop with the ROS2 implementation is  $2,912 \mu s$ .



**Figure 11.10.** Histogram of the execution times of the force-controlled robot movement distributed RealCaPP implementation (2 hops)



**Figure 11.11.** Histogram of the execution times of the force-controlled robot movement distributed ROS2 implementation (2 hops)

Figure 11.10 shows the execution time for the distributed execution implemented with the RealCaPP architecture on three computation nodes. This means that each substep is executed on a separate IPC. The average execution time was  $404.58 \mu s$  with a standard deviation of  $28.94 \mu s$ . The maximum execution time has been  $734 \mu s$ . Thus, the WCET for the distributed execution with two hops with the RealCaPP implementation is  $734 \mu s$ .

WCET RealCaPP  
Execution Time  
(distributed 2 hops):  
 $734 \mu s$

Figure 11.11 shows the execution in ROS2 distributed over three IPCs. The average execution time was  $1,734.77 \mu s$ . The standard deviation was almost  $100 \mu s$  ( $95.68 \mu s$ ), which is very high. The maximum execution time for the measurements performed was  $3,707 \mu s$ . Thus, the WCET for the distributed execution with two hops with the ROS2 implementation is  $3,707 \mu s$ .

WCET ROS2  
Execution Time  
(distributed 2 hops):  
 $3,707 \mu s$

Comparison  
of execution  
times: RealCaPP  
vs. ROS2

All test series, both in the local and in all distributed setups, show that the implementation with the RealCaPP architecture achieves significantly better execution time values than the reference implementation with the ROS2 architecture. Puck et al. [173] has measured the performance for the transmission of ROS2 messages in a local and distributed setup. The measured values also correspond to the transmission values that were achieved with ROS2 in the shown setup. Thus, it can be excluded that there were errors in the ROS2 configuration. In all cases, the WCET of the RealCaPP implementation was below the minimum value of the ROS2 implementation. There was no overlap in the execution times of the two architectures. In the local setup, the largest deviation of the RealCaPP WCET ( $39 \mu\text{s}$ ) from the ROS2 WCET ( $383 \mu\text{s}$ ) was seen with a difference of almost factor 10. This high deviation is mainly due to the overhead of DDS in the ROS2 implementation, although shared memory is used. The standard deviations in the local evaluation also differ by a factor of approximately 10. The RealCaPP implementation has a very good standard deviation of  $2.28 \mu\text{s}$  while in the ROS2 implementation, the execution values vary strongly with a standard deviation of  $28.73 \mu\text{s}$ . But even with distributed execution distributed to tree IPCs, the RealCaPP implementation WCET ( $734 \mu\text{s}$ ) was better than the ROS2 reference implementation WCET ( $3,707 \mu\text{s}$ ) by a factor of 5. The distribution of the execution values for the distributed case also differs greatly between the RealCaPP and the ROS2 implementations. Here, the standard deviations of  $28.94 \mu\text{s}$  for the RealCaPP implementation and  $95.68 \mu\text{s}$  for the ROS2 implementation are more than factor 3 apart. The reason for the poorer runtime of ROS2 could be the overhead of DDS and the network transfer without TSN.

It was thus possible to demonstrate in all cases that the real-time deadline of 1 ms ( $1,000 \mu\text{s}$ ) could be met for the RealCaPP implementation of the force-controlled robot movement. In the reference implementation in ROS2, the real-time deadline was already broken during the distributed execution with one hop. Since network transmission has a decisive influence on execution time, care should be taken when designing RTS Networks to ensure that as few external connections as possible are created. Especially constellations, where data has to be exchanged between network parts at more than one point, can be critical.

**Summary.** In this chapter, the results obtained in the course of this thesis are once again presented and evaluated. The thesis concludes with an outlook on possible further research questions.

# 12

## Conclusion and Outlook

<b>12.1 Conclusion of the Thesis</b> . . . . .	<b>183</b>
<b>12.2 Outlook and Future Work</b> . . . . .	<b>185</b>

The goal of this work was the development of a real-time Plug & Produce architecture. The architecture developed was successfully validated in four case studies. With the architecture, it is possible to implement real-time capable processes in robotic systems. This was successfully demonstrated using the examples of hand-guiding, the assembly and mounting of components, and the brewing of coffee. These case studies also show how flexible the developed concept can be applied.

The first part of this chapter provides a summary of the goals achieved. In addition, Section 12.1 checks whether all points of the research question are answered. At the end of this dissertation, an outlook is given in Section 12.2, and possible further research questions are discussed.

### 12.1 Conclusion of the Thesis

The initial research question was the following:

*How can flexible robot-based production systems be designed to add or replace production resources on the fly without complex integration and reprogramming, and how can these resources be combined into a distributed system that can perform critical tasks in real-time?*

The research question can now be answered based on the shown results and concepts. The Real-Time Capable Plug & Produce (RealCaPP) architecture is a software architecture for programming flexible robot-based production systems. In the architecture, production resources can be added and replaced easily. Due to a self-introduction mechanism of the resources, an effortless integration is possible, without adapting the implementation. Through modular software components, it is also possible to adapt

or add processes. These software components can be easily integrated at runtime, and a plant gets new skills through the integration. These software components can be distributed arbitrarily in the system and executed in real-time despite the distribution.

The shown RealCaPP architecture is based on a two-part communication middleware (cf. Chapter 4). OPC UA client-server communication is used for non-real-time communication, like for the introduction of new components. In order to exchange process data in real-time in the distributed system, OPC UA Pub-Sub over TSN is used as real-time communication middleware. Other essential components of the architecture are resources. Through an Asset Administration Shell (AAS), the production resources are made RealCaPP capable. The AAS contains descriptions of the resource's properties and capabilities, the hardware driver, and basic callable skills (Basic Skills) in the form of services. It is also possible to add additional software modules to the AAS of the resource, like Composed Skills. The resources also contain interfaces for the two communication types. In addition, there are central points in the architecture where knowledge about the plant is collected, and new components can be registered.

For the integration of new components, the OPC UA Discovery Service is used (cf. Chapter 5). Via non-real-time communication, the self-description of the new resource is then integrated into the global knowledge base of the system. For the self-descriptions and the knowledge base, ontologies are used. Ontologies provide a structured and standardized way to represent knowledge. This structured representation enables machines to perform automated reasoning and inference, allowing more precise and consistent decision-making based on the data. In combination with querying, reasoning is used to automatically find plant configurations for given processes (cf. Chapter 6). The ability to link the data into ontologies makes adding additional constraints for the configuration search easy.

Once a suitable resource configuration has been found, communication between the resources is required for the execution of a process. OPC UA Pub-Sub over TSN is used to communicate between the resources in real-time. For TSN, time slots are defined for sending messages. For this purpose, a dynamic configuration was developed based on a developed OPC UA Pub-Sub Registry, in which the resources can automatically establish real-time communications with each other (cf. Chapter 7).

In order to add new processes at runtime without having to reprogram the entire system, modular software components were developed that can be integrated at runtime. The RealCaPP service architecture was developed for this purpose (cf. Chapter 8). Real-Time Services (RTSs) are uniform services that can be executed in real-time. RTSs have ports that allow multiple RTSs to be connected to each other and represent the data flow between the services. In addition, services can also contain functions that can be executed. Processes are implemented as RTS Networks, which consist of several RTSs connected by ports. RTS Networks can be run locally, or they can be run in a distributed manner. In distributed execution, ports are transferred between computation nodes via the real-time capable middleware. The same applies to the function calls on distributed RTSs. By executing RTS in real-time and exchanging data in distributed systems in real-time, the distributed execution of an RTS Network also remains real-time capable.

The RTSs and RTS Networks are thus the execution environment for Basic Skills and Composed Skills.

To ensure the reusability of the modular software components and the associated skills, abstract implementations have been designed for the different classes of resources (cf. Chapter 9). For the different actuator resources, such as the robot, the gripper, and the screwdriver, respective abstract RTS implementations were developed. The same was done for the sensor resources, such as the force-torque sensor or simple digital input modules. An attempt was made to work out the commonalities for each class. Specializations were used for specific types. A vacuum gripper, for example, has all the functions of a gripper but has additional functionalities that only vacuum grippers have.

Subsequently, real resources were implemented according to the abstract definitions. Two different robot cells were used to check whether the architecture is suitable for implementing flexible robot-based production. In three different industrial case studies, the architecture was able to show its strength (cf. Chapter 10). It was even possible to show how the skills developed for the industrial case studies could be reused to make espressos with a robot and a portafilter machine.

Finally, the real-time capability of the RealCaPP architecture was demonstrated (cf. Chapter 11). For this purpose, the Worst-Case Execution Times (WCETs) were measured for the exemplary real-time critical case study of force-controlled screwing. For comparison, the same case study was implemented with the Robot Operating System 2 (ROS2) using a real-time capable DDS implementation. In both the local and the distributed executions on multiple computation nodes, 1/3 smaller WCETs could be achieved with the RealCaPP realization. In the non-distributed setup, the WCETs of the RealCaPP implementation were less than 1/5 of the ROS2 WCETs.

In summary, RealCaPP is an architecture for programming flexible robotic cells that can adapt to new resource contexts and new processes while allowing executions in hard real-time in a distributed system consisting of many production resources. This provides an enabler for Lot-Size-1 production. Thus, the production system can be adapted according to the product.

## 12.2 Outlook and Future Work

The RealCaPP architecture already offers many possibilities for implementing flexible robot-based production systems as shown in Chapter 10, but there is still potential for expansion.

Since much time is currently needed for time synchronization in the dynamic setup of real-time communication, alternative synchronization approaches are needed for resources that are not connected to the TSN network from the beginning, like robot tools on a tool holder. The aim is to evaluate whether it is possible to enable time synchronization via wireless networks to reduce the time synchronization period when connecting to the TSN network.

Another important aspect that needs further research is the automatic distribution of the RTS to the resources. Currently, the distribution of the RTSs is done by the user. If necessary, existing approaches from other SOAs can be used for the automatic distribution of the services in the RealCaPP system. Distribution optimization criteria could be the limited computing capacity of some resources or distributions with as few communication hops as possible to ensure real-time.

Currently, only very simple component descriptions are used to adapt the skills to the respective product. It has turned out that this is where ontologies reach their limits. Pure descriptions are usually insufficient to handle all kinds of previously unknown components with process reliability. A possible solution to this problem could be a combination of the ontology-based description of the parts, 3D simulations that are as realistic as possible, and Artificial Intelligence (AI) techniques like machine learning. On the one hand, this can be used to enable an optimal design adapted to the part for a given plant configuration. On the other hand, it can be used to find optimal plant configurations for a given task and part. The ontology can query possible plant configurations as shown in Chapter 6. This configuration set can be run in the simulation to estimate the best configuration.

Other aspects that have not yet been considered are collaborative multi-robot applications. Currently, only robotic applications are considered where one robot performs a task or several robots work in a shared workspace doing cooperating tasks, where only one robot moves at a time in a critical area. Collaborative multi-robot tasks can be, for example, the simultaneous handling of one large component or the handover of parts between two robots while the robots are moving. For this purpose, the detection of plant configurations must be adapted to enable the configuration of robot teams. The restrictions here are that a tool can only be used once in a team. Also, new skills must be developed to enable multi-robot collaboration. For example, skills should be added to synchronize the movements of multiple robots, or skills should be added to enable collision-free path planning of robot teams.

Plug & Produce provides the basis for creating a flexible production system that can be adapted over time. Plug & Produce can be combined with self-organization approaches to implement even larger production systems. Combining these two approaches allows the production system to become more adaptable, efficient, and responsive to changing conditions. The goal would be to enter a product blueprint to be produced in the system. The blueprint is automatically broken down into individual process steps. Based on these process steps, possible plant configurations are identified. The system automatically plans the optimal sequence for the process steps, considering the production processes already in progress. It may be necessary to reconfigure the system during production. However, care should be taken to ensure that as few reconfigurations as possible are made. For example, unnecessary, time-consuming tool changes should be avoided.

A major problem that remains and one that is not easily solved, is the Plug & Produce capability of resources. Currently, Asset Administration Shells (AASs) have been developed for each production resource, making resources Plug & Produce capable. However, this means that for each new resource, a new AAS must be created to make the resource Plug & Produce capable. Therefore, an attempt should be made to estab-



lish a vendor-independent Plug & Produce standard, in which any manufacturer of a production resource can provide a Plug & Produce capable resource. In summary, a vendor-independent standard for Plug & Produce is essential to promote interoperability, reduce integration costs, improve efficiency, and ensure production systems can effectively adapt to changing needs.



# Bibliography

- [1] BaSys 4.0: Basissystem Industrie 4.0. URL <https://www.basys40.de/>. (last visited: 26.04.2023).
- [2] openMOS: Open Dynamic Manufacturing Operating System for Smart Plug-and-Produce Automation Components. URL <https://www.openmos.eu/>. (last visited: 06.06.2023).
- [3] Simulink – Made for Model-Based-Design, note =(last visited: 12.07.2023), author=Math Works. URL <https://mathworks.com/products/simulink.html>.
- [4] *Robotics: Modelling, Planning and Control*. Springer London, London, 2009. ISBN 978-1-84628-642-1. doi: 10.1007/978-1-84628-642-1\_12.
- [5] IEEE Standard for Local and metropolitan area networks– Virtual Bridged Local Area Networks Amendment 12: Forwarding and Queuing Enhancements for Time-Sensitive Streams. *IEEE Std 802.1Qav-2009 (Amendment to IEEE Std 802.1Q-2005)*, pages 1–72, 2010. doi: 10.1109/IEEEESTD.2010.8684664.
- [6] IEEE Standard for Local and metropolitan area networks – Bridges and Bridged Networks - Amendment 25: Enhancements for Scheduled Traffic. *IEEE Std 802.1Qbv-2015 (Amendment to IEEE Std 802.1Q-2014 as amended by IEEE Std 802.1Qca-2015, IEEE Std 802.1Qcd-2015, and IEEE Std 802.1Q-2014Cor 1-2015)*, pages 1–57, 2016. doi: 10.1109/IEEEESTD.2016.8613095.
- [7] IEEE Standard for Ethernet Amendment 5: Specification and Management Parameters for Interspersing Express Traffic. *IEEE Std 802.3br-2016 (Amendment to IEEE Std 802.3-2015 as amended by IEEE Std 802.3bw-2015, IEEE Std 802.3by-2016, IEEE Std 802.3bq-2016, and IEEE Std 802.3bp-2016)*, pages 1–58, 2016. doi: 10.1109/IEEEESTD.2016.7592835.
- [8] IEEE Standard for Local and Metropolitan Area Networks–Bridges and Bridged Networks – Amendment 31: Stream Reservation Protocol (SRP) Enhancements and Performance Improvements. *IEEE Std 802.1Qcc-2018 (Amendment to IEEE Std 802.1Q-2018 as amended by IEEE Std 802.1Qcp-2018)*, pages 1–208, 2018. doi: 10.1109/IEEEESTD.2018.8514112.
- [9] IEEE Standard for Ethernet. *IEEE Std 802.3-2018 (Revision of IEEE Std 802.3-2015)*, pages 1–5600, 2018. doi: 10.1109/IEEEESTD.2018.8457469.
- [10] IEEE Standard for a Precision Clock Synchronization Protocol for Networked Measurement and Control Systems. *IEEE Std 1588-2019 (Revision of IEEE Std 1588-2008)*, pages 1–499, 2020. doi: 10.1109/IEEEESTD.2020.9120376.
- [11] IEEE Standard for Local and Metropolitan Area Networks–Bridges and Bridged Networks. *IEEE Std 802.1Q-2022 (Revision of IEEE Std 802.1Q-2018)*, pages 1–2163, 2022. doi: 10.1109/IEEEESTD.2022.10004498.
- [12] A. Alves. *OSGI in Depth*. Simon and Schuster, 2011.
- [13] A. Ambler and R. Popplestone. Inferring the positions of bodies from specified spatial relationships. *Artificial Intelligence*, 6(2):157–174, 1975. ISSN 0004-3702. doi: [https://doi.org/10.1016/0004-3702\(75\)90007-7](https://doi.org/10.1016/0004-3702(75)90007-7).
- [14] N. Ando, T. Suehiro, and T. Kotoku. A software platform for component based RT-system development: OpenRTM-aist. In *International Conference on Simulation, Modeling, and Programming for Autonomous Robots*, pages 87–98. Springer, 2008.

- [15] A. Angerer, A. Hoffmann, A. Schierl, M. Vistein, and W. Reif. Robotics API: Object-oriented software development for industrial robots. *Journal of Software Engineering for Robotics*, 4 (1):1–22, 2013.
- [16] T. Arai, Y. Aiyama, Y. Maeda, M. Sugi, and J. Ota. Agile assembly system by “plug and produce”. *CIRP annals*, 49(1):1–4, 2000.
- [17] T. Arai, Y. Aiyama, M. Sugi, and J. Ota. Holonic assembly system with Plug and Produce. *Computers in Industry*, 46(3):289–299, 2001.
- [18] I. S. Association et al. IEEE Std 802.1 AS-2011, IEEE Standard for Local and Metropolitan Area Networks—Timing and Synchronization for Time-Sensitive Applications in Bridged Local Area Networks. *Mar*, 30:292, 2011.
- [19] ATI Industrial Automation. F/T Sensor: mini45. URL [https://www.ati-ia.com/products/ft/ft\\_models.aspx?id=mini45](https://www.ati-ia.com/products/ft/ft_models.aspx?id=mini45). (last visited: 31.07.2023).
- [20] S. Bader, E. Barnstedt, H. Bedenbender, B. Berres, M. Billmann, and M. Ristin. Details of the Asset Administration Shell-Part 1: the exchange of information between partners in the value chain of Industrie 4.0 (Version 3.0 RC02). 2022.
- [21] H. Bedenbender, A. Bentkus, U. Epple, T. Hadlich, M. Hankel, R. Heidel, and M. Wooschlaeger. Relationships between I4.0 Components—Composite components and smart production. *Federal Ministry for Economic Affairs and Energy (BMWi), Berlin*, 2017.
- [22] H. Bedenbender, M. Billmann, U. Epple, T. Hadlich, M. Hankel, R. Heidel, O. Hillermeier, M. Hoffmeister, H. Huhle, M. Jochem, et al. Examples of the Asset Administration Shell for Industrie 4.0 Components – Basic Part. *ZVEI white paper*, 2017.
- [23] Bennulf, Mattias and Danielsson, Fredrik and Svensson, Bo. Identification of resources and parts in a Plug and Produce system using OPC UA. *Procedia Manufacturing*, 38: 858–865, 2019.
- [24] I. C. Bertolotti and G. Manduchi. *Real-time embedded systems: open-source operating systems perspective*. CRC press, 2017.
- [25] L. Biagiotti and C. Melchiorri. *Trajectory planning for automatic machines and robots*. Springer Science & Business Media, 2008.
- [26] S. Biffl, A. Lüder, and D. Gerhard. *Multi-disciplinary engineering for cyber-physical production systems: data models and software solutions for handling complex engineering projects*. Springer, 2017.
- [27] H. Boley, A. Paschke, and O. Shafiq. RuleML 1.0: the overarching specification of web rules. In *International Workshop on Rules and Rule Markup Languages for the Semantic Web*, pages 162–178. Springer, 2010.
- [28] B&R Industrial Automation GmbH . Automation PC 2200 – Compact Intel Atom technology. URL <https://www.br-automation.com/en/products/industrial-pcs/automation-pc-2200/>. (last visited: 19.08.2023).
- [29] D. Brickley. RDF vocabulary description language 1.0: RDF schema. *W3C recommendation*, 2004.
- [30] P. Brooks. EtherNet/IP: Industrial Protocol White Paper, 2001. URL [https://literature.rockwellautomation.com/idc/groups/literature/documents/wp/enet-wp001\\_-en-p.pdf](https://literature.rockwellautomation.com/idc/groups/literature/documents/wp/enet-wp001_-en-p.pdf). (last visited: 30.07.2023).

- [31] D. Bruckner, R. Blair, M. Stanica, A. Ademaj, W. Skeffington, D. Kutscher, S. Schriegel, R. Wilmes, K. Wachswender, L. Leurs, et al. OPC UA TSN - A new solution for industrial communication. *Whitepaper. Shaper Group*, 168, 2018.
- [32] D. Bruckner, M.-P. Stănică, R. Blair, S. Schriegel, S. Kehrer, M. Seewald, and T. Sauter. An Introduction to OPC UA TSN for Industrial Communication Systems. *Proceedings of the IEEE*, 107(6):1121–1131, 2019. doi: 10.1109/JPROC.2018.2888703.
- [33] H. Bruyninckx. Open robot control software: the OROCOS project. In *Proceedings 2001 ICRA. IEEE international conference on robotics and automation (Cat. No. 01CH37164)*, volume 3, pages 2523–2528. IEEE, 2001.
- [34] H. Bruyninckx and P. Soetens. Generic real-time infrastructure for signal acquisition, generation and processing. In *Fourth Real-time Linux Workshop*, 2002.
- [35] G. C. Buttazzo. *Hard real-time computing systems: predictable scheduling algorithms and applications*, volume 24. Springer Science & Business Media, 2011.
- [36] C++ Micro Services. C++ Micro Services, 2023. URL <http://cppmicroservices.org/>. (last visited: 14.06.2023).
- [37] S. Cavalieri and F. Chiacchio. Analysis of OPC UA performances. *Computer Standards & Interfaces*, 36(1):165–177, 2013. ISSN 0920-5489. doi: <https://doi.org/10.1016/j.csi.2013.06.004>.
- [38] N. Chungoora, A.-F. Cutting-Decelle, R. I. Young, G. Gunendran, Z. Usman, J. A. Harding, and K. Case. Towards the ontology-based consolidation of production-centric standards. *International Journal of Production Research*, 51(2):327–345, 2013.
- [39] M. Compton, P. Barnaghi, L. Bermudez, R. Garcia-Castro, O. Corcho, S. Cox, J. Graybeal, M. Hauswirth, C. Henson, A. Herzog, V. Huang, K. Janowicz, W. D. Kelsey, D. Le Phuoc, L. Lefort, M. Leggieri, H. Neuhaus, A. Nikolov, K. Page, A. Passant, A. Sheth, and K. Taylor. The SSN ontology of the W3C semantic sensor network incubator group. *Journal of Web Semantics*, 17:25–32, 2012. ISSN 1570-8268. doi: <https://doi.org/10.1016/j.websem.2012.05.003>.
- [40] D. M. Considine and G. D. Considine. *Standard handbook of industrial automation*. Springer Science & Business Media, 2012.
- [41] S. S. Craciunas, R. S. Oliver, M. Chmelík, and W. Steiner. Scheduling real-time communication in IEEE 802.1 Qbv time sensitive networks. In *Proceedings of the 24th International Conference on Real-Time Networks and Systems*, pages 183–192, 2016.
- [42] A.-F. Cutting-Decelle, R. I. Young, J.-J. Michel, R. Grangel, J. Le Cardinal, and J. P. Bourey. ISO 15531 MANDATE: a product-process-resource based approach for managing modularity in production management. *Concurrent Engineering*, 15(2):217–235, 2007.
- [43] DIN. DIN SPEC 91345: 2016-04, Reference Architecture Model Industrie 4.0 (RAMI4.0), 2016.
- [44] J. Domingue, D. Fensel, and J. A. Hendler. *Handbook of semantic web technologies*, volume 1. Springer, 2011.
- [45] K. Dorofeev and A. Zoitl. Skill-based engineering approach using OPC UA Programs. In *2018 IEEE 16th international conference on industrial informatics (INDIN)*, pages 1098–1103. IEEE, 2018.

- [46] DSM Messtechnik GmbH. MultiPro 3G – Modern high-end control system for hand-held nutrunners and built-in nutrunners. URL <https://www.dsm-messtechnik.de/en/control-system/multipro-3g/>. (last visited: 23.07.2023).
- [47] F. Dürr and N. G. Nayak. No-wait packet scheduling for IEEE time-sensitive networks (TSN). In *Proceedings of the 24th International Conference on Real-Time Networks and Systems*, pages 203–212, 2016.
- [48] Eclipse. Eclipse iceoryx – An inter-process-communication middleware. URL <https://iceoryx.io/latest/>. (last visited: 12.07.2023).
- [49] K. Erciyas and K. Erciyas. *Distributed real-time systems*. Springer, 2019.
- [50] Espresso Coffee Machines Manufacture GmbH. S-Automatik 64 – On-Demand Grinder with Timer. URL <https://www.ecm.de/en/products/details/product/Product/Details/s-automatik-64/>. (last visited: 21.07.2023).
- [51] C. Eymüller, J. Hanke, A. Hoffmann, M. Kugelmann, and W. Reif. Real-time capable OPC-UA programs over TSN for distributed industrial control. In T. Sauter, F. Vasques, L. L. Bello, V. Vyatkin, A. A. Nogueiras, and S. Wilker, editors, *2020 25th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA), 8-11 Sept. 2020, Vienna, Austria*, pages 278 – 285, 2020. doi: 10.1109/ETFA46521.2020.9212171.
- [52] C. Eymüller, J. Hanke, A. Hoffmann, W. Reif, M. Kugelmann, and F. Grätz. RealCaPP: Real-time capable Plug & Produce communication platform with OPC UA over TSN for distributed industrial robot control. In *2021 IEEE 17th International Conference on Automation Science and Engineering (CASE)*, pages 585–590. IEEE, 2021.
- [53] C. Eymüller, C. Wanninger, A. Hoffmann, and W. Reif. Semantic Plug and Play – Self-Descriptive Modular Hardware for Robotic Applications. *International Journal of Semantic Computing*, 12(04):559–577, 2018.
- [54] C. Eymüller, J. Hanke, A. Hoffmann, A. Poeppel, C. Wanninger, and W. Reif. Towards a Real-Time Capable Plug & Produce Environment for Adaptable Factories. In *2021 26th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA)*, pages 1–4, 2021. doi: 10.1109/ETFA45728.2021.9613729.
- [55] C. Eymüller, J. Hanke, A. Poeppel, and W. Reif. Towards Self-Configuring Plug & Produce Robot Systems Based on Ontologies. In *2023 9th International Conference on Automation, Robotics and Applications (ICARA)*, pages 23–27, 2023. doi: 10.1109/ICARA56516.2023.10126075.
- [56] D. Faggioli, F. Checconi, M. Trimarchi, and C. Scordino. An EDF scheduling class for the Linux kernel. In *Proceedings of the 11th Real-Time Linux Workshop*, pages 1–8, 2009.
- [57] X. Fan. *Real-time embedded systems: design principles and engineering practices*. Newnes, 2015.
- [58] B. Finkemeyer, T. Kröger, D. Kubus, M. Olschewski, and F. M. Wahl. MiRPA: Middleware for robotic and process control applications. In *Workshop on Measures and Procedures for the Evaluation of Robot Architectures and Middleware at the IEEE/RSJ International Conference on Intelligent Robots and Systems, San Diego, CA, USA*, pages 78–93, 2007.
- [59] B. Finkemeyer, T. Kröger, and F. M. Wahl. A middleware for high-speed distributed real-time robotic applications. In *Robotic Systems for Handling and Assembly*, pages 193–212. Springer, 2011.
- [60] P. Gerum. Xenomai-Implementing a RTOS emulation framework on GNU/Linux. *White Paper, Xenomai*, page 81, 2004.

- [61] B. Goetz and R. Eckstein. An Introduction to Real-Time Java Technology. *The Real-Time Specification for Java (JSR 1)*, 2008. URL <https://www.oracle.com/technical-resources/articles/javase/jsr-1.html>. (last visited: 20.06.2023).
- [62] I. Grangel-González, L. Halilaj, S. Auer, S. Lohmann, C. Lange, and D. Collarana. An RDF-based approach for implementing industry 4.0 components with Administration Shells. In *2016 IEEE 21st International Conference on Emerging Technologies and Factory Automation (ETFA)*, pages 1–8, 2016. doi: 10.1109/ETFA.2016.7733503.
- [63] T. R. Gruber. A translation approach to portable ontology specifications. *Knowledge Acquisition*, 5(2):199–220, 1993. ISSN 1042-8143. doi: <https://doi.org/10.1006/knac.1993.1008>. URL <https://www.sciencedirect.com/science/article/pii/S1042814383710083>.
- [64] M. Gutiérrez, A. Ademaj, W. Steiner, R. Dobrin, and S. Punnekkat. Self-configuration of IEEE 802.1 TSN networks. In *2017 22nd IEEE International Conference on Emerging Technologies and Factory Automation (ETFA)*, pages 1–8, 2017. doi: 10.1109/ETFA.2017.8247597.
- [65] C. Gärtner, A. Rizk, B. Koldehofe, R. Hark, R. Guillaume, and R. Steinmetz. Leveraging Flexibility of Time-Sensitive Networks for dynamic Reconfigurability. In *2021 IFIP Networking Conference (IFIP Networking)*, pages 1–6, 2021. doi: 10.23919/IFIPNetworking52078.2021.9472834.
- [66] C. Gärtner, A. Rizk, B. Koldehofe, R. Guillaume, R. Kundel, and R. Steinmetz. On the Incremental Reconfiguration of Time-sensitive Networks at Runtime. In *2022 IFIP Networking Conference (IFIP Networking)*, pages 1–9, 2022. doi: 10.23919/IFIPNetworking55013.2022.9829815.
- [67] M. Hankel. The Reference Architectural Model Industrie 4.0 (rami 4.0). *Zvei – German Electrical and Electronic Manufacturers’ Association*, 2(2):4–9, 2015.
- [68] Z. Hanzálek, P. Burget, and P. Šucha. Profinet IO IRT Message Scheduling. In *2009 21st Euromicro Conference on Real-Time Systems*, pages 57–65, 2009. doi: 10.1109/ECRTS.2009.18.
- [69] R. Heidel. *Industrie 4.0: The reference architecture model RAMI 4.0 and the Industrie 4.0 component*. Beuth Verlag GmbH, 2019.
- [70] G. C. Hillar. *MQTT Essentials-A lightweight IoT protocol*. Packt Publishing Ltd, 2017.
- [71] A. Hoffmann, A. Angerer, F. Ortmeier, M. Vistein, and W. Reif. Hiding real-time: A new approach for the software development of industrial robots. In *2009 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 2108–2113. IEEE, 2009.
- [72] A. Hoffmann, A. Angerer, A. Schierl, M. Vistein, and W. Reif. Service-oriented Robotics Manufacturing by reasoning about the Scene Graph of a Robotics Cell. In *ISR/Robotik 2014; 41st International Symposium on Robotics*, pages 1–8, 2014.
- [73] S. Hoppe. OPC UA - Interoperability for Industrie 4.0 and the Internet of Things. *OPC Foundation*, v7, 2018. URL <https://opcfoundation.org/wp-content/uploads/2017/11/OPC-UA-Interoperability-For-Industrie4-and-IoT-EN.pdf>. (last visited: 12.04.2023).
- [74] I. Horrocks, P. F. Patel-Schneider, H. Boley, S. Tabet, B. Grosfand, and M. Dean. *W3C recommendation*. (last visited: 01.06.2023).
- [75] I. Horrocks, P. F. Patel-Schneider, S. Bechhofer, and D. Tsarkov. OWL rules: A proposal and prototype implementation. *Journal of web semantics*, 3(1):23–40, 2005.
- [76] S. Hu and B. Yu. *Big Data Analytics for Cyber-Physical Systems*. Springer, 2020.

- [77] R. Hummen, S. Kehrer, and O. Kleineberg. TSN - Time Sensitive Networking. *Hirschmann, USA, WP00027*, 2016.
- [78] IEEE. 802.1 Time-Sensitive Networking (TSN) Task Group, 2016. URL <https://1.ieee802.org/tsn/>. (last visited: 01.04.2023).
- [79] InnoRoute GmbH. Real-Time HAT – Real-Time Communication with the Raspberry PI, . URL <https://innoroute.com/realtimehat/>. (last visited: 30.07.2023).
- [80] InnoRoute GmbH. TrustNode – Transforming Networks: Creating Networks of tomorrow, . URL [https://innoroute.com/trustnode\\_router/](https://innoroute.com/trustnode_router/). (last visited: 19.08.2023).
- [81] International Electrotechnical Commission. OPC Unified Architecture: Overview and Concepts. *IEC62541*, 2010.
- [82] International Electrotechnical Commission. IEC 61131-3:2013 : Programmable controllers - Part 3: Programming languages. *Measurement and control devices*, 3.0, 2013.
- [83] International Electrotechnical Commission. IEC 61158-1: Industrial communication networks – Fieldbus specifications – Part 1: Overview and guidance for the IEC 61158 and IEC 61784 series. *Digital Data Communications for Measurement and Control—Fieldbus for Use in Industrial Control Systems*, 1, 2014.
- [84] IO-Link Community. IO-Link System Description – Technology and Application, 2018. URL [https://io-link.com/share/Downloads/At-a-glance/IO-Link\\_System\\_Description\\_eng\\_2018.pdf](https://io-link.com/share/Downloads/At-a-glance/IO-Link_System_Description_eng_2018.pdf). (last visited: 17.07.2023).
- [85] ISO Central Secretary. ISO 8373:1994: Manipulating industrial robots – Vocabulary. Standard ISO 8373:1994, International Organization for Standardization, 1994. URL <https://www.iso.org/standard/15532.html>.
- [86] ISO Central Secretary. ISO/IEC TS 23619:2021: Information technology – C++ extensions for reflection. Standard ISO/IEC TS 23619:2021, International Organization for Standardization, 2021. URL <https://www.iso.org/standard/76425.html>.
- [87] C. Jang, S.-I. Lee, S.-W. Jung, B. Song, R. Kim, S. Kim, and C.-H. Lee. OPRoS: A New Component-Based Robot Software Platform. *ETRI journal*, 32(5):646–656, 2010.
- [88] C. Jang, B. Song, S. Jung, K.-H. Lee, and S. Kim. Real-time supporting of OPRoS component Platform. In *2011 8th International Conference on Ubiquitous Robots and Ambient Intelligence (URAI)*, pages 640–641, 2011. doi: 10.1109/URAI.2011.6145899.
- [89] E. Järvenpää, N. Siltala, O. Hylli, and M. Lanz. The development of an ontology for describing the capabilities of manufacturing resources. *Journal of Intelligent Manufacturing*, 30(2):959–978, 2019. doi: 10.1007/s10845-018-1427-6.
- [90] E. Järvenpää, N. Siltala, O. Hylli, H. Nylund, and M. Lanz. Semantic rules for capability matchmaking in the context of manufacturing system design and reconfiguration. *International Journal of Computer Integrated Manufacturing*, 36(1):128–154, 2023. doi: 10.1080/0951192X.2022.2081361.
- [91] F. Kanehiro, Y. Ishiwata, H. Saito, K. Akachi, G. Miyamori, T. Isozumi, K. Kaneko, and H. Hirukawa. Distributed Control System of Humanoid Robots based on Real-time Ethernet. In *2006 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 2471–2477, 2006. doi: 10.1109/IROS.2006.281691.
- [92] S. Kanno, J. Hermann, M. Damm, P. Rübél, D. Rusin, M. Jacobi, B. Mittelsdorf, T. Kuhn, and P. O. Antonino. Enabling SMEs to industry 4.0 using the BaSyx middleware: A



- case study. In *Software Architecture: 15th European Conference, ECSA 2021, Virtual Event, Sweden, September 13-17, 2021, Proceedings*, pages 277–294. Springer, 2021.
- [93] M. Kaspar, J. Bock, Y. Kogan, P. Venet, M. Weser, and U. E. Zimmermann. Tool and technology independent function interfaces by using a generic OPC UA representation. In *2018 IEEE 23rd International Conference on Emerging Technologies and Factory Automation (ETFA)*, volume 1, pages 1183–1186. IEEE, 2018.
- [94] J. Kay. Introduction to Real-time Systems, 2016. URL [https://design.ros2.org/articles/realtime\\_background.html](https://design.ros2.org/articles/realtime_background.html). (last visited: 08.08.2023).
- [95] J. Klech. Tuning Guide – Advanced tuning procedures to optimize latency in RHEL for Real Time. URL [https://access.redhat.com/documentation/de/red\\_hat\\_enterprise\\_linux\\_for\\_real\\_time/7/html/tuning\\_guide/isolating\\_cpus\\_using\\_tuned-profiles-realtime](https://access.redhat.com/documentation/de/red_hat_enterprise_linux_for_real_time/7/html/tuning_guide/isolating_cpus_using_tuned-profiles-realtime). (last visited: 08.08.2023).
- [96] A. Köcher, C. Hildebrandt, L. M. V. da Silva, and A. Fay. A formal capability and skill model for use in plug and produce scenarios. In *2020 25th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA)*, volume 1, pages 1663–1670. IEEE, 2020.
- [97] H. Kopetz and W. Steiner. *Real-Time Communication*. Springer, 2022.
- [98] C. Kormanyos. *Real-time C++: efficient object-oriented and template microcontroller programming*. Springer, 2013.
- [99] H. Koziulek, A. Burger, M. Platenius-Mohr, J. Rückert, and G. Stomberg. OpenPnP: A Plug-and-Produce Architecture for the Industrial Internet of Things. In *2019 IEEE/ACM 41st International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*, pages 131–140, 2019. doi: 10.1109/ICSE-SEIP.2019.00022.
- [100] P. Kremen, M. Smid, and Z. Kouba. OWLDiff: A practical tool for comparison and merge of OWL ontologies. In *2011 22nd International Workshop on Database and Expert Systems Applications*, pages 229–233. IEEE, 2011.
- [101] KUKA Deutschland GmbH. KUKA.EtherNet KRL – The control system of the future KR C4, . URL [https://www.kuka.com/-/media/kuka-downloads/imported/87f2706ce77c4318877932fb36f6002d/kuka\\_pb\\_controllers\\_en.pdf?rev=e8d133f6ddc64b3cb5329645eea0cc8e&hash=90B1C2377022BEC4AA9EAF67B28F5735](https://www.kuka.com/-/media/kuka-downloads/imported/87f2706ce77c4318877932fb36f6002d/kuka_pb_controllers_en.pdf?rev=e8d133f6ddc64b3cb5329645eea0cc8e&hash=90B1C2377022BEC4AA9EAF67B28F5735). (last visited: 02.08.2023).
- [102] KUKA Deutschland GmbH. KKUKA KR C4: The Power of Control, . URL <https://www.kuka.com/en-de/products/robot-systems/robot-controllers/kr%c2%a0c4>. (last visited: 02.08.2023).
- [103] KUKA Deutschland GmbH. KRL – Application and robot programming, . URL <https://www.kuka.com/en-de/services/engineering/application-and-robot-programming>. (last visited: 02.08.2023).
- [104] KUKA Deutschland GmbH. KUKA.RobotSensorInterface – Simplifies challenging sensor applications, . URL [https://www.kuka.com/en-de/products/robot-systems/software/application-software/kuka\\_robotsensorinterface](https://www.kuka.com/en-de/products/robot-systems/software/application-software/kuka_robotsensorinterface). (last visited: 02.08.2023).
- [105] KUKA Deutschland GmbH. Linear Unit KL 4000, . URL <https://www.kuka.com/en-de/products/robot-systems/robot-periphery/linear-units/kl-4000>. (last visited: 15.07.2023).

- [106] KUKA Deutschland GmbH. KR 10 R900-2, . URL [https://www.kuka.com/-/media/kuka-downloads/imported/8350ff3ca11642998dbdc81dcc2ed44c/0000290002\\_de.pdf](https://www.kuka.com/-/media/kuka-downloads/imported/8350ff3ca11642998dbdc81dcc2ed44c/0000290002_de.pdf). (last visited: 15.07.2023).
- [107] KUKA Deutschland GmbH. KR 6 R900 sixx, . URL [https://www.kuka.com/-/media/kuka-downloads/imported/8350ff3ca11642998dbdc81dcc2ed44c/0000205456\\_de.pdf](https://www.kuka.com/-/media/kuka-downloads/imported/8350ff3ca11642998dbdc81dcc2ed44c/0000205456_de.pdf). (last visited: 18.07.2023).
- [108] KUKA Deutschland GmbH. KR 90 R3700 prime K, . URL [https://www.kuka.com/-/media/kuka-downloads/imported/8350ff3ca11642998dbdc81dcc2ed44c/0000189663\\_en.pdf](https://www.kuka.com/-/media/kuka-downloads/imported/8350ff3ca11642998dbdc81dcc2ed44c/0000189663_en.pdf). (last visited: 15.07.2023).
- [109] S. Lankes, A. Jabs, and M. Reke. A time-triggered Ethernet protocol for real-time CORBA. In *Proceedings Fifth IEEE International Symposium on Object-Oriented Real-Time Distributed Computing. ISIRC 2002*, pages 215–222. IEEE, 2002.
- [110] S. M. LaValle. *Planning algorithms*. Cambridge university press, 2006. ISBN 9780511546877. doi: 10.1017/CBO9780511546877.
- [111] A. Lawan. *Ontology-based knowledge representation and semantic search information retrieval: case study of the underutilized crops domain*. PhD thesis, University of Nottingham, 2018.
- [112] S. Lemaignan, A. Siadat, J.-Y. Dantan, and A. Semenenko. MASON: A Proposal For An Ontology Of Manufacturing Domain. In *IEEE Workshop on Distributed Intelligent Systems: Collective Intelligence and Its Applications (DIS'06)*, pages 195–200, 2006. doi: 10.1109/DIS.2006.48.
- [113] H. Lingyan, P. Jie, and X. Yong. Experimental Research about the Impact of IEEE 802.1P on Real-Time Behavior of Switched Industrial Ethernet. In *2008 ISECS International Colloquium on Computing, Communication, Control, and Management*, volume 2, pages 403–406, 2008. doi: 10.1109/CCCM.2008.117.
- [114] Linux manual page. `clock_gettime()` – clock and time functions. URL [https://man7.org/linux/man-pages/man3/clock\\_gettime.3.html](https://man7.org/linux/man-pages/man3/clock_gettime.3.html). (last visited: 19.08.2023).
- [115] L. Liu and M. T. Özsu. *Encyclopedia of database systems*, volume 6. Springer, 2009.
- [116] S. Macenski, T. Foote, B. Gerkey, C. Lalancette, and W. Woodall. Robot Operating System 2: Design, architecture, and uses in the wild. *Science Robotics*, 7(66), 2022. doi: 10.1126/scirobotics.abm6074.
- [117] B. Madiwalar, B. Schneider, and S. Profanter. Plug and Produce for Industry 4.0 using Software-defined Networking and OPC UA. In *2019 24th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA)*, pages 126–133. IEEE, 2019.
- [118] R. Mall. *Real-time systems: theory and practice*. Pearson Education India, 2009.
- [119] F. Manola, E. Miller, B. McBride, et al. RDF primer. *W3C recommendation*, 10(1-107):6, 2004.
- [120] P. Mantegazza, E. Dozio, and S. Papacharalambous. RTAI: Real time application interface. *Linux Journal*, 2000(72es):10, 2000.
- [121] E. A. Marks and M. Bell. *Service-oriented architecture: a planning and implementation guide for business and technology*. John Wiley & Sons, 2008.
- [122] J. McAffer, P. VanderLei, and S. Archer. *OSGi and Equinox: Creating highly modular Java systems*. Addison-Wesley Professional, 2010.

- [123] G. McCluskey. Using Java reflection. URL <https://www.oracle.com/technical-resources/articles/java/javareflection.html>. (last visited: 16.06.2023).
- [124] D. L. McGuinness, F. Van Harmelen, et al. OWL web ontology language overview. *W3C recommendation*, 10, 2004.
- [125] ME-Meßsysteme GmbH. GSV-8DS – 8-channel strain gauge amplifier in aluminum housing, . URL <https://www.me-systeme.de/shop/en/electronics/gsv-8-analog-digital/gsv-8ds/gsv-8ds>. (last visited: 31.07.2023).
- [126] ME-Meßsysteme GmbH. K6D80 – 6-axis force-torque sensor, . URL <https://www.me-systeme.de/shop/en/sensors/force-sensors/k6d/k6d803>. (last visited: 31.07.2023).
- [127] G. Miao, J. Zander, K. W. Sung, and S. B. Slimane. *Fundamentals of mobile data networks*. Cambridge University Press, 2016.
- [128] L. Monostori. Cyber-physical production systems: Roots, expectations and R&D challenges. *Procedia Cirp*, 17:9–13, 2014.
- [129] B. Motik, R. Shearer, and I. Horrocks. Optimized reasoning in description logics using hypertableaux. In *Automated Deduction–CADE-21: 21st International Conference on Automated Deduction Bremen, Germany, July 17-20, 2007 Proceedings 21*, pages 67–83. Springer, 2007.
- [130] L. Nägele. *PaRTs: automatische Programmierung in der robotergestützten Fertigung*. doctoralthesis, Universität Augsburg, 2021.
- [131] OMG: Object Management Group. CORBA – Common Object Request Broker Architecture, Version 2.2. Standard, Standards Development Organization, 1998. URL <https://www.omg.org/spec/CORBA/2.2/About-CORBA>.
- [132] OMG: Object Management Group. Data Distribution Service for Real-time Systems, Version 1.2. Standard, Standards Development Organization, 2007. URL <https://www.omg.org/spec/CORBA/2.2/About-CORBA>.
- [133] OMG: Object Management Group. RTC - Robitc Technology Component Version 1.1. Standard, Standards Development Organization, 2012. URL <https://www.omg.org/spec/RTC>.
- [134] M. Onori, N. Lohse, J. Barata, and C. Hanisch. The IDEAS project: plug & produce at shop-floor level. *Assembly automation*, 32(2):124–134, 2012.
- [135] OPC Foundation. *OPC Unified Architecture Part 1: Overview and Concepts*. IEC62541, 2017.
- [136] OPC Foundation. *OPC Unified Architecture Part 14: PubSub*. IEC62541, 2018.
- [137] OPC Foundation. *OPC 40010-10 – OPC UA Companion-Specification – OPC UA for Robotics Part 1: Vertical Integration*. VDMA, 2019.
- [138] OPC Foundation. *OPC Unified Architecture Part 10: Programs*. IEC62541, 2021.
- [139] OPC Foundation. *OPC Unified Architecture Part 4: Services*. IEC62541, 2021.
- [140] OPC Foundation. *OPC Unified Architecture Part 12: Discovery and Global Services*. IEC62541, 2022.
- [141] OPC Foundation. *OPC Unified Architecture Part 5: Information Model*. IEC62541, 2022.
- [142] OPC Foundation. *OPC Unified Architecture Part 6: Mappings*. IEC62541, 2022.
- [143] OPC Foundation. Unified Architecture, 2023. URL <https://opcfoundation.org/about/opc-technologies/opc-ua/>. (last visited: 12.04.2023).

- [144] OROCOS. Kinematics and Dynamics Library (KDL), 2023. URL <https://www.orocos.org/kdl.html>. (last visited: 11.07.2023).
- [145] OROCOS. The OrocOS Real-Time Toolkit (RTT), 2023. URL <https://www.orocos.org/rtt/>. (last visited: 11.07.2023).
- [146] OSGi Alliance. OSGi Core Release 8. URL <https://docs.osgi.org/specification/osgi.core/8.0.0/framework/introduction.html>. (last visited: 18.06.2023).
- [147] J. Otto and O. Niggemann. Automatic parameterization of automation software for plug-and-produce. In *Workshops at the Twenty-Ninth AAAI Conference on Artificial Intelligence*, 2015.
- [148] F. Palm and U. Epple. openAAS – Die offene Entwicklung der Verwaltungsschale. *Automation 2017: Technology Networks Processes*, 2293:103–104, 2017.
- [149] Y. Pane, M. H. Arbo, E. Aertbeliën, and W. Decré. A System Architecture for CAD-Based Robotic Assembly With Sensor-Based Skills. *IEEE Transactions on Automation Science and Engineering*, 17(3):1237–1249, 2020. doi: 10.1109/TASE.2020.2980628.
- [150] G. Pardo-Castellote. OMG Data-Distribution Service: architectural overview. In *23rd International Conference on Distributed Computing Systems Workshops, 2003. Proceedings.*, pages 200–206, 2003. doi: 10.1109/ICDCSW.2003.1203555.
- [151] Pepperl+Fuchs GmbH. IO-Link-Master (EIP/MOD) ICE2-8IOL-K45P-RJ45. URL [https://www.pepperl-fuchs.com/germany/de/classid\\_4996.htm?view=productdetails&prodid=97763](https://www.pepperl-fuchs.com/germany/de/classid_4996.htm?view=productdetails&prodid=97763). (last visited: 30.07.2023).
- [152] A. Perzylo, N. Somani, M. Rickert, and A. Knoll. An ontology for CAD data and geometric constraints as a link between product models and semantic robot task descriptions. In *2015 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 4197–4203, 2015. doi: 10.1109/IROS.2015.7353971.
- [153] A. Perzylo, J. Grothoff, L. Lucio, M. Weser, S. Malakuti, P. Venet, V. Aravantinos, and T. Deppe. Capability-based semantic interoperability of manufacturing resources: A BaSys 4.0 perspective. *IFAC-PapersOnLine*, 52(13):1590–1596, 2019.
- [154] Peter Soetens. Distributing OrocOS Components with CORBA, 2023. URL <https://www.orocos.org/stable/documentation/rtt/v2.x/doc-xml/orocos-transport-corba.pdf>. (last visited: 11.07.2023).
- [155] J. Pfrommer. open62541 – Open Source OPC UA. URL <https://www.open62541.org/#>. (last visited: 22.08.2023).
- [156] J. Pfrommer, M. Schleipen, and J. Beyerer. Pprs: Production skills and their relation to product, process, and resource. In *2013 IEEE 18th Conference on Emerging Technologies & Factory Automation (ETFA)*, pages 1–4. IEEE, 2013.
- [157] J. Pfrommer, D. Stogl, K. Aleksandrov, S. Escalda Navarro, B. Hein, and J. Beyerer. Plug & produce by modelling skills and service-oriented orchestration of reconfigurable manufacturing systems. *at-Automatisierungstechnik*, 63(10):790–800, 2015.
- [158] J. Pfrommer, A. Ebner, S. Ravikumar, and B. Karunakaran. Open source OPC UA PubSub over TSN for realtime industrial communication. In *2018 IEEE 23rd international conference on emerging technologies and factory automation (ETFA)*, volume 1, pages 1087–1090. IEEE, 2018.
- [159] H. S. Pinto, A. Gómez-Pérez, and J. P. Martins. Some issues on ontology integration. *IJCAI and the Scandinavian AI Societies. CEUR Workshop Proceedings*, 1999.

- [160] Plattform Industrie 4.0. Usage View of Asset Administration Shell. 2019.
- [161] P. Pop, M. L. Raagaard, M. Gutierrez, and W. Steiner. Enabling fog computing for industrial automation through time-sensitive networking (TSN). *IEEE Communications Standards Magazine*, 2(2):55–61, 2018.
- [162] L. Prenzel, A. Zoitl, and J. Provost. Iec 61499 runtime environments: A state of the art comparison. In *Computer Aided Systems Theory–EUROCAST 2019: 17th International Conference, Las Palmas de Gran Canaria, Spain, February 17–22, 2019, Revised Selected Papers, Part II 17*, pages 453–460. Springer, 2020.
- [163] L. Prenzel, S. Hofmann, and S. Steinhorst. Real-time dynamic reconfiguration for IEC 61499. In *2022 IEEE 5th International Conference on Industrial Cyber-Physical Systems (ICPS)*, pages 1–6. IEEE, 2022.
- [164] F. Prinz, M. Schoeffler, A. Lechler, and A. Verl. End-to-end Redundancy between Real-time I4.0 Components based on Time-Sensitive Networking. In *2018 IEEE 23rd International Conference on Emerging Technologies and Factory Automation (ETFA)*, volume 1, pages 1083–1086, 2018. doi: 10.1109/ETFA.2018.8502553.
- [165] F. Prinz, M. Schoeffler, A. Lechler, and A. Verl. Dynamic Real-time Orchestration of I4.0 Components based on Time-Sensitive Networking. *Procedia CIRP*, 72:910–915, 2018. ISSN 2212-8271. doi: <https://doi.org/10.1016/j.procir.2018.03.174>. URL <https://www.sciencedirect.com/science/article/pii/S2212827118303329>. 51st CIRP Conference on Manufacturing Systems.
- [166] M. Priya and C. A. Kumar. A survey of state of the art of ontology construction and merging using formal concept analysis. *Indian journal of science and technology*, 8(24): 1–7, 2015.
- [167] S. Profanter, K. Dorofeev, A. Zoitl, and A. Knoll. OPC UA for plug & produce: Automatic device discovery using LDS-ME. In *2017 22nd IEEE International Conference on Emerging Technologies and Factory Automation (ETFA)*, pages 1–8, 2017. doi: 10.1109/ETFA.2017.8247569.
- [168] S. Profanter, A. Breitzkreuz, M. Rickert, and A. Knoll. A hardware-agnostic OPC UA skill model for robot manipulators and tools. In *2019 24th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA)*, pages 1061–1068. IEEE, 2019.
- [169] S. Profanter, A. Perzylo, M. Rickert, and A. Knoll. A Generic Plug & Produce System Composed of Semantic OPC UA Skills. *IEEE Open Journal of the Industrial Electronics Society*, 2:128–141, 2021. doi: 10.1109/OJIES.2021.3055461.
- [170] S. Profanter, A. Perzylo, M. Rickert, and A. Knoll. A Generic Plug & Produce System Composed of Semantic OPC UA Skills. *IEEE Open Journal of the Industrial Electronics Society*, 2:128–141, 2021.
- [171] Profitec GmbH. Pro 800 – Hand Level Dipping System with PID Control. URL <https://www.profittec-espresso.com/en/products/pro800-neu>. (last visited: 18.07.2023).
- [172] E. Prud’hommeaux, S. Harris, and A. Seaborne. SPARQL 1.1 Query Language. *W3C recommendation*, 2013. URL <http://www.w3.org/TR/sparql11-query>. (last visited: 17.05.2023).
- [173] L. Puck, P. Keller, T. Schnell, C. Plasberg, A. Tanev, G. Heppner, A. Roennau, and R. Dillmann. Performance evaluation of real-time ROS2 robotic control in a time-synchronized distributed network. In *2021 IEEE 17th International Conference on Automation Science and Engineering (CASE)*, pages 1670–1676. IEEE, 2021.

- [174] QNX (Blackberry). Qnet – Native Networking, . URL [https://www.qnx.com/developers/docs/7.0.0/#com.qnx.doc.neutrino.sys\\_arch/topic/qnet.html](https://www.qnx.com/developers/docs/7.0.0/#com.qnx.doc.neutrino.sys_arch/topic/qnet.html). (last visited: 12.07.2023).
- [175] QNX (Blackberry). QNX Neutrino Real-Time Operating System (RTOS), . URL <https://blackberry.qnx.com/en/products/foundation-software/qnx-rtos>. (last visited: 12.07.2023).
- [176] M. Quigley, K. Conley, B. Gerkey, J. Faust, T. Foote, J. Leibs, R. Wheeler, A. Y. Ng, et al. ROS: an open-source Robot Operating System. In *ICRA workshop on open source software*, volume 3, page 5. Kobe, Japan, 2009.
- [177] Raspberry Pi Foundation. Raspberry Pi 4 – Your tiny, dual-display, desktop computer. URL <https://www.raspberrypi.com/products/raspberry-pi-4-model-b/>. (last visited: 30.07.2023).
- [178] Real-Time Innovations (RTI). Connex Professional DDS. URL <https://www.rti.com/products/connex-dds-professional>. (last visited: 12.07.2023).
- [179] F. Reghenzani, G. Massari, and W. Fornaciari. The Real-Time Linux Kernel: A Survey on PREEMPT-RT. *ACM Comput. Surv.*, 52(1), feb 2019. doi: 10.1145/3297714.
- [180] T. Richardson and A. J. Wellings. RT-OSGi: Integrating the OSGi framework with the real-time specification for Java. *Distributed, Embedded and Real-time Java Systems*, pages 293–322, 2012.
- [181] J. D. Rojas, O. Arrieta, and R. Vilanova. *Industrial PID controller tuning*. Springer, 2021.
- [182] E. Romiti, J. Malzahn, N. Kashiri, F. Iacobelli, M. Ruzzon, A. Laurenzi, E. M. Hoffman, L. Muratore, A. Margan, L. Baccelliere, S. Cordasco, and N. Tsagarakis. Toward a Plug-and-Work Reconfigurable Cobot. *IEEE/ASME Transactions on Mechatronics*, pages 1–13, 2021. doi: 10.1109/TMECH.2021.3106043.
- [183] Ros.org. ROS ActionLib. URL <http://wiki.ros.org/actionlib>. (last visited: 14.07.2023).
- [184] M. Sadiku, Y. Wang, S. Cui, and S. M. Musa. Cyber-physical systems: A literature review. *European Scientific Journal*, 13(36):52–58, 2017.
- [185] S. Salmon. How to make C++ more real-time friendly, 2014. URL <https://www.embedded.com/how-to-make-c-more-real-time-friendly/>. (last visited: 08.08.2023).
- [186] K. Sandkuhl, H. Koç, and J. Stirna. Capability-as-a-service: Towards context aware business services. In *2014 IEEE 18th International Enterprise Distributed Object Computing Conference Workshops and Demonstrations*, pages 324–332. IEEE, 2014.
- [187] M. Schleipen, A. Lüder, O. Sauer, H. Flatt, and J. Jasperneite. Requirements and concept for plug-and-work. *at-Automatisierungstechnik*, 63(10):801–820, 2015.
- [188] Schmalz GmbH. CobotPump ECBPI – Intelligent electrical vacuum generator for handling airtight and slightly porous workpieces. URL <https://www.schmalz.com/en-us/vacuum-technology-for-automation/vacuum-components/vacuum-generators/vacuum-generators-end-of-arm/vacuum-generators-ecbpi-308294/>. (last visited: 31.07.2023).
- [189] K. Schweichhart. Reference Architectural Model Industrie 4.0 (RAMI 4.0). *Plattform Industrie 4.0*, 40, 2016.
- [190] A. Seaborne. SPARQL 1.1 Property Paths. *W3C recommendation*, 2010. URL <https://www.w3.org/TR/sparql11-property-paths/>. (last visited: 04.06.2023).

- [191] N. Siltala, E. Järvenpää, and M. Lanz. Creating resource combinations based on formally described hardware interfaces. In S. Ratchev, editor, *Precision Assembly in the Digital Age*, pages 29–39, Cham, 2019. Springer International Publishing.
- [192] N. Siltala, E. Järvenpää, and M. Lanz. A method to evaluate interface compatibility during production system design and reconfiguration. *Procedia CIRP*, 81:282–287, 2019. ISSN 2212-8271. doi: <https://doi.org/10.1016/j.procir.2019.03.049>. 52nd CIRP Conference on Manufacturing Systems (CMS), Ljubljana, Slovenia, June 12-14, 2019.
- [193] E. Sirin, B. Parsia, B. C. Grau, A. Kalyanpur, and Y. Katz. Pellet: A practical OWL-DL reasoner. *Journal of Web Semantics*, 5(2):51–53, 2007.
- [194] Standardization International. ISO/IEC 7498-1: 1994 information technology–open systems interconnection–basic reference model: The basic model. *International Standard ISO/IEC*, 74981:59, 1996.
- [195] Standardization International. IEC 61131-9:2022 – Programmable controllers - Part 9: Single-drop digital communication interface for small sensors and actuators (SDCI). Standard IEC 61131-9:2022, International Electrotechnical Commission, 2022. URL <https://webstore.iec.ch/publication/68534>.
- [196] Stefan Holzer Feinmechanik e.K. HTC 180 – Holzer Robotic Tool Changer. URL [https://www.holzer-feinmechanik.de/media/files/flyer\\_holzer\\_htc300\\_englisch.pdf](https://www.holzer-feinmechanik.de/media/files/flyer_holzer_htc300_englisch.pdf). (last visited: 31.07.2023).
- [197] Stoeger Automation GmbH. SPATZ - STÖGER Pick&Place screwdriving robot with automatic tool change and feed unit for fasteners. URL <https://www.stoeger.com/en/screwdriving-robot-with-automatic-tool-change.html>. (last visited: 21.07.2023).
- [198] G. Stumme and A. Maedche. FCA-Merge: Bottom-up merging of ontologies. In *IJCAI*, volume 1, pages 225–230, 2001.
- [199] W. M. Taha, A.-E. M. Taha, and J. Thunberg. *Cyber-Physical Systems: A Model-Based Approach*. Springer Nature, 2021.
- [200] A. S. Tanenbaum. *Distributed Systems: Principles and paradigms*. 2007.
- [201] A. L. Tavares and M. T. Valente. A gentle introduction to OSGi. *ACM SIGSOFT Software Engineering Notes*, 33(5):1–5, 2008.
- [202] E. Trunzer, A. Calà, P. Leitão, M. Gepp, J. Kinghorst, A. Lüder, H. Schauerte, M. Reifferscheid, and B. Vogel-Heuser. System architectures for industrie 4.0 applications: Derivation of a generic architecture proposal. *Production Engineering*, 13:247–257, 2019.
- [203] M. Vistein, A. Angerer, A. Hoffmann, A. Schierl, and W. Reif. Interfacing industrial robots using realtime primitives. In *2010 IEEE International Conference on Automation and Logistics*, pages 468–473. IEEE, 2010.
- [204] J. Walter, K. Grüttner, and W. Nebel. Using IEC 61499 and OPC-UA to implement a self-organising plug and produce system. In *The 5th International Workshop on Model-driven Robot Software Engineering (MORSE 2018)*, 2018.
- [205] K. Wang and K. Wang. *Embedded real-time operating systems*. Springer, 2017.
- [206] Weiss Robotics GmbH & Co. KG. CRG 200 – Servo-electric Gripping Module. URL <https://weiss-robotics.com/servo-electric/crg-series/product/crg-serie-495/>. (last visited: 31.07.2023).

- [207] X. Ye, J. Jiang, C. Lee, N. Kim, M. Yu, and S. H. Hong. Toward the Plug-and-Produce Capability for Industry 4.0: An Asset Administration Shell Approach. *IEEE Industrial Electronics Magazine*, 14(4):146–157, 2020. doi: 10.1109/MIE.2020.3010492.
- [208] V. Yodaiken et al. The RTLinux manifesto. In *Proc. of the 5th Linux Expo*, 1999.
- [209] H. Zeng, W. Zheng, M. Di Natale, A. Ghosal, P. Giusto, and A. Sangiovanni-Vincentelli. Scheduling the flexray bus using optimization techniques. In *Proceedings of the 46th Annual Design Automation Conference*, pages 874–877, 2009.
- [210] Zimmer Group GmbH. FWR50 – Automated tool change without external activation, . URL <https://www.zimmer-group.com/en/technologien-komponenten/komponenten/handhabungstechnik/werkzeugwechsler/automatisch/serie-fwr/produkte/fwr50f-00-a>. (last visited: 31.07.2023).
- [211] Zimmer Group GmbH. GEP2016IL-03-B – 2-Jaw Parallel Gripper, . URL <https://www.zimmer-group.com/en/technologies-components/components/handling-technology/grippers/electric/2-jaw-parallel-grippers/series-gep2000/products/gep2016il-03-b>. (last visited: 31.07.2023).
- [212] Zimmer Group GmbH. GEH6180IL-03-B – 2-Jaw Parallel Grippers with Long Stroke, . URL <https://www.zimmer-group.com/en/technologies-components/components/handling-technology/grippers/electric/2-jaw-parallel-grippers-with-long-stroke/series-geh6000il/products/geh6180il-03-b>. (last visited: 30.07.2023).
- [213] P. Zimmermann, E. Axmann, B. Brandenbourger, K. Dorofeev, A. Mankowski, and P. Zanini. Skill-Based Engineering and Control on Field-Device-Level with OPC UA. In *2019 24th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA)*, pages 1101–1108. IEEE, 2019.
- [214] A. Zoitl and V. Vyatkin. IEC 61499 architecture for distributed automation. *IEEE Industrial Electronics Magazine*, 3(4):7–23, 2009.
- [215] C. Zunino, D. Malena, G. Cena, S. Scanzio, and A. Valenzano. Black-Box Analysis of the Publish-Subscribe Notification Latency in Real OPC UA Servers. In *2021 26th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA)*, pages 1–4, 2021. doi: 10.1109/ETFA45728.2021.9613259.



# List of Figures

1.1	Research questions with an example . . . . .	3
1.2	Structure and interrelations between the main components of the real-time capable Plug & Produce architecture. The gray boxes indi- cate the chapters in which the individual components are described. . . . . .	6
2.1	The Reference Architecture Model Industry 4.0 (RAMI 4.0) [189] . .	11
3.1	Basic components of a flexible robot cell . . . . .	22
3.2	Possible execution sequence of the exemplary task sequence . . . .	25
3.3	Industrial high flexible robot cell . . . . .	26
3.4	A robot cell for making coffee . . . . .	27
4.1	Structure of the Asset Administration Shell (Adapted from [22]) . .	31
4.2	OPC UA system architecture for the combination of communication interfaces and the information model (Adapted from [143]) . . . . .	32
4.3	OPC UA Client-Server interaction (Adapted from [135]) . . . . .	33
4.4	TSN time division multiplexing with reserved time slots to enable the transmission of periodic real-time data . . . . .	35
4.5	System architecture for a real-time capable Plug & Produce envi- ronment using OPC UA and OPC UA over TSN as middleware. (See Eymüller et al. [54]) . . . . .	37
5.1	System architecture for a RealCaPP environment focusing on the description of the resources, the discovery via OPC UA and the global knowledge bases (registries) (cf. Figure 4.5) . . . . .	44
5.2	Example of an RDF-Graph . . . . .	45
5.3	Deployment diagram of an active Plug & Produce resource . . . . .	47
5.4	Deployment diagram of a system with an passive Plug & Produce resource in combination with another hardware device . . . . .	48
5.5	Component diagram of the global registry . . . . .	49
5.6	Sequence diagram of adding a resource . . . . .	50
5.7	Sequence diagram of getting resource information of the new resources	50
5.8	RDF-graph of the basic class structure of the RealCaPP ontology . .	52
5.9	Distributed knowledge base: separation of instantiation from the class structure. . . . .	53

6.1	System architecture for a RealCaPP environment focusing on the description of the resources and the knowledge base of the skill registry (cf. Figure 4.5) . . . . .	58
6.2	Ontology for describing resources and their interrelationships. . . . .	61
6.3	Simplified example of a connected resource with the different interface descriptions . . . . .	63
6.4	Example for a subgraph with the required skills screw and move to a position . . . . .	69
7.1	System architecture for a RealCaPP environment focusing on the description of the communication middleware with OPC UA and OPC UA over TSN (cf. Figure 4.5) . . . . .	74
7.2	TSN time division multiplexing with reserved time slots to enable the transmission of periodic real-time data . . . . .	75
7.3	TSN Time-Aware Scheduler with the realization of the Time-Aware Gate Control . . . . .	75
7.4	The different TSN configuration approaches (Adapted from [77]) . . . . .	76
7.5	PTP synchronization procedure (Adapted from [10]) . . . . .	78
7.6	Exemplary structure of a dynamic real-time control network with four TSN End Devices . . . . .	80
7.7	Activity diagram of the management of TSN End Devices by the TSN Controller (Adapted from [52]) . . . . .	82
7.8	Example of a time slot array for one TSN End Device [52] . . . . .	83
8.1	System architecture for a RealCaPP environment focusing on the distributed real-time execution of modular software components (cf. Figure 4.5) . . . . .	88
8.2	OSGi System Layers [146] . . . . .	89
8.3	Standard OPC UA Program finite state machine with four default states and nine default transitions. (Adapted from [138]) . . . . .	91
8.4	Structure of the DataContainer . . . . .	95
8.5	Class diagram of the class DataContainer . . . . .	96
8.6	A single Real-Time Service (RTS) with InPorts, OutPorts and Functions . . . . .	97
8.8	Realization of external connections. One RTS Network is distributed to two devices. . . . .	100
8.9	Class diagram of Real-Time Services (RTSs) . . . . .	101
8.10	Class diagram of the class ApplicationService . . . . .	103
8.11	Cube example ontology with a resource configuration that can grip cubes . . . . .	104
8.12	Overview of all hierarchy levels for the execution of processes in the RealCaPP architecture . . . . .	105
8.13	Excerpt from the ontology describing RTSs and RTS Networks . . . . .	108
8.14	Application example for distributed OPC UA Programs with one Master and one Slave: Distributed execution of a trajectory on a robot arm (See Eymüller et al. [51]) . . . . .	111

8.15	Distributed OPC UA Programs finite state machine [51] . . . . .	112
8.16	Component diagram of a distributed Master OPC UA Program. The structure of the Slave is identical (See Eymüller et al. [51]) . . . . .	114
8.17	Example for executing distributed RTS Networks with distributed OPC UA Programs . . . . .	114
9.1	System architecture for a RealCaPP environment focusing on the the hardware drivers and the Basic Skills (cf. Figure 4.5) . . . . .	120
9.2	Blending motion of two linear motions from A to B and B to C with the blend radius $r$ . . . . .	122
9.3	Abstract RTS of an industrial robot . . . . .	124
9.4	Abstract RTS of a force-torque sensor . . . . .	125
9.5	Abstract RTS of a digital input module . . . . .	126
9.6	Abstract RTS of a basic gripper . . . . .	127
9.7	Specialization of gripper controls by inheritance of abstract basic gripper RTS . . . . .	128
9.8	Abstract RTS of a basic screwer . . . . .	130
9.9	Specialization of screwer controls by inheritance of abstract basic screwer RTS . . . . .	131
9.10	Abstract RTS of a basic tool changer . . . . .	132
9.11	Abstract RTS of a digital output module . . . . .	133
10.1	Flexible industrial robot cell of the WiR innovation laboratory with two robots, additional production systems, and processing stations . . . . .	137
10.2	Structure of the complete end effector of the KUKA KR90. A six-axis force-torque sensor is attached to the robot flange, to which the automatic tool changer is attached. Different tools can be attached to the tool changer via tool plates. . . . .	138
10.3	Processing station with an additional robot, active clamping units, and the CP Lab production system . . . . .	139
10.4	Robarista robot cell with an industrial robot and a hand lever portafilter espresso machine and the other equipment needed for the preparation of coffee with the robot . . . . .	140
10.5	Abridged presentation of the Basic Skill of a KUKA KR robot as RTS implementation . . . . .	142
10.6	Structure of the GEH6180 gripper with tool adapter and IO-Link master for use on the KR90 . . . . .	143
10.7	Composite structure diagram of the gripper for the active and passive resource implementation. Structures located on the tool side are grayed out. The RealCaPP capable OPC UA over TSN interfaces are marked orange. . . . .	144
10.8	Raspberry Pi 4 with Real-Time Hat extension board for TSN capability on two ethernet ports . . . . .	145
10.9	Structure of the tool holder power supply for the supply of the tool and its control devices . . . . .	146

10.10	Structure of the SPATZ30 screwer with tool adapter for use on the KR90 . . . . .	147
10.11	Structure of the screwer with the integration into the RealCaPP architecture . . . . .	148
10.12	Composite structure diagram of the force-torque sensor of KR90. The RealCaPP capable OPC UA over TSN interfaces are marked orange. . . . .	149
10.13	Composite structure diagram of the grippers for the KR10 and KR6. Structures located on the tool side are grayed out. The RealCaPP capable OPC UA over TSN interfaces are marked orange. . . . .	150
10.14	Grippers of the KR6 and KR10 in the Robarista cell . . . . .	151
10.15	Force-controlled hand guiding of the KR90 in the WiR innovation lab . . . . .	152
10.16	RTS implementation of the force-torque controlled hand guiding . . . . .	153
10.17	Circuit board component . . . . .	154
10.18	Process steps for the assembly of the circuit board component . . . . .	155
10.19	Abstract encapsulated RTS of the Composed Skill pick and place . . . . .	156
10.20	Top view CAD drawing of the housing part with the annotated gripping options . . . . .	157
10.21	Possible hardware configurations for handling the housing part . . . . .	157
10.22	Construction plan of the aluminum structure consisting of two aluminum profiles that are screwed together via a bracket . . . . .	158
10.23	Abstract encapsulated RTS of the Composed Skill force controlled screwing . . . . .	159
10.24	Activity diagram for the processing steps for the assembly of the aluminum L-structure . . . . .	160
10.25	Activity diagram of the assembly process of the aluminum structure distributed between the two robot resources . . . . .	162
10.26	Overhanding of profile 1 with the bracket from the KR90 to the KR10 . . . . .	163
10.27	Tightening of the profile 1 with bracket onto profile 2 by the KR90, while the KR10 is holding the profile 1 with the bracket . . . . .	164
10.28	KR10 placing the fully assembled aluminum structure on the table . . . . .	164
10.29	Snapshot of the ontology showing the system state in which all the preconditions for clamping the portafilter in the coffee machine are met . . . . .	166
10.30	Snapshot of the ontology showing the system state after the portafilter has been successfully clamped into the coffee machine . . . . .	167
10.31	Schematic representation of the lever trajectory of the robot on the coffee machine . . . . .	168
10.32	Pictures of the execution of the Robarista . . . . .	169
11.1	Evaluation scenarios for the measurement of runtimes using the example of force-controlled robot motion . . . . .	174
11.2	Histogram of the Force-Torque Sensor RTS execution time . . . . .	176
11.3	Histogram of the Force-Torque Position Controller RTS execution time . . . . .	176

11.4	Histogram of the Industrial Robot RTS execution time . . . . .	176
11.5	Histogram of the round trip time of the OPC UA over TSN network transmission . . . . .	177
11.6	Histogram of the execution times of the force-controlled robot movement local RealCaPP implementation . . . . .	179
11.7	Histogram of the execution times of the force-controlled robot movement local ROS2 implementations . . . . .	179
11.8	Histogram of the execution times of the force-controlled robot movement distributed RealCaPP implementation (1 hop) . . . . .	180
11.9	Histogram of the execution times of the force-controlled robot movement distributed ROS2 implementation (1 hop) . . . . .	180
11.10	Histogram of the execution times of the force-controlled robot movement distributed RealCaPP implementation (2 hops) . . . . .	181
11.11	Histogram of the execution times of the force-controlled robot movement distributed ROS2 implementation (2 hops) . . . . .	181

## List of Listings

6.1	The anatomy of a SPARQL query. (Adapted from [44, pp. 308]) . . .	61
6.2	Simplified SPARQL query to create a subgraph with resources that are connectable and have specified skills. . . . .	67
6.3	SPARQL query for searching suitable configurations with the path length of the configuration . . . . .	68
8.1	ASK SPARQL query for the second precondition of the cube example	104
8.2	SPARQL query for determining the execution order. The RTSs are output in an ordered sequence. . . . .	109
10.1	SPARQL ASK query for checking whether all preconditions for the activity 2 (Pick Bracket and Place on Profile 1) are fulfilled . . . . .	161
10.2	SPARQL ASK query for checking whether all preconditions for clamping the portafilter in the coffee machine have been met . . . . .	166

## List of Equations

6.1	Example property chain for the relation <code>hasGrandparent</code> . . . . .	59
6.2	Example SWRL rule for classifying heavy-weight robots. . . . .	59
6.3	Example SWRL rule for creating new instances . . . . .	60
6.4	Property chain for the relation, that two resources are geometrically connectable. This rule also exists for electrical and data connections. . . . .	64
6.5	SWRL rule for the <code>isConnectableToResource</code> relation, that two resources are generally connectable to each other. . . . .	64
6.6	SWRL rule for the <code>isConnectableToResource</code> relation, that two resources are generally connectable to each other. . . . .	64
6.7	Exemplary SWRL rule for the derivation of the skill <code>pick and place</code> . . . . .	65
8.1	<i>rts</i> function of an RTS . . . . .	97
8.2	WCET of an RTS Network . . . . .	99
8.3	Property chain for the inference of the relation <code>connectedToRTS</code> . . . . .	109

## List of Tables

6.1	Result table of the SPARQL query for suitable configurations on the example subgraph. Empty columns were omitted. . . . .	69
11.1	Times for the self-introduction of resources in milliseconds. Comparison between the sequential and parallel implementation. . . . .	172
11.2	Times for initializing and starting real-time communication in seconds. Comparison of times with clocks synchronization and already existing clock synchronization. . . . .	173

# Supervised Theses

Björn DÖSCHEL. “Integration of Jerk-Limited Trajectories for Multi-Link Robots into the Robot Operating System”. Bachelor Thesis. University of Augsburg, 2020

Salome LANGEHEINECKE. “Generische Greifstrategien für Pick-and-Place mittels semantischer Bauteilbeschreibungen”. Bachelor Thesis. University of Augsburg, 2020

Markus KUGELMANN. “Entwicklung einer Echtzeit-Kommunikationsplattform für die verteilte Robotersteuerung”. Master Thesis. University of Augsburg and KUKA Deutschland GmbH, 2021

Tobias HOFSTETTEN. “Planung von Hardwarekonfigurationen für Roboteranwendungen mithilfe von semantischen Netzen”. Master Thesis. University of Augsburg, 2022

Johannes TINTENHERR. “Entwicklung einer Echtzeit-Kommunikationsschnittstelle für die verteilte Multi-Roboter-Kooperation”. Master Thesis. University of Augsburg, 2022

Maximilian Enrico MÜLLER. “Robarista: Orchestrierung von Anlagen- und Roboterfähigkeiten für eine Barista Anwendung”. Bachelor Thesis. University of Augsburg, 2023

Nicolai SANDMANN. “Anwendung Semantischer Annotationen für die Anwendungsspezifische Roboter-Endeffektor-Auswahl”. Bachelor Thesis. University of Augsburg, 2023





## Own Publications

1. Christian EYMÜLLER, Julian HANKE, Alwin HOFFMANN, Markus KUGELMANN, and Wolfgang REIF. “Real-time capable OPC-UA Programs over TSN for distributed industrial control”. In 2020 25th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA), volume 1, pages 278–285, 2020.
2. Christian EYMÜLLER, Julian HANKE, Alwin HOFFMANN, Alexander POEPEL, Constantin WANNINGER, and Wolfgang REIF. “Towards a Real-Time Capable Plug & Produce Environment for Adaptable Factories”. In 2021 26th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA), pages 1–4, 2021.
3. Christian EYMÜLLER, Julian HANKE, Alwin HOFFMANN, Wolfgang REIF, Markus KUGELMANN, and Florian GRÄTZ. “RealCaPP: Real-time capable Plug & Produce communication platform with OPC UA over TSN for distributed industrial robot control”. In 2021 IEEE 17th International Conference on Automation Science and Engineering (CASE), pages 585–590, 2021.
4. Christian EYMÜLLER, Julian HANKE, Alexander POEPEL, and Wolfgang REIF. “Towards Self-Configuring Plug & Produce Robot Systems Based on Ontologies”. In 2023 9th International Conference on Automation, Robotics and Applications (ICARA), pages 23–27, 2023.
5. Christian EYMÜLLER, Constantin WANNINGER, Alwin HOFFMANN, and Wolfgang REIF. “Semantic Plug and Play – Self-Descriptive Modular Hardware for Robotic Applications”. *International Journal of Semantic Computing*, 12(04):559– 577, 2018.
6. Julian HANKE, Christian EYMÜLLER, Alexander POEPEL, Julia REICHMANN, Anna TRAUTH, Markus SAUSE, and Wolfgang REIF. “Sensor-guided motions for robot-based component testing”. In 2022 Sixth IEEE International Conference on Robotic Computing (IRC), pages 81–84, 2022.
7. Julian HANKE, Christian EYMÜLLER, Julia REICHMANN, Anna TRAUTH, Markus SAUSE, and Wolfgang REIF. “Software-defined testing facility for component testing with industrial robots”. In 2022 IEEE 27th International Conference on Emerging Technologies and Factory Automation (ETFA), pages 1–8, 2022.
8. Alexander POEPEL, Christian EYMÜLLER, and Wolfgang REIF. “SensorClouds: A Framework for Real-Time Processing of Multi-modal Sensor Data for Human-Robot-Collaboration”. In 2023 9th International Conference on Automation, Robotics and Applications (ICARA), pages 294–298, 2023.
9. Martin SCHÖRNER, Constantin WANNINGER, Raphael KATSCHINSKY, Simon HORNING, Christian EYMÜLLER, Alexander POEPEL, and Wolfgang REIF. “UAV Inspection of Large Components: Determination of Alternative Inspection Points and Online Route Optimization”. In 2023 IEEE/ACM 5th International Workshop on Robotics Software Engineering (RoSE), pages 45–52, 2023.

10. Anthony STEIN, Christian EYMÜLLER, Dominik RAUH, Sven TOMFORDE, and Jörg HÄHNER. Interpolation-based classifier generation in XCSF. In 2016 IEEE Congress on Evolutionary Computation (CEC), pages 3990–3998, 2016.
11. Constantin WANNINGER, Sebastian ROSSI, Martin SCHÖRNER, Alwin HOFFMANN, Alexander POEPEL, Christian EYMÜLLER, and Wolfgang REIF. “ROSSi a graphical programming interface for ROS 2”. In 2021 21st International Conference on Control, Automation and Systems (ICCAS), pages 255–262. IEEE, 2021.
12. Constantin WANNINGER, Christian EYMÜLLER, Alwin HOFFMANN, Oliver KOSAK, and Wolfgang REIF. “Synthesizing capabilities for collective adaptive systems from self-descriptive hardware devices bridging the reality gap”. In Leveraging Applications of Formal Methods, Verification and Validation. Distributed Systems: 8th International Symposium, ISoLA 2018, Limassol, Cyprus, November 5-9, 2018, Proceedings, Part III 8, pages 94– 108. Springer, 2018.