Chair of Robotics, Artificial Intelligence and Real-time Systems
TUM Department of Informatics
Technical University of Munich

From Unpacking to Plug & Produce – Flexible Components Integration for Robots in Industry 4.0

Stefan Profanter

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

Doktor der Naturwissenschaften (Dr. rer. nat.)

genehmigten Dissertation.

Vorsitzender:

Prof. Dr. Daniel Cremers

Prüfende der Dissertation:

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

Die Dissertation wurde am 19.08.2020 bei der Technischen Universität München eingereicht und durch die Fakultät für Informatik am 31.03.2021 angenommen.

Abstract

Flexible components integration is one of the major challenges in the current fourth industrial revolution aiming at the computerization of manufacturing. Its focus is interoperability, virtualization, decentralization, real-time capability, service orientation, and modularity. Especially in the domain of industrial robots exchanging robots and their tools in production lines is connected with a high adaption effort. A typical robot work cell is composed of devices from various manufacturers, which rely on their specific control interfaces. To reduce the setup and reconfiguration time, a hardware-agnostic Plug & Produce system is required.

In this doctoral thesis, I am presenting a solution for the following research question: How can a Plug & Produce industrial robot cell be built up in a way that single system components and robot tools can be exchanged without the need of reprogramming or manually adapting control applications. To answer this question, I am splitting the main question into multiple smaller challenges.

Information exchange between system components is an essential part of adaptable systems. Therefore, I first evaluate the performance of different communication protocols which are typically used in the industrial automation domain. This evaluation includes OPC UA, ROS, DDS, and MQTT and evaluates the round-trip time, throughput, and other metrics such as CPU load in different situations.

Based on the results of this evaluation, I use OPC UA for the detailed implementation of my proposed Plug & Produce system architecture. To achieve a Plug & Produce system without the necessity of pre-configuration, automatic component detection is necessary. Based on the definition of the OPC UA Discovery Services, I implement local discovery mechanisms in the used open-source OPC UA stacks and evaluate their applicability.

After a component is discovered in the network, it needs to semantically describe its low-level functionalities, also called skills, so that other components in the system can re-use these skills. The main contribution of this thesis is a strongly typed generic skill model that can be used as a control interface for any Industry 4.0 component, be it either hardware or software, with a focus on reusability of skills across different platforms and domains. This skill model is based on the semantic annotation of OPC UA, but can also be adapted to other middlewares as long as they provide comparable features. On hard-

ware components, the presented skill model is implemented by device adapters which provide proprietary hardware-dependent functionalities through the generic OPC UA skill interface. The hierarchical composition of such skills allows for additional abstraction and grouping of functionalities. Based on this skill model, I provide various C++ classes as part of my proposed system architecture, used to detect available skills in the network (Skill Detector), or control such skills (Generic Skill Client).

The proposed skill model and its concepts are evaluated on three different robot cells consisting of robots and tools from different manufacturers. The control of the components is achieved via my generic skill interface described in this thesis. The overall Plug & Produce architecture is evaluated in a robot work cell setup with a robot, tool changer, and two grippers (parallel and vacuum) — all controlled through my OPC UA skill interface. Different components and their skills are automatically detected and parametrized based on the higher-level task.

The evaluation shows the significant benefit of the proposed system and its applicability in a Plug & Produce environment in the field of robot-supported industrial production. Although it is necessary to adapt existing hardware to support the proposed semantic skill concept, the initial one-time effort yields reoccurring efficiency gains in system reconfiguration. In particular, small lot production benefits from reduced changeover times.

Zusammenfassung

Die Integration flexibler Komponenten ist eine der größten Herausforderungen in der gegenwärtigen vierten industriellen Revolution, die auf die Digitalisierung und Modernisierung der Fertigungsindustrie abzielt. Ihr Schwerpunkt liegt auf Interoperabilität, Virtualisierung, Dezentralisierung, Echtzeitfähigkeit, Serviceorientierung und Modularität. Insbesondere im Bereich der Industrieroboter ist der Austausch von Robotern und ihren Werkzeugen in Produktionslinien mit einem hohen Anpassungsaufwand verbunden. Eine typische Roboterarbeitszelle setzt sich aus Geräten verschiedener Hersteller zusammen, die ihre eigenen spezifischen Steuerungsschnittstellen anbieten. Um die Einrichtungs- und Rekonfigurationszeit zu reduzieren, ist ein hardware-agnostisches Plug & Produce-System erforderlich.

In dieser Doktorarbeit präsentiere ich eine Lösung für die folgende zentrale Forschungsfrage: Wie kann eine Plug & Produce Industrieroboterzelle aufgebaut werden, die einen einfachen Austausch einzelner Systemkomponenten und Roboterwerkzeuge ermöglicht, ohne dass eine Neuprogrammierung oder Anpassung von Steuerungsapplikationen erforderlich ist. Um diese Frage zu beantworten, teile ich die Hauptfrage in mehrere kleineren Problemstellungen auf.

Bei anpassungsfähigen Systemen ist der Informationsaustausch zwischen Systemkomponenten ein wesentlicher Bestandteil. Daher untersuche ich zunächst die Leistungsfähigkeit verschiedener Kommunikationsprotokolle, die typischerweise im Bereich der industriellen Automatisierung verwendet werden. Diese Evaluierung umfasst OPC UA, ROS, DDS und MQTT und bewertet die Round-Trip-Zeit, den Durchsatz und andere Metriken wie die CPU-Last in verschiedenen Situationen.

Auf der Grundlage der Ergebnisse dieser Evaluierung verwende ich für die detaillierte Implementierung meiner leistungsfähigen Plug & Produce-Systemarchitektur OPC UA als Kommunikationsprotokoll. Um ein Plug & Produce-System ohne der Notwendigkeit einer Vorkonfiguration zu erreichen, ist eine automatische Komponentenerkennung erforderlich. Basierend auf der Definition der OPC UA Discovery Services implementiere ich lokale Erkennungsmechanismen in den verwendeten Open Source OPC UA Stacks und evaluiere deren Anwendbarkeit.

Nachdem eine Komponente im Netzwerk erkannt wird, muss sie ihre Low-Level-Funktionalitäten, auch Skills genannt, semantisch beschreiben, sodass andere Komponenten im System diese Skills wiederverwenden können. Als Hauptbeitrag dieser Arbeit entwickle ich ein stark typisiertes generisches Skill-Modell, das als Steuerungsschnittstelle für jede beliebige Industrie 4.0-Komponente, sei es Hardware oder Software, verwendet werden kann, wobei der Schwerpunkt auf der Wiederverwendbarkeit von Skills über verschiedene Plattformen und Domänen hinweg liegt. Dieses Skill-Modell basiert auf der semantischen Modellierung von OPC UA, es kann aber von jedem Kommunikationsprotokoll umgesetzt werden, welches ähnliche Funktionalitäten bietet. Das hier vorgestellte Skill-Modell wird auf Hardware-Komponenten durch Geräteadapter implementiert, die proprietäre hardwareabhängige Funktionalitäten über meine generische OPC UA Skill-Schnittstelle bereitstellen. Die hierarchische Zusammensetzung solcher Skills ermöglicht eine zusätzliche Abstraktion und Gruppierung von Funktionalitäten. Auf der Grundlage dieses Skill-Modells stelle ich unter anderem verschiedene C++ Klassen als Teil der von mir vorgeschlagenen Systemarchitektur zur Verfügung, die zur Erkennung verfügbarer Skills im Netzwerk (Skill Detector) oder zur Steuerung solcher Skills (Generic Skill Client) verwendet werden.

Das vorgeschlagene Skill-Modell und dessen Konzepte werden auf drei verschiedenen Roboterzellen evaluiert, die aus Robotern und Werkzeugen von verschiedenen Herstellern bestehen. Die Steuerung dieser Komponenten wird über die von mir vorgeschlagene Skill-Schnittstelle erreicht. Die komplette Plug & Produce-Architektur wird in einer Roboterarbeitszelle mit einem Roboter, einem Werkzeugwechsler und zwei Greifern (parallel und vakuum) evaluiert, wobei alle Komponenten über meine generische OPC UA Skill-Schnittstelle gesteuert werden. Verschiedene Komponenten und ihre Skills werden automatisch erkannt und auf der Grundlage der übergeordneten Aufgabe parametrisiert.

Die Auswertung zeigt den erheblichen Nutzen des von mir vorgeschlagene Systems und deren Anwendbarkeit in einem Plug & Produce-System im Bereich der robotergestützten industriellen Fertigung. Obwohl es notwendig ist, die vorhandene Hardware über spezielle Geräte-Adapter zu integrieren, um das vorgeschlagene Konzept der semantischen Skills zu unterstützen, führt der anfängliche einmalige Aufwand zu wiederkehrenden Effizienzgewinnen bei der Rekonfiguration des Systems. Insbesondere die Kleinserienfertigung profitiert von reduzierten Umrüstzeiten.

Contents

List of Figures

Acronyms

AAS Asset Administration Shell. 17, 83
AI Artificial Intelligence. 14
API Application Programming Interface. 84, 86, 104, 105

CAD Computer-Aided Design. 17
CoAP Constrained Application Protocol. 21

DDL Device Description Language. 24
DDS Data Distribution System. 4, 21, 22, 32, 35–44, 46, 48–50, 83, 126
DHCP Dynamic Host Configuration Protocol. 57, 67

GDS Global Discovery Server. 60, 61

I4.0 Industry 4.0. 13, 14, 16, 17, 33, 55, 56, 58–60, 65, 66, 92, 95
IoT Internet of Things. 22, 32, 33, 37
IP Internet Protocol. 38, 39, 56, 57, 61, 66, 67

KB Knowledge Base. 98, 99, 102

LDS Local Discovery Server. 58, 60–62, 68, 70
LDS-ME Local Discovery Server with Multicast Extension. 61, 62, 64–72, 97, 101, 118

mDNS Multicast Domain Name System. 69–71, 92
MES Manufacturing Execution System. 98
MQTT Message Queue Telemetry Transport. 4, 21, 22, 32, 37–44, 46, 48–50, 83, 126
MQTT-SN MQTT For Sensor Networks. 21, 39
MSB Manufacturing Service Bus. 65–67, 71

OPC UA Open Platform Communications Unified Architecture. 4, 16, 21–29, 32–36, 38–42, 44, 46–50, 55, 57, 60–72, 76–78, 81–83, 85–93, 97, 98, 100, 101, 104–106, 108–110, 113–116, 118–121, 123, 124, 126–129, 131–135, 137

PLC Programmable Logic Controllers. 13, 26

RAMI 4.0 Reference Architecture Model Industrie 4.0. 15, 17, 66, 128
ROS Robot Operating System. 4, 21, 22, 25, 28, 32, 36–44, 46–50, 83, 105, 126

SDN Software-Defined Networking. 71, 123

SME small- and medium-sized enterprises. 14, 16, 25
sMES semantic Manufacturing Execution System. 98, 100, 102
SOAP Simple Object Access Protocol. 21

TCP Transmission Control Protocol. 21, 22, 35, 37–39, 41–44, 46, 48, 50, 56, 57, 91, 105
TSN Time-Sensitive Networking. 23, 35, 39, 40, 81

UDP User Datagram Protocol. 21, 22, 35, 37–39, 41, 43, 46, 48, 50, 56, 57
USB Universal Serial Bus. 2, 3, 89, 100

VDMA Verband Deutscher Maschinen- und Anlagenbau e. V.. 15–17, 81, 88, 124, 128

1 Introduction

In the last years and up to the current year 2020, there is a strong trend to lot-size one production, which means that every produced product is unique in its production process. This is a strong contrast to typical production lines where one specific product is produced many times the same way, such as a car body. To adapt to this new trend, flexible production cells are required which are easy to reconfigure and to adapt.

1.1 The Big Picture

The transition from rigid production lines to automated flexible production cells is currently connected with a lot of effort and existing problems:

- Suitable devices need to be purchased, which are compatible with existing devices in the setup. Device manufacturers did not yet agree on an **established communication and control standard** for industrial automation devices.
- After unpacking the device, a **suitable adapting software or hardware component** needs to be developed to adapt the proprietary interfaces, to be able to integrate them into the existing environment.
- This adaption has the effect that the device can communicate with other devices in the same system, but other devices are not yet able to **automatically detect** this new device and adapt their process to it.
- A more **generic functionality description** is required which allows hardware-agnostic composition of functionality to achieve a real Plug & Produce system.
- Exchanging a device by other devices with similar functionality but from different manufacturers typically requires **reprogramming** of the control application to adapt to the new device properties.

The described problems in flexible component integration are major challenges in Plug & Produce production environments. The term Plug & Produce was formed around the year 2000 by [Arai et al., 2000]. The authors define the term as a “methodology to introduce a new manufacturing device into a manufacturing system easily and quickly”. The main idea behind the Plug & Produce concept is derived from the well-known Plug &

Play concept in the domain of computer systems: A Universal Serial Bus (USB) device can be plugged into a computer and is immediately available to be used without the need to manually program a driver for it. A USB mouse manufacturer can re-use the existing implementation of such USB drivers to support its basic functionality. If a mouse supports more advanced features, the manufacturer can develop a specific driver, which still implements the basic USB mouse protocol, extended with the special functionality.

Achieving the same level of automated integration of components in manufacturing shop floors is still a major challenge. The Multi-Annual-Roadmap (MAR) of the EU SPARC programme [SPARC, 2020] especially identifies configurability as one of the key system abilities of Plug & Produce systems. Standardized interfaces, and the identification of system functionalities, are basic requirements to achieve automatic configuration of devices.

The German Industry 4.0 Index 2019 [Staufen AG, 2019] shows that in the year 2019 companies are struggling with transforming their automation efforts from small scale individual projects to implementing the concept of smart factories on operational basis. This study also shows the main motivation behind adopting Industry 4.0: improvement of internal efficiency, increase in transparency of processes, and reduction of costs.

This thesis is addressing the main motivation behind Industry 4.0 by presenting novel concepts for smart factories, with a focus on the area of industrial robots and their components.

In the field of industrial robots there are multiple challenges which must be addressed to implement the concept of Plug & Produce. Different types of robots (articulated, scara, cartesian, spherical, parallel, cylindrical, and others) and their functionality must be described in a generic standardized way. To handle Plug & Produce of robot tools (e.g., parallel gripper, vacuum gripper, drilling tool, screwdriver), they also need a generic description of their functionality and interfaces, which allow easy integration without reprogramming. Software algorithms and other software components automatically need to adapt to new system configurations. At the same time, the interface to the composed system should not change to ensure that higher-level components are still able to control the system.

1.2 Benefits of Flexible and Standardized Integration

Flexible component integration will allow easy set-up of flexible manufacturing systems, especially industrial robot cells, and thereby reducing integration time by using well-defined interfaces and device descriptions. Algorithms, which are based on these

interfaces, can be reused for the newly setup system, even if it is from a different manufacturer or if it combines different components.

The approach proposed in this thesis also leads to a lower change-over effort, if the manufacturing cycle is modified, or the shop floor setup changes. Additionally, the engineering complexity of the whole system is reduced by decoupling and re-using standardized interfaces and therefore the first-time-right quality increases.

By using current standards and collaborating with standardization groups for future standardization activities, long term sustainability of this thesis results is significantly increased, and interoperability of different manufacturers is improved. Due to standardized interfaces, information can be easily accessed and extracted from the system and used for other use-cases, e.g., predictive maintenance, anomaly detection, or process optimization. The current trend of predictive maintenance and big data analysis requires a lot of data which, in the best case, is gathered through identical non-changing interfaces [Yan et al., 2017].

1.3 Research Question

The main research question which is addressed in this thesis is shown in Figure 1.1 and stated as the following:

> How can a Plug & Produce industrial robot cell be built up in a way that single system components and robot tools can be exchanged without the need of reprogramming or manually adapting control applications?

This is an important feature for the current Industry 4.0 movement, where new machines need to be integrated into shop floors on demand with as little re-configuration and re-programming as possible.

To answer this question, additional subsequent questions need to be answered: Which communication protocol and middleware is the best suitable one for this use-case? How can an automatic device detection and initialization be achieved, similar to USB devices? How are functionalities of a device best described to be reusable and controllable generically? Is such a system able to compete performance-wise with traditional systems?

The next section gives some proposed solutions to these questions. The proposed solutions are elaborated further in the progress of this thesis.

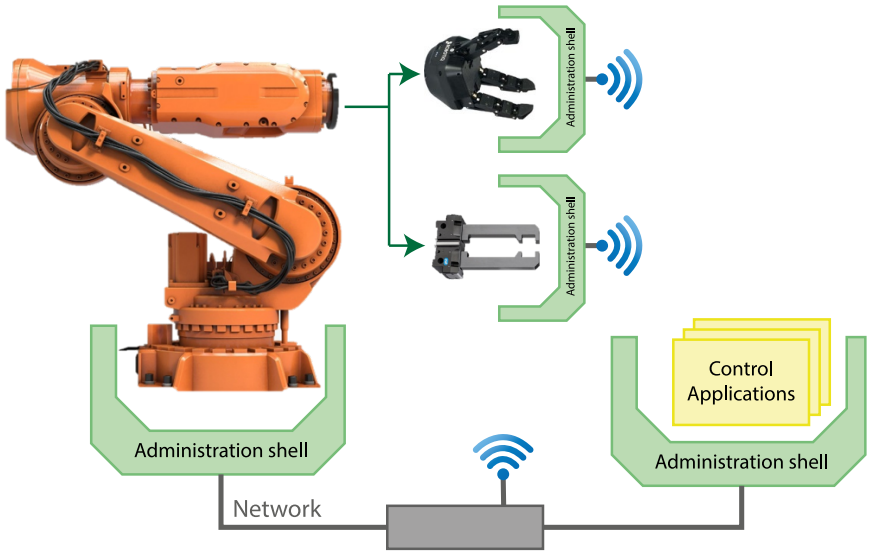


Figure 1.1: The main research question of this thesis shown on the example of an industrial robot, multiple tools, and control applications. Exchange of a component should not require any reprogramming of the remaining components.

1.4 Proposed Solutions

My main research question is split up into multiple subsequent questions. Proposed solutions and approaches how to answer these subsequent questions are discussed in this section.

The first step to achieve a Plug & Play like architecture is to find a suitable communication channel and protocol between the components, i.e., an **evaluation of middlewares** must be conducted. A middleware is a combination of software tools and standardized protocols supporting the information exchange between different components of a system. Different middlewares are currently used in the industrial domain, e.g., Robot Operating System (ROS), Data Distribution System (DDS), Message Queue Telemetry Transport (MQTT), or Open Platform Communications Unified Architecture (OPC UA). All these middlewares provide a varying feature set for different use-cases. Thus, designing a Plug & Produce system requires the evaluation of middlewares and comparing their features for suitability especially for Industry 4.0 applications. The main contribution in this part is to create a comprehensive comparison and evaluation of current middlewares and communication platforms for the application of Plug & Produce, especially in the field of industrial automation and robotics.

Since a suitable communication platform only provides a basic way of communication and information exchange without any system-specifics, the next step will be to define a concept for system configuration based on the middleware. Newly plugged in compo-

nents should automatically announce their services to other components in the system. Independent of the component's type it must configure itself for the network architecture and then be able to discover the server where it should register itself. One of the major issues in this step is to find the correct server in any network setup since there may be many controlling servers on different hierarchical levels. Thus, **hierarchical discovery** must be implemented in a way that a device or component itself does need as less pre-configuration as possible but is still able to cover a wide area of applications. The expected contribution will be a transparent discovery system which allows easy implementation for system integrators and component manufacturers, and easy detection of new components in the system to use their provided services.

As soon as a newly plugged in component is detected by its controlling entity (server) there are two steps necessary: first the component needs to offer its functionality through basic methods (skills), and the server needs to know how to control the component by using additional semantic information associated with the component's description. Offering skills implies that the device must implement a **device adapter** which wraps the robot's (or more generally the device's) functionality to allow other components to trigger the skill through the middleware. A main contribution of this thesis is to find a generalized way of implementing the device adapters for different sensors and actors, especially industrial robots and its tools, and at the same time being able to also support more specialized devices or subtypes.

The device's **self-description** is used by the controlling entity or server to detect the device type and its functionality. Based on this self-description, the server must decide which operations the device can perform and for which tasks it is suitable. The major challenge for the self-description is to develop a general understanding which allows defining at least the device's basic functionality in common terms. Since there are different device types, it is particularly challenging to find a common ground for all these types and still provide enough flexibility to also support more constrained device types.

Using hierarchical structuring of components, where one super component can have multiple subcomponents, allows abstraction of low-level functionalities and at the same time composing an easy to use interface for the combined components. Since a super component consists of multiple sub-components, **reconfiguration of components** also needs to be handled. It should be possible to replace subcomponents with similar components or remove these, without the need of reprogramming the super component.

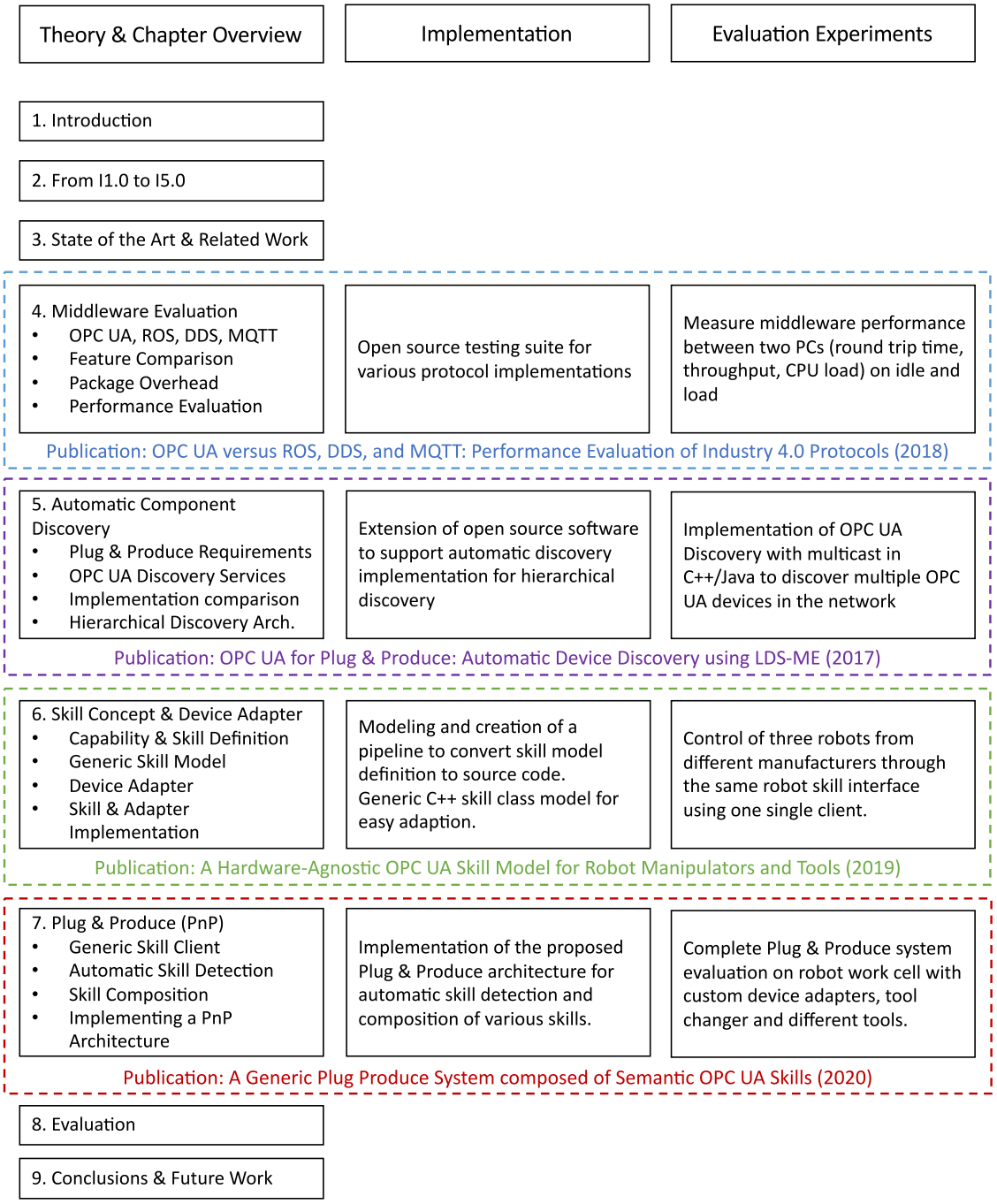


Figure 1.2: The structural layout of this thesis. The main contributions which correspond to the four main chapters are marked with dashed boxes, each of which was published in a peer-reviewed paper.

1.5 Structure of the Thesis

As shown in Figure 1.2 this thesis starts with an overview of current approaches in the industrial automation domain, followed by proposed solutions and related work. The proposed solutions from previous section are then separated in their own chapters within this thesis. These chapters describe the research results leading to a solution for each of the described problems. At the end, these results are evaluated in multiple real-world robotic system demonstrators showing the applicability of the proposed system architecture and standardized interfaces.

Throughout this thesis I will use the exemplified setup shown in Figure 1.3 to highlight the relevance of each main chapter as part of the whole system architecture.

The central communication platform is an Ethernet-based middleware. Hardware components (bottom) represent devices being part of the system setup, while software components (top) represent algorithms required for process execution. Since almost every manufacturer delivers devices with proprietary interfaces, they require additional device adapters to be able to communicate with other components. Components can use skill composition to create higher-level functionality through low-level skills.

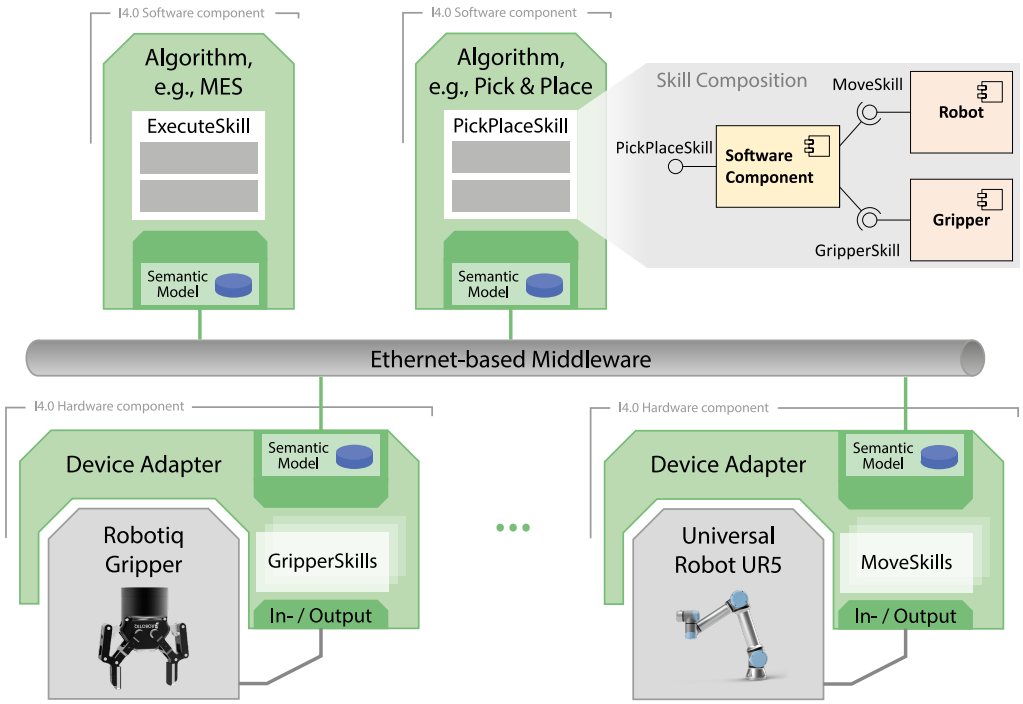


Figure 1.3: Exemplified system setup with hardware (bottom) and software (top) components connected through an Ethernet-based middleware.

1.6 List of Publications

The following list includes all publications where I contributed during the progress of this thesis. My contributions include, but are not limited to, the development of theoretical concepts, implementation of these concepts, and experimental evaluation. All the publications below are peer-reviewed.

The following list (ordered by publication date) presents my publications which are highly relevant to this Doctoral Thesis and where I am the main author. A more detailed description of the paper content can be found in the various chapters of this thesis.

- **Profanter, S., Perzylo, A., Somani, N., Rickert, M., & Knoll, A. (2015). Analysis and Semantic Modeling of Modality Preferences in Industrial Human-Robot Interaction, In *IEEE International Conference on Intelligent Robots and Systems*. https://doi.org/10.1109/IROS.2015.7353613**
- **Profanter, S., Dorofeev, K., Zoitl, A., & Knoll, A. (2018). OPC UA for Plug & Produce: Automatic Device Discovery using LDS-ME, *IEEE International Conference on Emerging Technologies and Factory Automation (ETFA)*. https://doi.org/10.1109/ETFA.2017.8247569**
- **Profanter, S., Tekat, A., Dorofeev, K., Rickert, M., & Knoll, A. (2019). OPC UA versus ROS, DDS, and MQTT: Performance Evaluation of Industry 4.0 Protocols, *Proceedings of the IEEE International Conference on Industrial Technology (ICIT)*. https://doi.org/10.1109/ICIT.2019.8755050**
- **Profanter, S., Breitkreuz, A., Rickert, M., & Knoll, A. (2019, September). A Hardware-Agnostic OPC UA Skill Model for Robot Manipulators and Tools, *Proceedings of the IEEE International Conference on Emerging Technologies and Factory Automation (ETFA)*, IEEE. https://doi.org/10.1109/ETFA.2019.8869205**
- **Profanter, S., Perzylo, A., Rickert, M., & Knoll, A. (2020). A Generic Plug & Produce System composed of Semantic OPC UA Skills. *Submitted for review in IEEE Open Journal of the Industrial Electronics Society***

The following list (ordered by publication date) presents publications where I contributed in collaboration with my colleagues.

- Perzylo, A., Somani, N., Profanter, S., Rickert, M., & Knoll, A. (2015b). Toward Efficient Robot Teach-In and Semantic Process Descriptions for Small Lot Sizes, *Robotics: Science and Systems (RSS), Workshop on Combining AI Reasoning and Cognitive Science with Robotics*
- Perzylo, A., Somani, N., Profanter, S., Rickert, M., & Knoll, A. (2015a). Multimodal Binding of Parameters for Task-based Robot Programming Based on Semantic Descriptions of Modalities and Parameter types, *CEUR Workshop Proceedings*

- Perzylo, A., Somani, N., Profanter, S., Gaschler, A., Cai, C., Griffiths, S., Rickert, M., Lafrenz, R., & Knoll, A. (2015). Ubiquitous Semantics: Representing and Exploiting Knowledge, Geometry, and Language for Cognitive Robot Systems. *IEEE/RAS International Conference on Humanoid Robots (HUMANOIDS), Workshop Towards Intelligent Social Robots - Current Advances in Cognitive Robotics*, (November)
- Haage, M., Profanter, S., Kessler, I., Perzylo, A., Somani, N., Sörnmo, O., Karlsson, M., Robertz, S., Nilsson, K., Resch, L., & Marti, M. (2016). On cognitive robot woodworking in SMErobotics, *International Symposium on Robotics (ISR)*
- Perzylo, A., Somani, N., Profanter, S., Kessler, I., Rickert, M., & Knoll, A. (2016). Intuitive Instruction of Industrial Robots: Semantic Process Descriptions for Small Lot Production, *Proceedings of the IEEE International Conference on Intelligent Robots and Systems (IROS)*. https://doi.org/10.1109/IROS.2016.7759358
- Dorofeev, K., Cheng, C.-H., Guedes, M., Ferreira, P., Profanter, S., Zoitl, A., Guedes, M., Profanter, S., & Zoitl, A. (2017, September). Device Adapter Concept towards Enabling Plug&Produce Production Environments, *IEEE International Conference on Emerging Technologies and Factory Automation (ETFA)*. https://doi.org/10.1109/ETFA.2017.8247570
- Perzylo, A., Rickert, M., Kahl, B., Somani, N., Lehmann, C., Kuss, A., Profanter, S., Beck, A. B., Haage, M., Hansen, M. R., Nibe, M. T., Roa, M. A., Sornmo, O., Robertz, S. G., Thomas, U., Veiga, G., Topp, E. A., Kessler, I., & Danzer, M. (2019). SMErobotics: Smart robots for Flexible Manufacturing. *IEEE Robotics and Automation Magazine*, *26*(1). https://doi.org/10.1109/MRA.2018.2879747
- Perzylo, A., Profanter, S., Rickert, M., & Knoll, A. (2019, September). OPC UA NodeSet Ontologies as a Pillar of Representing Semantic Digital Twins of Manufacturing Resources, *2019 24th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA)*, IEEE. https://doi.org/10.1109/ETFA.2019.8868954
- Madiwalar, B., Schneider, B., & Profanter, S. (2019). Plug and Produce for Industry 4.0 using Software-defined Networking and OPC UA, *Proceedings of the IEEE International Conference on Emerging Technologies And Factory Automation (ETFA)*
- Dorofeev, K., Profanter, S., Cabral, J., Ferreira, P., & Zoitl, A. (2019). Agile Operational Behavior for the Control-Level Devices in Plug&Produce Production Environments. *IEEE International Conference on Emerging Technologies and Factory Automation, ETFA*, (July). https://doi.org/10.1109/ETFA.2019.8869208

During my Doctoral Thesis, I also supervised various Bachelor's and Master's Theses which are listed here:

- Kraft, M. (2016). Bachelor Thesis: Design and Evaluation of an Intuitive Graphical User Interface for Industrial Robotic System Programming
- Tekat, A. (2018). Master Thesis: Implementation of a Test Suite to Compare the Performance of Different Industry 4.0 Communication Protocols

- Akin Başer, Ç. (2018). Master Thesis: Extension of an OPC UA Test Suite for Performance Evaluation of Embedded Devices using open62541
- Breitkreuz, A. (2019). Bachelor Thesis: Implementation of Modular Robot Force Skills in OPC-UA for Industrial Robots
- Madiwalar, B. (2019). Master Thesis: Plug & Produce for Industry 4.0 using Software-defined Networking and OPC UA
- Schmid, L. (2019). Bachelor Thesis: Modular Robot Tools With a generic Software Interface for Flexible Manufacturing using Robots
- Wielander, F. (2020). Master Thesis: Knowledge-based Skill Editor for the Hierarchical Composition of Device Capabilities
- Turiy, T. (2020). Bachelor Thesis: Control Constructs, Error Handling and Parameter Representation for a Knowledge-based Robot Skill Editor

2 From Industry 1.0 to Industry 5.0

The manufacturing industry is under constant change and new challenges come with new customer requirements. Over the last decades new inventions allowed new ways for producing goods, while at the same time the demands for more complex and customized products increased proportionally.

This chapter gives a short overview of the major historical changes in industrial automation, from the industrial revolution to the digital transformation. In addition, significant current technologies and concepts are described which are referenced in this thesis.

2.1 A retrospective on industrial automation

For centuries, human workforce was used to manufacture goods by hand. Within the last 250 years, these processes of manufacturing changed significantly due to different momentous inventions. The past industrial revolution can be separated into four main periods and extended by an outlook on a possible next significant change after today. Main characteristics of each period are given in Table 2.1 and described in the following sections.

Industrial Revolution	**1st**	**2nd**	**3rd**	**4th**	**5th**
Period	1780 - 1870	1870 - 1941	1941 - 2010	2010 - Today	Future
Driving Factor	Mechanization	Electrification	Automation	Digitalization	Personalization
Main Invention	Steam Engine	Assembly Lines	Electronics & Computer	Connected Devices	AI & Collaboration

Table 2.1: Main characteristics from the first Industrial Revolution up to the current 4th revolution, with a possible next industrial change in the future.

2.1.1 Industry 1.0 (1780 - 1870)

The first industrial revolution was introduced by using the power of water and steam to aid manufacturer workers and mainly influenced the textile industry. A fixed date cannot be determined, when the first industrial revolution started [Freeman and Soete, 1997]. It was around the year 1780 when in Great Britain the manufacturing of cloths changed from small decentralized cottages to more centralized manufacturing. This was possible with increasing use of mechanical machines: In 1784 Edmund Cartwright invented the power loom supported by the first steam engines, which increased the production of a single worker by more than 40 times [Ayres, 1989]. First versions of such steam engines were already developed by the beginning of the year 1700. Only when James Watt made some fundamental changes to the initial design in the years 1778 and following, more and more industries started to use the rotary power of the steam engines to drive their machinery. Steam-powered iron production, new inventions in chemical processing, and development of machining tools for milling, cylinder boring and screw cutting gave the that industrial change an even bigger momentum.

2.1.2 Industry 2.0 (1870 - 1941)

The steam engine was constantly improved, its efficiency increased, reaching higher rotation speeds while decreasing the energy consumption. New processes for ironing lead to the invention of steel, which provided higher quality and production speedup to support the new demands for such materials, especially for new railroad tracks and structural steel for buildings. With the first electrical generators around 1870, and the invention of the light bulb and the telephone, the way people worked and lived fundamentally changed [Constable and Somerville, 2004]. Before that time people mainly worked during daylight, when now they could work in multiple shifts even during the night with the support of artificial light. This change also introduced the transition to production lines where one worker performed one specific task, instead of building the whole product in parallel. Henry Ford perfected this shift to mass production by introducing the first assembly line for car manufacturing around the year 1915. With electricity came new developments in the ironing industry (Aluminum) and electrical motors empowered the development of new machines. The start of the second world war mainly marks the end of this second industrial revolution.

2.1.3 Industry 3.0 (1941 - 2010)

The invention of more complex electronic devices, such as the transistor and, later, integrated circuit chips led to the next significant change in industrial manufacturing. Therefore, this period is sometimes also referred as digital revolution. In the first stages, fully automated machines supplemented or even replaced human operators. Further advances in the information technology allowed even more automation in manufacturing, e.g., by the development of Programmable Logic Controllers (PLC) in the late 1960s.

Early computers, such as the Zuse Z3 (1941) or ENIAC (1945), were significantly improved, to reach the compactness of a home computer around 1970. Local connection of these devices had a major influence on the manufacturing shop floor. The most significant invention was the World Wide Web (WWW) by Tim Berners-Lee in 1989. In the following years many more intelligent devices were developed (e.g., digital cameras, smartphones, tablets), which sped up the transition to a more digital age.

Another significant invention was the industrial robot: In 1938 Griffith P. Taylor invented the Robot Gargantua, a crane-like device used to set wooden blocks in preprogrammed patterns. Unimation was the first company to produce a robot for industrial usage in the years following 1961, called Unimate [Nof, 1985]. KUKA was the first company to build a robot which had six electro mechanically driven axes in the year 1973, which they named Famulus [Singh and Sellappan, 2013]. Over the years the field of robotics made many advancements, one of which are humanoid robots imitating the human biology.

2.1.4 Industry 4.0 (2010 - Today)

The term Industry 4.0 (I4.0) is often used as a synonym for the fourth industrial revolution and originates from the German government, when it released its new high-tech strategy in the year 2011 with the goal of computerization of manufacturing.

The main trend of I4.0 includes the introduction of new technologies in manufacturing, i.e., interoperability, virtualization, decentralization, real-time capability, service orientation, and modularity [Directorate General for Internal Policies, 2016]. The goal is to highly interconnect machines, sensors, and other devices vertically and horizontally, while collecting data in the cloud for more intelligent, decentralized decision of product-based manufacturing.

Vertical integration stands for interconnecting all levels of an enterprise, from the production line up to the high-level business processes. Horizontal integration incorpo-

rates the interconnection of all machines, devices, and human workers on one level even across different locations [Schebek et al., 2017].

With these achievements in flexible manufacturing, small- and medium-sized enterprises (SME) should reach a higher level of competitiveness. Nowadays, SMEs need to cope with increasing customization demands from their customers [Koch et al., 2014; Perzylo, Rickert, et al., 2019]. In an adaptable manufacturing shop floor new intelligent machines and devices need to be integrated as fast and easy as possible, and the time to set up and configure should be short [Mehrabi et al., 2000].

This challenge of fast and easy integration of new devices and machines is still not solved and it is the focus of this thesis.

2.1.5 Industry 5.0 - What the future could be

This subsection is a visionary outlook to the future. It is based on current trending technologies which may play an important role in future manufacturing. All presented industrial revolutions had or have the impact, that people can do their jobs quicker and more efficiently. Therefore, it is straightforward to assume, that the fifth industrial revolution will deal with challenges and tasks currently performed by human workers. Looking at the solutions developed in Industry 4.0, human workers are still required for machine maintenance, shop floor planning, setup, and to assist machines while handling complex tasks. At the same time research currently focuses on automating almost everything while the human worker is sometimes forgotten [Kinzel, 2017]. The future needs to focus on reintegrating human hands and minds back into the industrial world. This can be achieved by strong collaboration of humans and machines and introducing more Artificial Intelligence (AI) into these systems. The human intelligence needs to work together with cognitive computing. Humans will add additional value to the tasks of machines to converge from mass production to mass customization and personalization.

2.2 Reference Models and Guidelines

One of the central aspects of I4.0 is the concept of components, which have a corresponding digital counterpart. This digital counterpart is often called digital twin as it represents a digital version of the corresponding component. To achieve a common understanding for symbols, alphabet, vocabulary, grammar, semantics, and culture, different guidelines and reference models have been developed. This section briefly describes

three well-known publications, while there are many more similar documents available by different organizations.

2.2.1 Reference Architecture Model Industrie 4.0 (RAMI 4.0)

One of the widely accepted models is the Reference Architecture Model Industrie 4.0 (RAMI 4.0) [ZVEI, 2015b] by a union of various organizations (ZVEI, VDI/VDE, DKE, Verband Deutscher Maschinen- und Anlagenbau e. V. (VDMA) and more). RAMI 4.0 collects the essential elements of Industry 4.0 in a three-dimensional layer structure as shown in Figure 2.1.

The RAMI 4.0 model can be used especially for the migration into the Industry 4.0 world as it is a tool to locate a new concept inside the model's coordinate system. Its goal is to give an orientation for standardization activities and show missing links between specific concepts to have a common understanding on those concepts.

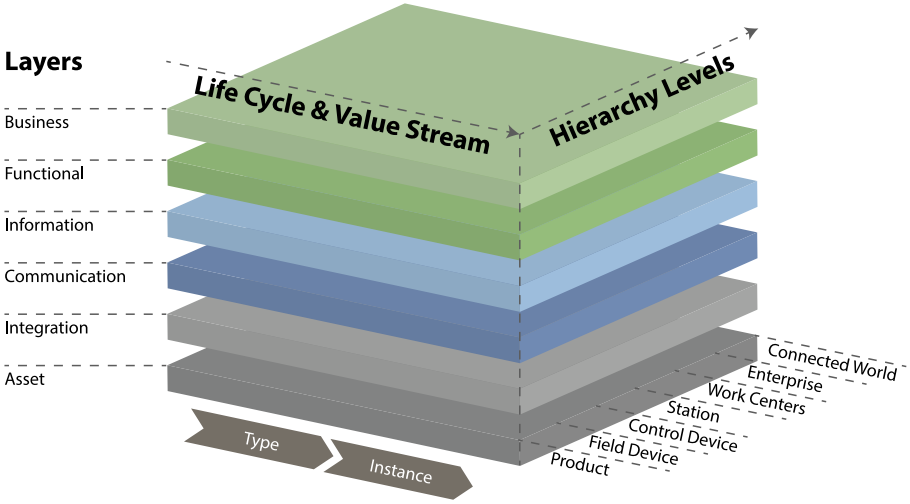


Figure 2.1: Three-dimensional map showing the most important aspects of Industry 4.0 according to RAMI [ZVEI, 2015b].

The *Hierarchy Levels* axis on the right side represents various functionalities inside a production facility according to the IEC 62264 standard, extended with Product and Connected World. It separates the axis into possible hierarchical levels where the technology can be assigned to, from the Product Level, through Field Level, Stations, whole Enterprises up to the Cloud.

The *Life Cycle & Value Stream* axis on the left represents the life cycle of a production line or product: an instance is created from a type. Type is subdivided into development

and maintenance usage, going from the construction of a product (drawings, construction plans) to its model maintenance, e.g., manuals or software updates. Instance is subdivided into production and maintenance usage for a specific product instance, i.e., the production of a product, its serial number, maintenance cycles or error messages.

The third axis *Layers* defines the digital location of a concept and hereby assigns a specific functionality to that component. The two top layers business and functionality cover organization and business processes and the functionality of a product. Information describes which data a product has to include, while communication describes the methodology to access this data. Integration provides the transition between the physical and digital world, while asset is the real physical product.

2.2.2 VDMA Guideline for the Introduction of OPC UA

A more technical related guideline for the adaption of Industry 4.0 concepts, especially OPC UA, was released by VDMA in 2017 [VDMA and Fraunhofer IOSB-INA, 2017].

The authors state the following:

> The self-information capability of Industry 4.0 communication reduces the configuration effort and facilitates user understanding. [...] The information model is the "operating manual", which describes the use of components, machines, and plants.

These statements are not only truly relevant in the context of this thesis, but also for the adaption of Industry 4.0 concepts in SMEs.

The authors suggest adapting OPC UA in existing production environments in three migration steps:

1. In the first step one should focus on implementing OPC UA interfaces for existing devices and provide data for condition monitoring.
2. The second step goes already in the direction of standardized models for specific domains (OPC UA Companion Specifications, see Section 4.2.1).
3. In the last step, a more advanced information model should be developed, which supports hardware-independent communication between different I4.0 components.

Many enterprises already focus their work onto creating OPC UA interfaces for condition monitoring in their shop floors. Only a few are already at the second step to integrate existing companion specifications. For the last step, hardware-independent

models are still missing and currently being developed. This is also where the results of this thesis are significantly contributing in collaboration with the VDMA, by providing a concept for a generic skill interface within the corresponding working group.

2.2.3 Industry 4.0 Component & Asset Administration Shell

Based on the RAMI 4.0, ZVEI released a reference model for an Industry 4.0 component [ZVEI, 2015a]. It is the first model based on RAMI 4.0 and describes basic properties of cyber-physical systems, especially hardware and software components. An important property of such a component is that it collects all relevant data throughout its life cycle electronically in a secure container. This secure container is defined as Asset Administration Shell (AAS). This container contains data (Computer-Aided Design (CAD), connection diagrams, manuals), provides certain functions (project planning, configuration), and offers services to access its data and functions. It is a digital twin of the physical product.

The encapsulation of components using the administration shell provides a standardized interface for other I4.0 components, and to higher level control systems. Additionally, such I4.0 components can be logically nested to group multiple sub-components into a bigger component (see Figure 2.2).

This thesis focuses on the extension of the component's functionality and standardized description of its service interface to achieve easier integration of such components into a complex production setup.

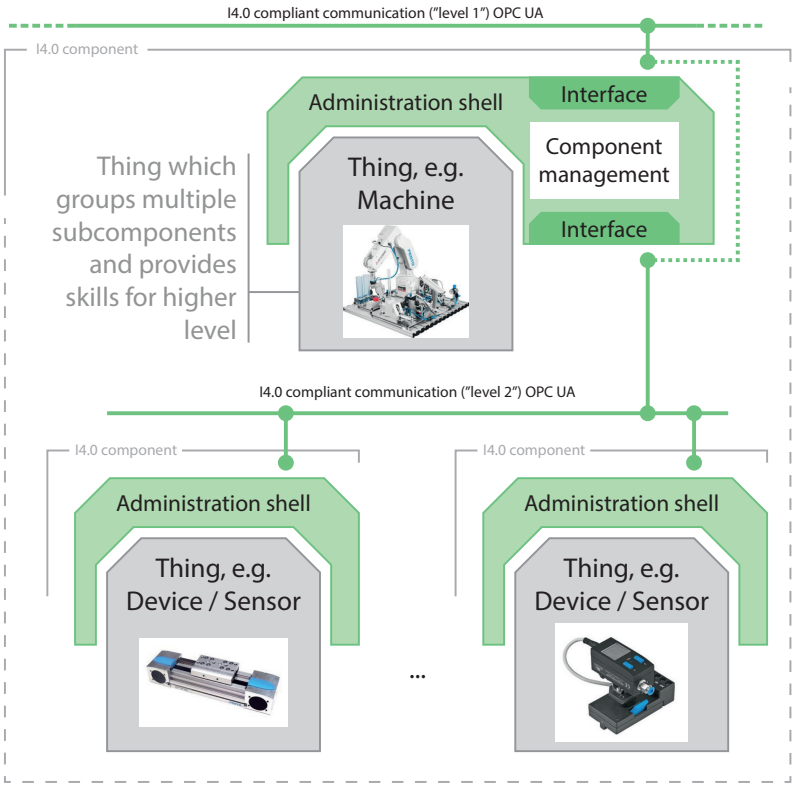


Figure 2.2: The administration shell embodies machines, devices or sensors and provides a standardized interface to other Industry 4.0 components. An administration shell may also act as a logically nested group of multiple sub-components [ZVEI, 2015b].

3 State of the Art & Related Work

Already 20 years ago, reconfigurable manufacturing was identified as one of the six key challenges for the year 2020 by the National Research Council [National Research Council, 1998]. In [Yusuf et al., 1999] the authors identified agile manufacturing as a key technology for an increasingly competitive market of fast changing customer requirements, while one of the major research issues is the definition of component interfaces using scientific knowledge [Mehrabi et al., 2000]. Especially cost consideration, increased customer choice, and easy integration of new devices is one of the most important drivers of agility and evolvable production systems. More than ten years later, the German government released their new high-tech strategy in 2010 (see Section 2.1.4) and defined flexible and agile manufacturing as one of the main paradigms of the 21st century, while forming the term Industry 4.0.

Between these first steps in the direction of a Plug & Produce system for industrial manufacturing and today (year 2020), a lot of research has led to current advances. This chapter gives an overview of the current state of the art and related work not only at the beginning of writing this thesis (year 2015), but also current research results separated into multiple sections, similar to the main chapters of this thesis. I also show the research gaps in the related work and explain how this thesis solves the research gaps.

3.1 Field Bus and Middleware Communication

A substantial part of automation systems is information, the processing of information, and the flow of information. The importance of this information is growing proportionally with the size of such systems. In a Plug & Produce system, exchange of information is one of the key components. Therefore, I take a deeper look into state of the art and related work in the area of field busses and middlewares to identify the applicability of current solutions for my proposed system.

A first attempt in structuring the information flow was made within the scope of computer-aided manufacturing (CAM [Chang and Wysk, 1997]). The resulting structure, with a strict subdivision of information processing into hierarchical levels, is nowa-

days known as the automation pyramid shown on the left-hand side of Figure 3.1. The automation pyramid itself is not standardized, but rather is a concept to structure information flow [Sauter et al., 2011].

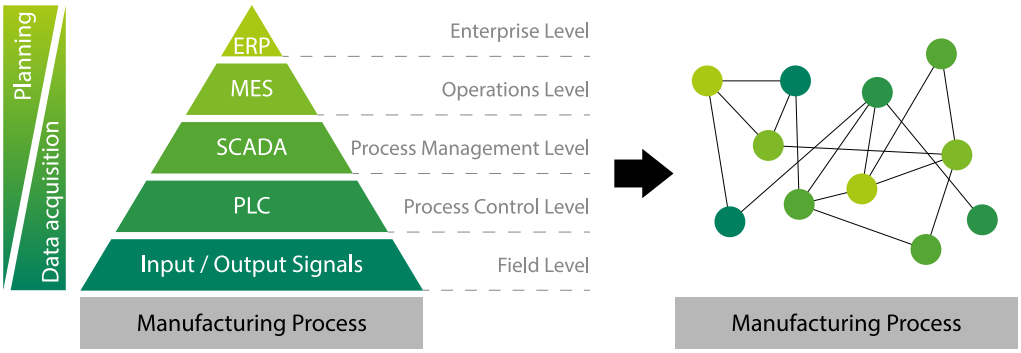


Figure 3.1: Automation pyramid and the transition to a fully connected non-layered Industry 4.0 architecture.

The automation pyramid is separated into multiple levels:

- The enterprise resource planning (ERP) is mainly a product-driven system that aims to integrate all business processes such as procurement, material management, logistics, sales, distribution, financial accounting, production and more.
- Manufacturing execution systems (MES) act as a bridge between the office and shop floor and are managing the information on the product to be produced, availability of production resources, and scheduling of the production itself.
- Supervisory control and data acquisition (SCADA) is responsible for the execution of a production recipe by controlling multiple lower-level devices and monitor the execution. A SCADA system typically controls multiple PLCs.
- Programmable logic controllers (PLC) are controlled by the SCADA system and are typically hard real-time systems to control and regulate directly connected machines and sensors.
- On the field level, there are final control elements such as sensors and actuator.

Every layer needs a way to exchange information with its neighboring layer. Typically, the higher layers (ERP, MES) use IP-based communication protocols, while lower levels (SCADA, PLC) use different types of proprietary field busses (e.g., controller area network (CAN), PROFIBUS, Modbus). With the success of Ethernet-based networks, significant effort was directed into getting this protocol down to the lowest field level. Due to the lack of real-time capabilities, it resulted in even more dedicated solutions, such as PROFINET or EtherCAT. Therefore, there are various protocols involved in the different layers. A more detailed look into these industrial communication protocols can be found in [Wollschlaeger et al., 2017].

The Time Sensitive Networking (TSN) working group within IEEE was formed in 2012, and has its goal to close this gap by defining a standard for real-time capable Ethernet. This technology will allow using standard Ethernet even on the field level and therefore replace currently used field busses [Decotignie, 2005]. Using real-time capable Ethernet-based middlewares the goal is to avoid a strictly layered architecture and transition to a fully interconnected system with one single middleware, as shown in Figure 3.1 on the right-hand side. Such interconnected systems allow communication between all components without protocol translation, and can be used together with a cloud infrastructure, where sensor data is directly pushed into cloud databases for predictive maintenance systems.

To ensure high performance on the information flow, choosing a performant and feature-rich middleware is crucial. Therefore, numerous performance tests have already been conducted in various domains. In the IoT domain the question of the communication protocol's performance evaluation is seen as a crucial one, because usual scenarios include resource-constrained devices communicating with each other over low bandwidth or unreliable wireless networks. In [Chen and Kunz, 2016] a comparison of bandwidth consumption and latency of most common IoT protocols, including MQTT, Constrained Application Protocol (CoAP), and DDS is given, where the authors conclude that DDS outperforms MQTT in poor network conditions while consuming higher bandwidth. Since CoAP is based on User Datagram Protocol (UDP), unpredictable packet loss may occur. In [Mun et al., 2016] the authors evaluate CoAP, MQTT, MQTT For Sensor Networks (MQTT-SN), Transmission Control Protocol (TCP), and WebSockets, and compare energy performance and CPU power consumption for each of the protocols. It can be seen from the results that protocols with low implementation overhead, like TCP and WebSockets, result in a lower energy consumption which correlates to the induced CPU load. A performance comparison of data usage and the time spent to send and receive messages for MQTT and OPC UA can be found in [Rocha et al., 2018]. The authors conclude that MQTT uses marginally less data to transmit the same payload, while OPC UA delivers lower performance in transmitting a message to multiple clients. Unfortunately, the authors do not state which implementations they use, therefore the results are not comparable to other similar research results.

Other papers present performance evaluations for a specific protocol: [Veichtlbauer et al., 2017] shows the good performance of an OPC UA server in a field device, measuring response times, memory, and CPU utilization of an OPC UA Server running on an Altera Cyclone I FPGA. [Cavalieri and Cutuli, 2010] evaluates the performance of different OPC UA features, such as security, binary transport, and Simple Object Access Protocol (SOAP) transport, while [Haskamp et al., 2017] gives an overview of the features of different OPC UA implementations, including open62541, an open-source C (C99) implementation of OPC UA used in my thesis. A more detailed comparison between ROS and ROS2, especially considering different DDS implementations such as Connext,

OpenSplice, and Fast RTPS is evaluated in [Maruyama et al., 2016]. They show that using DDS for ROS2 gives a significant performance improvement compared to ROS1.

A survey of supported communication paradigms between OPC UA and DDS is presented in [Pfrommer et al., 2016], with a focus on how both protocols can be run in hybrid deployments.

There are still major gaps in the presented middleware evaluations, such as the theoretical comparison of the protocol definitions, a side-by-side comparison of various features of used middlewares, and the performance especially in non-ideal situations, e.g., if there is a high network or high CPU load caused by other applications. Therefore, my evaluation in Chapter 4 is not only comparing the performance of multiple protocols (OPC UA, DDS, MQTT, ROS) in different situations (idle, high CPU load, high network load) using a single test suite, but is also investigating the theoretical performance comparison based on its binary package definition. The evaluation also focuses on typical industrial use cases, e.g., by evaluating the performance of multiple nodes on the same host, showing significant differences between the presented protocols.

3.2 Automatic Device Discovery

Automatic reconfiguration of system components in a Plug & Produce system depends on the ability to automatically discover newly plugged-in devices and components. This is not only important in the industrial domain, but also in various other domains such as home entertainment, Internet of Things (IoT), or networks in general.

In home and office networks, Universal Plug and Play (UPnP) allows detecting specific services on the network automatically, such as printers or file shares. The Devices Profile for Web Services (DPWS) is the successor of UPnP and was especially developed for embedded devices [OASIS, 2009]. The combination of DPWS and the Web Services-Discovery (WS-Discovery) allows automatic detection of DPWS-enabled devices within the network. A service on a DPWS-enabled device can be invoked by the clients using Web Services technology. The service interface is described using the Web Services Description Language (WSDL) which must be known to the client before calling a specific function. DPWS uses XML-based SOAP messages encapsulated in HTTP and transported via UDP or TCP. A possible solution for Ad-hoc field device integration using DPWS is shown in [Hodek and Schlick, 2012].

The comparison of DPWS with OPC UA shows that OPC UA is more suitable for limited hardware due to its flexibility in implementing small OPC UA applications with a specific purpose: It requires less memory by a factor of more than 90% [Dürkop et al., 2012],

even if those applications do not entirely fulfill the requirements of device-level Service-oriented Architecture (SOA) [Cândido et al., 2010]. In typical Service-oriented Architectures, DPWS is often used for service discovery and device integration [Eichhorn et al., 2010]. Compared to DPWS, OPC UA has better support for resource limited embedded devices using the Nano Embedded Device Server Profile with low requirements on the hardware performance (ARM9, 100Mhz, 64KB) [Imtiaz and Jasperneite, 2013]. Additionally, OPC UA has a well-defined meta model compared to the open approach of DPWS which offers greater extensibility, and more importantly, interoperability.

In [Dürkop et al., 2013] OPC UA is used for auto configuration of real-time ethernet systems. The authors' approach is to use a predefined OPC UA server where the device must register itself, without any automatic detection. Additionally, the focus of this paper is on specific real-time ethernet devices. OPC UA is currently in the process of integrating Time-Sensitive Networking (TSN), and thus configuration of real time ethernet in combination with automatic device discovery can be simplified to non-real-time ethernet [Nsaibi and Leurs, 2016].

Service discovery based on substring matching is used in the Service Location Protocol (SLP) defined as the RFC 2608 standard [Guttman, 1999], and is often used in LAN-enabled printers or to find network shares. It is mainly used in local networks, where each service is associated with a specific URL and a set of name/value pairs. SLP does not support more advanced semantic description of devices and their properties.

Automatic component discovery has to not only discover a device's presence, but it must also be able to determine the extended semantic functionality description, which is still a research gap, especially in the domain of industrial automation. The OPC UA Discovery Specification Part 12, released in July 2015 [OPC Foundation, 2015], is a fairly new addition to the standard and is adding device discovery to OPC UA, while discovery of device functionality is still missing in this specification. In the year 2016, at the time when I evaluated automatic device discovery, it was therefore not yet well implemented and evaluated, but already a very good basic ingredient to automatically discovering devices in the network. In Chapter 5, I present the basic concepts of the OPC UA Discovery Service set, including its multicast capabilities, and add the concept of hierarchical discovery to allow subdividing the network into multiple workstations and sub-devices. Since the basic implementation was still missing in almost all open source OPC UA implementations, I also implemented this part of the specification and contributed it to the open source community.

3.3 Device Description & Administration Shell

The transport of information between different entities in an industrial automation system is achieved via the used middleware or communication protocol. These middlewares typically do not cover the description and modeling of the transported information, especially assigning higher-level semantic meaning is typically not possible.

To achieve a Plug & Produce system, devices need to come with a specific self-description, so that other components in the system can infer the device's functionality, the semantic meaning, and especially infer the type of information which is transported.

On the level of field bus systems, the Device Description Language (DDL) has been developed in the 1990s to formally describe the parameters of field level device types. A software component can use different field devices described with DDL. DDL has laid the basis for unification and international standardization of device descriptions, e.g., Electronic DDL (EDDL) or Field Device Integration (FDI), all of which are used mainly for field busses [Runde et al., 2013].

These device descriptions focus on the field level, while OPC UA is aiming for a higher goal, to not only act as a device description, but as a component description for all levels, from ERP to Field Bus, while being platform- and manufacturer independent. Since there can be various types of components along these different levels, it is not trivial to get to one single description which fits all use-cases, especially for Plug & Produce. [Schleipen et al., 2015] shows that self-describing devices are essential to get to a Plug & Produce system.

Considering OPC UA gaining popularity, various researchers are aiming for modeling different services of manufacturing systems in OPC UA. Since robots make up a core component of such systems, it is important to explore how OPC UA can be used for robots effectively. The focus hereby lies especially in making robots easily exchangeable in the Industry 4.0 environment.

The recently released OPC UA Companion Specification for Robotics Part 1 [OPC Foundation, 2019a] was a first step in the direction of standardized OPC UA information models for industrial robots. Part 1 focuses on the vertical integration and describes the robot setup (e.g., power trains, axes, joint values), and mainly provides data for predictive maintenance, but does not define robot control interfaces. Getting to a standardized interface for robot control, including all different types of robots and their properties is not trivial. The next parts of this Companion Specification aim for a higher-level task control interface, through which a client can control complete program execution, but not single robot movements.

Previous publications by [Pfrommer et al., 2014], [Ferreira and Lohse, 2012] and [Schleipen et al., 2014] introduce skills and how these can be used in manufacturing systems in a broad sense. All these publications use AutomationML to generate their skill descriptions for an OPC UA server. AutomationML (Automation Markup Language) is an open standard XML-based data format for the storage and exchange of plant design data [Drath et al., 2008]. It can be used to model basic functionalities and skill interfaces, but introduces a further dependency on AutomationML. One of the goals of my thesis is to keep the number of dependencies as low as possible. Every additional dependency introduces a considerable hurdle to adapting a new concept in an SME which may need separate experts for every used technology. As shown throughout this thesis, a generic skill interface can be achieved by solely using OPC UA, without the additional dependency of AutomationML.

ROS is a community-driven middleware with support for various hardware by implementing the corresponding device drivers and providing a Publish/Subscribe interface. Hardware-independent robot control in ROS is implemented via *ros_control* [Meeussen et al., 2017], and similar via a specific Hardware Robot Information Model (HRIM) again based on ROS [Zamalloa et al., 2018]. Same as [Pedersen et al., 2015], where the authors present generic robot skills for manufacturing, *ros_control* only focuses on controlling robots, but does not provide a generic interface for other hardware components, such as grippers or other tools used in robot work cells. Different hardware in ROS is integrated with different interface descriptions, and therefore require distinct control applications. Pedersen also lists various advantages of using skills in combination with production systems, i.e., they are generic and allow product variety, programming skills needs to be intuitive, and they abstract the hardware layer. The presented skill model allows the abstraction of specific robot actions in combination with a task-level programming. Hereby, the authors focus is not specifically on the exchangeability and standardized interfaces as it is for this thesis.

The EU funded RobMoSys project[1] mainly focuses on the composition of robotics applications based on a skill definition integrated in a bigger software architecture, by applying model-driven techniques. It introduces a completely new ecosystem, which may result in a smaller motivation to adapt this complex system, especially for small- and medium-sized enterprises. On the RobMoSys Wiki[2] the authors claim that "OPC UA does not specifically aim for composition and is less suitable for composition of software components. It misses adequate abstractions and concepts, however, composability starts being addressed in OPC UA". As shown as part of this thesis, the statement is not valid, as OPC UA is well suitable for composition and abstraction and additional technologies, introducing more dependencies, are not required.

[1] https://robmosys.eu/

[2] https://robmosys.eu/wiki/other_approaches:opc-ua, last modified: 2019/05/20

It has already been shown that it is possible to automatically generate a device specific OPC UA address space based on various data sources [Girbea et al., 2011] and use self-describing devices for Plug & Produce [Schleipen et al., 2015]. The proposed approach, which uses the combination of custom PLC interfaces and custom algorithms, is still missing a concept to ensure a standardized interface for similar device categories. Other ideas present similar approaches, while they are all lacking a standardized communication and data model infrastructure or focus especially on general manufacturing systems, while neglecting the complexity of robots and their tools [Azaiez et al., 2016; Dürkop et al., 2014; Hammerstingl and Reinhart, 2015; Scheifele et al., 2014].

A technology-independent function interface based on the PLCopen[3] model is described in [Kaspar et al., 2018]. The authors define an OPC UA model based on OPC UA Programs by using PLCopen function blocks. Using these PLCopen based OPC UA programs the authors show that controlling a KUKA iiwa robot is possible through OPC UA. Even switching the gripper during runtime requires little reconfiguration from the user. It is mentioned in their conclusion that the next step is to implement interfaces in which entire robots can be swapped out and continue to fulfill a task, provided they match a specific interface. Abstraction of higher-level functionality (i.e., software components) is not handled at all.

All mentioned related work is mainly focusing on device descriptions for a specific limited use-case, or based on multiple different technologies. To achieve an easy component integration in Plug & Produce systems and low implementation effort, a device description is required which builds preferrably on one single technolgy and is well defined. This description also has to be generic in such a way that it can be used across all levels, from field devices up to the enterprise level. This is where my skill model presented in Chapter 6 has its strength: I show that not only robots from different manufacturers can be completely replaced without changing the client application, but also complete software components.

Similar to the approach of [Kaspar et al., 2018], my colleagues present in [Dorofeev and Zoitl, 2018] skill-based engineering using OPC UA Programs [OPC Foundation, 2019b] in combination with IEC 61499. OPC UA Programs provide a mechanism for the semantic description, invocation, and result feedback of stateful long-running functionalities. These concepts are improved in Chapter 6 by extending OPC UA Programs and their parameterization interface. More specifically, I define a concept to list input and output parameters in a common place and give the presented skills a semantic meaning. In [Dorofeev and Wenger, 2019] it is shown that skill-based architectures provide many advantages in comparison to traditional hierarchical approaches, especially for flexible component exchange.

[3] http://www.plcopen.org/

3.4 Plug & Produce

The term Plug & Produce was shaped around the year 2000 by [Arai et al., 2000]. The authors describe the core concept of Plug & Produce as a methodology which allows to introduce new manufacturing devices easily and quickly into production systems. Since then, various approaches were presented to achieve Plug & Produce systems. In this section, I am listing some of the most relevant research results which existed before the start of my thesis, and list current relevant research results.

Adaptive reconfiguration of the network (flat and multi-level hierarchies) based on device changes is one of the basic principles of Plug & Produce systems. [Knoll, 2001] evaluates the performance of highly dynamic distributed sensor networks and the impact on the network performance in robot applications and proposes an analytical model to evaluate different network types. Based on these research results, optimal placement of system components can be automatically determined. Enhancing the component description with additional semantic data not only allows to improve this automatic placement but is also necessary for the interaction of different system components.

In [Pfrommer et al., 2015], a Plug & Produce system is proposed, which focuses on the theoretical background of mapping skills to products, processes, and resources. Compared to my approach, they use a custom developed model which does not build upon well-established standards such as OPC UA. A combination of Semantic Web technologies and OPC UA is shown in [Jirkovsky et al., 2018]. The authors propose to use a central database to store semantic device information. This may be difficult to achieve on shop floors, in which devices are regularly exchanged. Also, when new devices or device types hit the market, this central database needs constant updates. My approach reduces this disadvantage with self-describing components without a need for central data storage. Other approaches that are based on IEC 61499 function blocks [Dai et al., 2019; Lepuschitz et al., 2011] use custom communication protocols for the connection of components. In [Girbea et al., 2014], a set of services for a service-oriented architecture based on OPC UA is presented. However, automatic discovery was not included. In [Perzylo, Profanter, et al., 2019], my colleagues and I show how OPC UA information models can be automatically transformed to an ontology-based representation, which allows to link the encoded information to other models, e.g., geometry, layout and topology, and process and product models. Particularly relevant for production system engineering and semantic interoperability of manufacturing resources, skills can be annotated with capability meta-models, which provide a semantic understanding of a skill's scope [Perzylo, Grothoff, et al., 2019].

A recent approach to achieve Plug & Play on the sensor level by retrofitting is presented in [Panda et al., 2020], where the authors based their work on the discovery mechanisms I published in my paper [Profanter et al., 2018], which is described in more detail in

Chapter 5. The description of the sensor interface is done through AutomationML in combination with eCl@ss and the sensor data is then published to the cloud. Since this publication focuses on the sensor level, the control interface for devices is not evaluated and still a major gap.

A reference architecture for a Plug & Produce system based on OPC UA and PLCopen is presented in [de Melo and Godoy, 2019; Koziolek et al., 2018], similar to the OpenPnP reference architecture [Koziolek et al., 2019; Koziolek et al., 2020] where the authors base their architecture on a IEC-61131 runtime, or [Panda et al., 2018] focusing on transforming EDDL descriptions into the OPC UA address space. These publications mainly focus on field devices and correspondingly the presented architecture can be used for signal mapping, but not for a generic skill concept. Especially the integration of high-level software components through the same standardized interface is not shown in these publications. Still, the authors show that the presented Plug & Produce concept "enables a faster commissioning process and minimizes the risk for human error due to high automation". In Plug & Produce systems, reducing errors and achieving a fault tolerant control is an important aspect to avoid total failure [Kambhampati et al., 2006]. This aspect gets even higher importance when using industrial robots which can be dynamically plugged into the production line [Maeda et al., 2007].

The publicly funded AutoPnP[4] project presents a model-based Plug & Produce approach especially for the higher-level production planning [Kainz et al., 2013]. The used models describe the type of station (machine or human cell), the current configuration, and deployed production plans. Primitive operation descriptions (e.g., drilling, transport, supply), their attributes (e.g., material, duration) and the modeled capabilities are used for automated higher-level material flow modeling, production planning, and scheduling [Keddis et al., 2014]. The evaluation of the AutoPnP project shows that using premodeled production system components reduces the changeover time on reconfiguration. Still, the initial modeling of the six used Modular Production System (MPS) stations from Festo took the authors approximately a week. In my presented approach I am not only targeting Plug & Produce on the level of stations, but also inside such stations without the need of manually crafting models for each station. Therefore, my approach is an enabler for the AutoPnP project as it also provides automated sub-station composition and standardized interfaces between components.

Another relevant term in the area of flexible industrial automation is the Reconfigurable Manufacturing System (RMS) paradigm. It is defined as a system "for rapid adjustment of production capacity and functionality, in response to new circumstances, by rearrangement or change of its components" [Mehrabi et al., 2000]. Such components can be hardware or software. A robot cell based on the RMS paradigm using ROS is shown in [Gašpar et al., 2020]. As shown in Chapter 4, ROS is not as performant as OPC UA and

[4] http://www.autopnp.com

its semantic expressiveness is limited. This is also, why in the mentioned work the authors use a central MongoDB database for skill exchange, which may not be practicable in the context of self-describing devices and skills.

Previously mentioned publications mainly focus on achieving Plug & Produce via predefined specific variables, and do not focus on a more generic approach for controlling any component. Generic control of components using skills based on OPC UA Programs is also shown in [Zimmermann et al., 2019] and was published at the same time as my relevant paper describing generic robot skills. The presented approach is similar to my approach with one significant difference: The focus of my skill model is not only on hardware-specific functionalities, but also software components, while keeping the same interface description. My proposed solution aims at providing a generic system architecture based on standardized skill models that can be applied to any type of component in the system, be it hardware or software.

In doing so, I focus on reusing well-established standards such as OPC UA and keeping the number of inter-dependencies as low as possible, e.g., by building on the discovery mechanisms of OPC UA and supporting standardization activities in various active working groups.

4 Middleware Evaluation

Partial results of the presented work in this chapter are published in my peer-reviewed publication [Profanter, Tekat, et al., 2019]. I am the main author of this publication, and my main contributions are in further detail described in this chapter. Some figures created and partial text written by me in this publication are directly included in this chapter.

This chapter presents an extensive evaluation to answer the research question: "Which communication protocol is the best suitable one for the Plug& Produce use-case?".

Middlewares are used as a general communication platform for any components in a system. Figure 4.1 is already known from Section 1.5 and highlights the focus of this chapter.

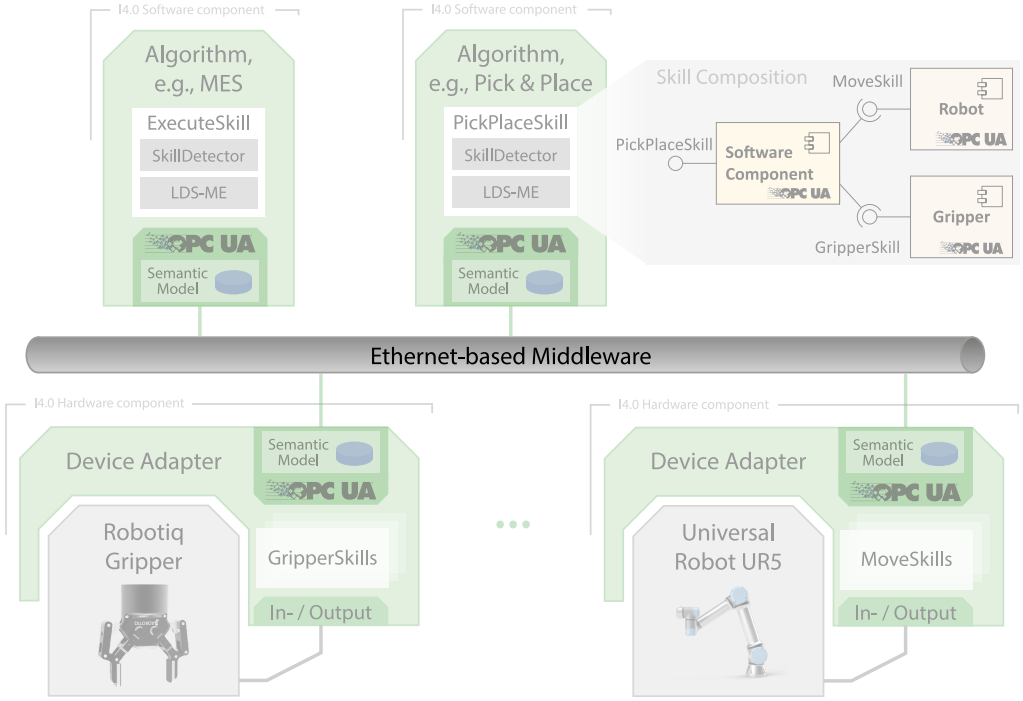


Figure 4.1: Exemplified system setup with different Industry 4.0 components. Chapter 4 focuses on the Ethernet-based Middleware part.

In the first section, I present the definition and basic requirements for a middleware in the domain of industrial automation. This is followed by an overall overview of the main characteristics on a subset of specific middlewares which have a high market relevance in the domain of industrial automation and IoT: OPC UA, DDS, ROS, and MQTT [Balador et al., 2017; Kaur and Kaur, 2017; Tsardoulias and Mitkas, 2017; Wollschlaeger et al., 2017].

This chapter is completed with an evaluation of these middlewares regarding their package overhead of each underlying protocol and the description of my custom developed software library for middleware performance evaluation, which can be adapted to any protocol. It delivers the corresponding performance values in a common format to compare it to other results. The evaluation includes performance measurements on different hardware to identify the suitability especially for real-time control of industrial robot systems which require low latency and reliable transport. During the evaluation execution, various system parameters and round-trip times were measured and compared afterwards.

4.1 Middleware Definition & Requirements

In [Bernstein, 1993] a middleware is defined as follows:

> To help solve customers' heterogeneity and distribution problems, and thereby enable the implementation of an information utility, vendors are offering distributed system services that have standard programming interfaces and protocols. These services are called middleware, because they sit "in the middle", layering above the OS and networking software and below industry-specific applications.

This definition includes some important points for the industrial automation domain. A middleware is a *collection of services* to abstract the operating system and networking software. These services must support standard protocols for *vendor independence* to be used in distributed systems. A middleware bridges the gap between software applications of various programming languages and individual subsystems on different hardware platforms and operating systems.

Over the last two decades, different middlewares and standardized protocols have been developed. Some of them with a strong focus on real-time communication, while others focus on Ethernet-based protocols which are slowly replacing conventional and proprietary field bus communication [Decotignie, 2005; Neumann, 2007; Rüth et al., 2017].

Since my goal is to cover all hierarchical levels of the automation pyramid, an **ethernet-based** middleware, including **platform independence** of the operating system, is mandatory for my proposed Plug & Produce system. A major advantage of Ethernet-based protocols over conventional field bus communication is the required hardware: These protocols re-use already existing Ethernet hardware and therefore this middleware communication can be more easily integrated into existing networks. A specific field bus normally requires special expensive hardware and custom protocol implementations. Additional requirements for a future-proof middleware are **privacy, integrity and security**. Information exchange with other system components is security critical, therefore encryption and authentication is mandatory for generic systems. **Information management and semantic modeling** is another crucial requirement to transparently process data from different sources and provide this data in an understandable format to other components. **Modularity and standardization** allows replacing system components and therefore subsystems can rely on standardized interfaces for modularity. This leads to low-cost integration of subsystems and includes the ability to react on changed requirements or new subsystems. These new subsystems need to be seamlessly integrated by providing **service detection and orchestration**. Since the number of system components may be large, **scalability** is also a significant performance factor of a suitable middleware.

Some of these requirements are also listed in [Trunzer et al., 2019] as basic requirements for an Industry 4.0 architecture. Supporting all these requirements makes a specific middleware an ideal basis for my proposed Plug & Produce architecture. In Section 4.3 I compare the middlewares based on their features and implemented requirements. The following section gives an overview on the middlewares I am using for the performance comparison.

4.2 Middleware Overview

A middleware is typically developed for a specific use-case in mind. Therefore, every protocol implementation has its own strengths and downsides in various domains. This section shows the main characteristics of the chosen middlewares with a focus on the IoT and Industry 4.0 (I4.0) domains.

4.2.1 OPC Unified Architecture (OPC UA)

Open Platform Communications Unified Architecture (OPC UA) is a service-oriented communication protocol for machine-to-machine communication. It has its origin in

industrial automation and is emerging into many other domains. OPC UA is the successor of the OPC Classic communication protocol. Both are standardized by the OPC Foundation[1]. OPC Classic was only available for the Microsoft Windows platform, as it was using the COM/DCOM (Distributed Component Object Model) for data exchange, compared to OPC UA which is platform independent.

In 2008 the OPC Foundation has released the first version of the OPC UA specification, which improves many features from the OPC Classic framework. The main goal of OPC UA is to provide an open cross-platform communication protocol with a strong focus on a semantic information model to describe the transferred data. The first specification was also released as the IEC 62541 (International Electrotechnical Commission) specification. Further extended and corrected specification documents are freely available on the official OPC Foundation web page[2].

OPC UA has a wide distribution and high adaption momentum in European manufacturing and is gaining more and more importance worldwide [Drahos et al., 2018]. A significant difference to other middlewares is its strong semantic data description model: The base specification defines the Core Information Model of OPC UA while additional Companion Specifications allow the extension of the base model as shown in Figure 4.2.

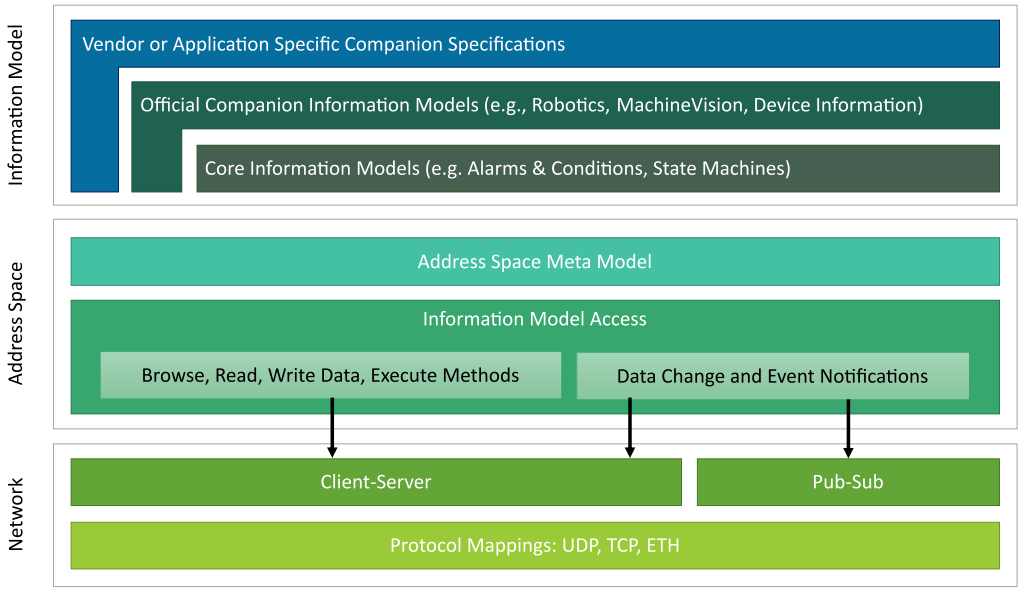


Figure 4.2: OPC UA has the strength in its information model capabilities. The Core Information Models can be extended via Companion Specifications.

[1]https://opcfoundation.org/
[2]http://reference.opcfoundation.org/

Figure 4.2 shows the main layers of OPC UA. On the network side, OPC UA supports two main messaging patterns: Request-Response and Publish-Subscribe. In Request-Response, there is one TCP connection for one communication channel, while Publish-Subscribe uses UDP multicast to send data from one publisher to many subscribers. In addition to TCP and UDP, OPC UA also supports other lower-level communication methods, like plain Ethernet frames (ETH). Since UDP itself does not support Quality of Service (QoS) by default, OPC UA is since 2019 gradually adapting TSN to achieve real-time ethernet capabilities similar to a proprietary field bus.

On top of the Network Layer is the Address Space. It is the data storage, also called information model, of an OPC UA instance. OPC UA provides different services to access the information model inside the server. The main access services are browsing for information, reading, and writing variables, and executing methods. More advanced services allow the modification of the information model itself by adding or deleting nodes and references. In OPC UA every data element is represented as a node in a directed graph, where the edges represent references. Every node has properties specific to its node type. This is similar to typical graph databases. The address space meta model defines the structure of the OPC UA address space, i.e., different node types (Objects, Variables, References, Views), their properties, and reference types. The address space services are extended with data and event notifications which notify the client or subscriber on changed data values. A detailed explanation of the OPC UA address space concepts and modeling notation is given in the Appendix A.

A set of different information models builds up the address space meta model. There are various core information models specified as part of the OPC UA specification, which are extended by official Companion Specifications, e.g., for Robotics or Device Information. An important feature of OPC UA is the possibility to extend existing information models with vendor-specific custom models.

A deeper insight into the OPC UA information modelling can be found in Section 6.4.1 and on my webpage[3]. Major parts of that website's content were written by myself as part of this doctoral thesis. Other topics on my website include getting started with OPC UA, implementation hints, and information model documentation.

4.2.2 Data Distribution Service (DDS)

DDS is a middleware with a focus on highly dynamic distributed systems. It is standardized by the Object Management Group (OMG)[4]. According to the OMG's website,

[3]https://opcua.rocks/custom-information-models/
[4]https://portals.omg.org/dds/

DDS is one of many protocols used in industry sectors such as railway networks, air traffic control, smart energy, medical services, military and aerospace, and industrial automation[5].

Similar to OPC UA, every DDS participant is started in its own process, which can be a publisher, subscriber, or both to decouple the participants in space, time, and flow between publishers and subscribers. DDS provides a typed interface, which can be used to read and write data. To achieve real-time capabilities, it has built-in Quality of Service (QoS) policy options, which are used for the communication channel [Pardo-Castellote, 2003].

Due to its data centric architecture, DDS uses a global data space as information storage where any participant can read and write. This data is not specifically stored inside one process but located in a distributed way across the publishers. Depending on the QoS policy, a specific amount of data and historical values are stored or cached. DDS supports many QoS policies, where mostly used ones are: durability (store data for future subscribers), lifespan (how long is the data valid), presentation (order of received packages), reliability (best-effort or reliable transmission), and deadlines (maximum time until new data is expected).

Single data values are identified in DDS via topics: It has a unique key, i.e., string name, and it is bound to a specific type. Therefore, each published data package describes itself, whereas in OPC UA the client browses the information model and infers the data description beforehand, without the need of reading and parsing the type description repeatedly.

DDS supports dynamic discovery without a central instance, similar to OPC UA (see Chapter 5). This discovery process allows to automatically find matches between a publisher and subscriber topic, independent of the process location.

4.2.3 Robot Operating System (ROS)

ROS is not an operating system in the sense of process scheduling, but more like a structured communication interface on top of a host's operating system. The main goals of ROS can be summarized as: peer-to-peer, tools-based, multi-lingual, thin, free and open-source [Quigley et al., 2009]. It was originally developed by Willow Garage and now maintained by the Open Source Robotics Foundation (OSRF) together with a large community[6].

[5] https://www.dds-foundation.org/who-is-using-dds-2/
[6] http://www.ros.org/

The main target of ROS is the research community as it provides a large ecosystem of tools and algorithms, called ROS packages, encouraging collaborative robotics software development. These ROS packages are mostly available as open-source software and are one of the strong points of this middleware.

Similar to DDS, ROS uses the Publish-Subscribe pattern, but on established TCP connections instead of connection-less UDP. A process which is publishing, or subscribing is called ROS node. The central *roscore* component manages the orchestration and discovery for ROS topics, which is a specific stream of data. A ROS topic is simply identified by its name. The data structure transferred inside a topic must be known to all subscribers, and typically cannot be updated on-the-fly during runtime.

ROS2, the successor of ROS, was released in the year 2019. In ROS2 the communication protocol implementation changed from a proprietary protocol to DDS, therefore it is built on top of the DDS protocol itself and supports various DDS implementations.

4.2.4 Message Queuing Telemetry Transport (MQTT)

MQTT is a lightweight Publish-Subscribe middleware which is widely used in the IoT domain. MQTT is open-source and focuses on a small code footprint and efficient bandwidth usage while handling high-latency and low bandwidth network connections.

The Organization for the Advancement of Structured Information Standards (OASIS) defined MQTT in the year 2014 as the protocol for IoT[7].

The central component of MQTT is the so-called broker. It stores all the data from every communication participant. This allows small devices to just fire and forget new data values, and therefore such devices do not need to store data over a long period. The data itself is grouped hierarchically, similar as it is achieved with DDS, and multiple devices can publish on the same topic. MQTT supports a basic set of QoS policies, e.g., to define how often a message should be re-sent until it is acknowledged, or if the data should be cached on the broker side.

[7] https://www.oasis-open.org/news/pr/foundational-iot-messaging-protocol-mqtt-becomes-international-oasis-standard, accessed July 2020

4.3 Feature Comparison

An overview of the most important features of the middlewares is shown in Table 4.1. These features are described in more detail in the following paragraphs.

Table 4.1: Comparison of the protocols used in the evaluation and their main features.

	OPC UA	**ROS**	**DDS**	**MQTT**
Communication	TCP, UDP, ETH	TCP, UDP	TCP, UDP, SHM	TCP
Patterns	Request-Response, Pub-Sub	Request-Response, Pub-Sub	(Request-Response), Pub-Sub	Pub-Sub
QoS	(Yes)	No	Yes	Yes
Authentication	User, PKI	(Mac)	PKI	User, PKI
Encryption	Yes	No	Yes	Yes
Standard API	No	No	Yes	No
Semantic Data	Yes	No	No	No

Usage domain OPC UA focuses on device interoperability, therefore its entire specification was defined with this in mind. It is an ideal candidate for systems where many devices are used. In comparison, DDS has a strong focus on software integration in mainly a single system type, where the hardware does not change over time. This is one of the main reasons why OPC UA is adopting really fast in the industrial automation domain: Typical productions lines are built up using different types of devices in highly flexible environments. In air traffic control, where DDS is often used, all system components are mainly from one manufacturer, and they are fine-tuned for this finalized system. The focus of ROS is on hardware abstraction and integration of different algorithms into one big system, hereby targeting the research community. Many robotic research laboratories and robotics applications use ROS for a quick setup and algorithm re-usability. The number of commercial products supporting ROS is still extremely low. In contrast, MQTT is focusing on one specific use-case: low network bandwidth and latency requirements. Therefore, MQTT is not as feature-rich as other presented Middlewares.

Communication The main task of a middleware is to exchange data with other system components. There exist many ways to achieve this task. The most typical and universally used way is to establish a TCP connection between two components. TCP has the main advantage that it handles connection parameters, package resending and package ordering as a built-in feature. TCP payload is encapsulated by an Internet Protocol (IP) frame, which holds data like source and destination port and IP addresses.

This IP payload is carried by an Ethernet (ETH, IEEE 802.3) frame to include the source and destination MAC (Media-Access-Control) address and prioritizing tags. All the presented middlewares support TCP as its communication type.
UDP is also based on IP but does not establish a dedicated connection between two endpoints or handles data retransmission. Since no established connection is needed, UDP is very suitable for Publish-Subscribe setups where a single publisher can send data to multiple subscribers. MQTT is the only middleware which does not provide a UDP connection. All data is sent via TCP to the central broker which distributes the data to all subscribers. MQTT-SN is an extension for MQTT adding UDP [Stanford-Clark and Truong, 2013]. In ROS all clients must support TCPROS, which is a ROS-specific TCP protocol, and optionally they can also support UDPROS based on UDP which is, status as of July 2020, only implemented in the C++ derivative of ROS.
OPC UA has an additional communication protocol support for direct sending of *Ethernet 802.3 (ETH)* frames without the usage of IP and TCP or UDP. The specification allows to directly send ethernet frames, which is especially required in combination with TSN [Pfrommer et al., 2018]. TSN adds real-time networking capabilities to OPC UA which can be used for QoS.
DDS supports *shared memory (SHM)* data exchange if two DDS participants are running on the same host system. Using shared memory instead of link-local connections on the same host reduces the overhead by removing the necessity to send data packages down to the network layer inside the operating system. Some DDS implementations handle the switch between SHM or network-based communication transparently to the implemented application.

Messaging pattern There are two main network-oriented architectural patterns, which describe the way how data is exchanged [Hohpe and Woolf, 2003]. Using *Request-Response*, one side sends a request to a specific entity which is directly responding to this request. It is typically implemented as synchronous communication and especially common in Client-Server architectures.
The *Publish-Subscribe (Pub-Sub)* pattern is a one-to-many connection: A publisher which sends the data does not necessarily know its subscribers. Depending on the specific implementation, the number of publishers and subscribers for the same data is not necessarily limited. Multiple publishers can publish to one subscriber, or there are multiple subscribers for a publisher. This pattern introduces better network scalability but reduces the component flexibility on the published data structure since all participants need to be updated, while in bigger system setups they are sometimes not known.
MQTT, ROS, and DDS mainly use the Pub-Sub pattern for data exchange. OPC UA was first specified for the Request-Response pattern, more specifically Client-Server, and in the year 2018 extended with the Pub-Sub pattern. Aside of Pub-Sub, ROS natively provides a Request-Response pattern, called ROS Service. DDS has a defined standard for Request-Response, but it is only available in a few implementations.

Quality of Service (QoS) QoS defines policies which can be applied to the data stream to assign different priorities or latency requirements to data packages. This allows applications to guarantee a specific quality of their provided communication service. Such QoS policies are important for real-time applications, especially in safety-critical environments. A higher QoS requirement typically leads to higher protocol and communication overhead.
DDS provides an extensive set of QoS policies, e.g., to define deadlines and lifespan, durability for late-joining participants, reliability for re-sending data on loss, and many more. MQTT defines a basic set of three QoS levels: level zero is best-effort delivery without the guarantee that the data is delivered. Level one guarantees that the message is delivered at least once to the receiver but could also be sent multiple times. Level two guarantees that each message is received only once by the intended recipients. OPC UA does not explicitly define QoS in its specification, but by relying on TSN, real-time capability can be achieved in Pub-Sub communication. The TSN specification defines many features for real-time networks. ROS does not define any QoS at all.

Authentication & Encryption Nowadays, every connected system should include authentication mechanisms for participants, to ensure they are allowed to receive and especially send data to other components in the system. This is crucial for safety-critical systems, especially in connection with hardware devices which may harm human beings. In addition to the authentication, the data itself should be encrypted to avoid clear-text data in the network stream for additional security.
OPC UA and MQTT support authentication via a username and password or by using a private key infrastructure (PKI). DDS only supports PKI authentication. ROS has even less authentication capabilities: it only supports MAC address-based authentication by using third-party packages.
Compared to DDS, MQTT and OPC UA, ROS is the only middleware which does not support application layer encryption: Data is sent as plaintext to other participants and is therefore not recommended using it in the industrial domain.

Standard API & Implementation A standard Application Programming Interface (API) does not influence the communication itself, but rather on the implementation side. DDS is the only middleware in the list which defines a standard API in its specification. This definition regulates how methods and data structures must be named. The goal is to achieve an easy exchange of the underlying implementation without changing the source code. My experiment with two DDS implementations (RTI Connext and PrismTech OpenSplice) using C++ showed that the main functionalities are exchangeable. As soon as it gets to more specific configuration settings, the API of both implementations differ, therefore such a standard API only provides a limited advantage. The

corresponding example source code is available under the Public Domain License on GitHub[8]

Semantic Data Representation To achieve semantic interoperability, flexible system integration requires a semantic description of the transported information to automatically infer its meaning without the need to code everything by hand. OPC UA has a feature-rich semantic annotation of the data by connecting available data nodes via specific reference types with each other. This semantic model is called OPC UA Information Model and can be mapped to typical semantic graph databases as it is shown in [Perzylo, Profanter, et al., 2019]. ROS, DDS, and MQTT use topic names to give the data a specific meaning. This string name must be parsed by clients to infer its meaning which could lead to misinterpretation or even wrong data matching.

4.4 Package Overhead

Every middleware protocol needs to include some meta information in the transmitted data to correctly decode and interpret the data on the remote side. These package headers require additional bits or bytes to be added to the data payload itself and therefore it is introducing an additional package overhead. The complete data package (header and payload) is then forwarded to the corresponding underlying protocol, e.g., TCP or UDP, to be transmitted.

Depending on the used protocol, the size of the package header differs, and therefore has a direct influence on the maximum theoretical bandwidth which can be reached. This section gives an overview of the package overhead for OPC UA, ROS, DDS, and MQTT based on the corresponding protocol specification. The evaluation was performed on different payload sizes (0 bytes, 100 bytes, 1000 bytes, 10000 bytes) since the header size may vary in these different cases.

Table 4.2 shows the protocol payload size which is passed from the middleware (OSI Layer 5, Session) to the UDP/TCP connection (OSI Layer 4, Transport). Since this is the payload for the transport protocol, its size is not influenced by lower-level protocols, e.g., if the ethernet frame needs to be split into multiple frames due to its Maximum Transmission Unit (MTU, typically 1500 bytes). The table also includes the required number of bytes, which are sent at the beginning of a connection establishment and at the end for connection shutdown. The following values for each middleware are

[8] https://github.com/Pro/dds-temperature

constructed from the protocol definition and then verified using Wireshark[9] network protocol analyzer.

Table 4.2: Package size in bytes transmitted as TCP/UDP payload for each protocol with given payload in bytes. The protocol overhead is calculated by the difference between data payload size and the TCP/UDP payload. The connection column is the sum of bytes for connection establishment and shutdown.

Payload	0	100	1000	10000	Connection
OPC UA C/S	96	196	1096	10 096	632
ROS	8	108	1008	10 008	8915
DDS	88	188	1088	10 088	8348
MQTT	5	105	1006	10 006	17

OPC UA To achieve comparable results between the used protocols, a simple write request to a variable was implemented, without the use of encryption. Encryption would add additional overhead, which would invalidate the comparability of the results.
The following steps are performed in OPC UA before and after a write request or any other service call can be executed:

1. open secure channel (OpenSecureChannelRequest): 132 bytes
2. get available endpoints (GetEndpointsRequest): 93 bytes
3. create a session (CreateSessionRequest): 138 bytes. The size depends on the hostname, in this case `localhost:4840` was used
4. activate the session (ActivateSessionRequest): 137 bytes. The size depends on the identity token length, in this case `open62541-anonymous-policy` was used
5. *send write request*
6. close the session (CloseSessionRequest): 75 bytes
7. close the secure channel (CloseSecureChannelRequest): 57 bytes
8. Sum of bytes for connection overhead: 632 bytes

ROS During the startup phase, every ROS node sends an XMLRPC request to the central roscore to exchange information about the current system state and available publisher and subscriber for the node's topics. If there is a matching topic from another node, a dedicated TCP connection is established between these two nodes using the TCPROS protocol. During the shutdown phase, the ROS node sends another XMLRPC request to the roscore to announce that it will go offline.

[9] https://www.wireshark.org/

1. outgoing XMLRPC requests to register with roscore: 5693 bytes
2. Subscriber node connects to Publisher via TCPROS
3. Publisher node sends publishing info: 176 bytes
4. *Publish data*
5. outgoing XMLRPC requests for the shutdown: 3046 bytes
6. Sum of bytes for connection overhead: 8915 bytes

DDS DDS does not use a central component but relies on its discovery process to find other participants in the same subnet. Since this discovery process is not standardized in the specification, the sequence and payload during this discovery process heavily depends on the used implementation. For the following evaluation, the eProsima Fast RTPS implementation was used. In the discovery phase, the participant sends out a multicast message to discover other participants and periodically sends a heartbeat. After two participants discovered each other, they exchange information about published and provided topics. This whole process sums up to 8348 bytes encapsulated in RTPS (Real-Time Publish Subscribe) packages delivered as UDP payload.

1. Send multicast message to discover participants
2. Periodically send heartbeat
3. Receive response of subscriber
4. *Publish data*
5. Disconnect from subscriber

MQTT MQTT uses its own TCP-based lightweight binary protocol. Its lightweight implementation can also be seen by looking at the number of bytes required for connection establishment to the broker. Note that the number of bytes transmitted in the following steps is dependent on the topic name. In this calculation a topic name with one single character was used.

1. Connect command: 15 bytes
2. *Publish data*
3. Disconnect request: 2 bytes
4. Sum of bytes for connection overhead: 17 bytes

The overall evaluation of package overhead shows that MQTT has the smallest overhead for connection establishment and the smallest overhead when sending out data messages. The package overhead of MQTT is almost the same as for ROS: Both protocols use dedicated TCP channels for different topics. This avoids the necessity to send additional metadata, as the channel is internally annotated with the corresponding metadata during the connection establishment. Yet, ROS requires the highest number

of bytes for connection handling. This is because XMLRPC messages are uncompressed XML text, which is directly sent over the network. The package overhead of DDS and OPC UA for data messages is higher, since these two protocols only use one single TCP connection, where different data values can be sent through the same channel, enclosed with the corresponding data description. This requires additional metadata to identify the data type. OPC UA is on the last place when it comes to the package overhead since it provides the highest amount of additional annotation data for each data package.
Compared to ROS and MQTT, DDS and OPC UA support the additional feature to transmit additional diagnostic information with every package, e.g., to include the server performance or other vendor-dependent diagnostics. In ROS and MQTT this data needs to be collected separately.

4.5 Performance Evaluation

To test the performance of OPC UA, DDS, ROS, and MQTT, I developed a test suite which allows testing different protocol implementations with the same testing mechanisms. The test suite is available as open-source software on my GitHub Account[10]. It is designed to deliver reproducible results with a single command, which also logs various key performance values (CPU, memory, RTT) in a file for later statistical evaluation.

4.5.1 Testing Setup

The hardware setup for the performance evaluation is shown in Figure 4.3. It is composed of a Linux client and Linux server PC which are connected via a Gigabit Ethernet switch (TP-Link TL-SG1024DE). Both Linux machines have the following specification: Intel i7-8700K CPU with 3.70 GHz and 64 GB RAM running on Ubuntu 16.04.4 with Preempt-RT Kernel 4.14.59-rt37. The real-time kernel was used to ensure high performance and reproducibility on the tests. An initial evaluation of the setup, independent of middleware protocols, results in an average round-trip time (RTT) ping of 0.35 ms between the two Linux machines. A measurement of the bandwidth using the Linux `iperf` command resulted in a value of up to 724 Mbit/s.

To measure the round-trip time (RTT) of a package, the request-reply pattern was implemented, where the server provides methods which can be called from the client side. For MQTT, DDS, and OPC UA Pub-Sub the request-reply pattern was implemented by using two separate topics: One for sending the request and the other for the reply. In

[10] https://github.com/Pro/middleware_evaluation

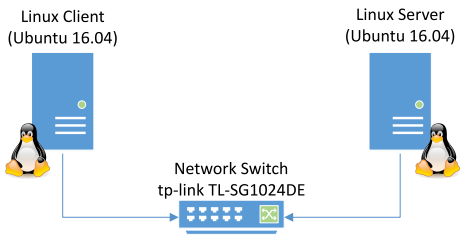


Figure 4.3: Middleware performance test setup to measure CPU usage, RAM usage, messages per second, and round-trip-time (RTT).

addition to the RTT, the following metrics are determined during the test run: CPU usage, RAM usage, messages per second.

All tests were executed with two distinct modes: The ACK mode sends a request with the given size and waits for a simple acknowledgement message (1 byte), which is useful to measure the response time for a single message. In the Echo mode, the transmitted package is echoed by the remote side in the response package to measure the throughput. The tests are executed sequentially repeating every payload 5000 times without any waiting time in between. As soon as the acknowledgement or echo response is received, the next package is sent. The first payload size is 2 bytes, repeated 5000 times, and then the payload is doubled for the next step, until a payload size of 32768 bytes is reached.

In addition to the ACK and Echo modes, a resting RTT test was performed: Before sending the next ACK request, the client waits for a random time between 0 and 3 seconds. This step is repeated as often as possible within 30 seconds and then the RTT is measured.

To test the performance of the middlewares in non-ideal scenarios, where other processes on the end-device cause high CPU load or there is high traffic on the network, third party tools were used to generate extensive network traffic and additional CPU load. Ostinato is a traffic generator which can create sequential interleaved streams of different protocols at different rates to generate an artificial network load [Srivastava et al., 2014]. This results in a full capacity utilization of the network stack. The `stress`[11] tool was used to create artificial CPU load. It is designed to apply configurable CPU, memory, I/O, and disk stress on the system.

As a final step, the performance of the middleware was evaluated, while 500 server instances of the same protocol are started at the same time on the server machine. The client machine will then simultaneously send 10 packages with a payload of 10 240 bytes to 10 nodes running on the server machine. These 10 nodes immediately forward the re-

[11] https://people.seas.harvard.edu/~apw/stress/

ceived package to the next node on the same machine. The last step is repeated 50 times on the server side, which results in 10 internal sending streams running in parallel. The last node sends the received messages back to the client, which will then measure the complete RTT of all 10 messages. The whole process is repeated 100 times to get an average value and eliminate outliers. This test will show the suitability of the protocol for the use case, where a lot of nodes are running on the same host and data is exchanged primarily between these nodes. This setup is often used with ROS, e.g., where a ROS node is reading a camera image, another node subscribes to this image, applies preprocessing, and outputs the result in a new topic.

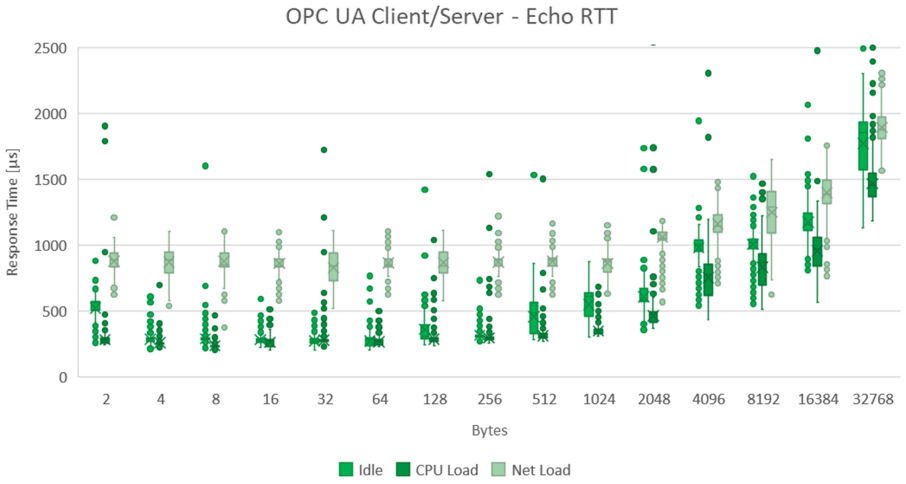
The following implementations of each protocol were used, all of which are available as open-source software. The used version of every implementation is the latest stable version in early November 2018. Commercial stacks exist for OPC UA, DDS, and MQTT, but are not included in this evaluation.

- **open62541:** OPC UA implementation. License MPL2.0. Version 0.3-rc2 for Client-Server, and Version df58cf8 of the master branch for Pub-Sub which was, as of November 2018, not included in the release version yet.
- **ROS C++:** ROS implementation. License BSD. Version Kinetic Kame
- **eProsima Fast RTPS:** DDS implementation. License Apache 2, Version 1.6.0
- **Eclipse Paho MQTT C:** MQTT implementation. License EPL1.0. Version 1.2.1

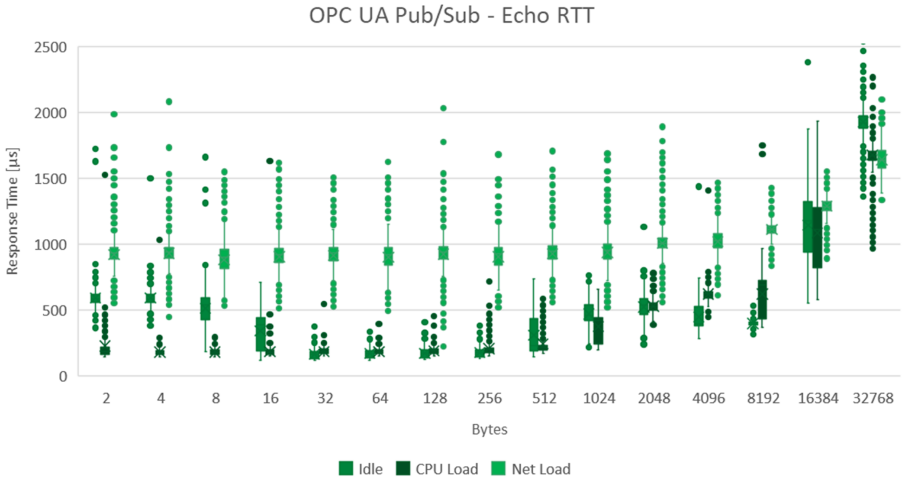
4.5.2 Performance Results

This subsection presents the results from the testing setup as described previously. I used the most common C/C++ open-source implementations of every protocol. For some protocols there exist other implementations in different programming languages, commerial and open-source. Therefore, the results are only valid for the used implementation and may look different for other implementations of the same protocol.

Figures 4.4 to 4.7 show an excerpt of some of the most interesting results from the executed tests. Visualization as box plots was used, which allows to see the median value (marked with a cross), upper and lower quartiles, and the variability outside the quartiles (marked by whiskers). Single points indicate outliers. The corresponding numeric values for the average round-trip-time (RTT) are also shown in Table 4.3. In addition to the protocols described in previous sections, two simple echo/ACK servers were implemented in C that listen for TCP and UDP connections and return the received data to the sender (echo) or acknowledge (ACK) the data by sending one single byte as a response. The results of these raw TCP and UDP implementations are included in the figures and show the best reachable RTT for the corresponding tests without any overhead from a specific middleware implementation.



(a) Echo RTT for different data message size using OPC UA Client-Server via TCP on Idle, CPU Load, and Network Load.



(b) Echo RTT for different data message size using OPC UA Publish-Subscribe via UDP on Idle, CPU Load, and Network Load.

Figure 4.4: Plots showing the RTT results for OPC UA Client-Server, and OPC UA Publish-Subscribe for an excerpt of the collected data.

Figures 4.4a, 4.4b and 4.5a to 4.5c show the box plots for all four protocols with the RTT of the message which is being sent to the server side and immediately returned by it. The first column of box plots shows the values for the system idle state (no additional load on the CPU or network). The second box column shows the RTT with 100 percent CPU server load, and the last box column shows the RTT under high network load.

Comparing the values for the server in idle state (first box column) versus the one with high CPU load (second box column), the test results indicate that the RTT of OPC UA and ROS does not correlate to the CPU load of the host and is nearly the same as in the

Table 4.3: Average RTT in microseconds for echo and ACK mode with various payloads (in bytes) as visualized in Figures 4.4a, 4.4b and 4.5a to 4.5c. Lower numbers are better. The minimum value of each load type across all middlewares (excluding raw TCP and UDP) is marked as: Min Idle, Min CPU, Min Net, Min ACK.

Mode	Middleware	Load	2	4	8	16	32	64	128	256	512	1024	2048	4096	8192	16 384	32 768
Echo	OPC UA C/S	Idle	520	283	288	280	270	270	360	313	456	546	606	978	1003	1171	1770
		CPU	279	258	233	260	276	265	284	293	310	346	461	752	830	955	1467
		Net	875	869	876	858	827	864	866	871	875	860	1056	1158	1248	1397	1894
	OPC UA P/S	Idle	590	589	510	338	162	270	169	174	305	469	521	446	395	1142	1924
		CPU	224	177	178	182	183	265	188	206	242	373	529	620	618	1083	1676
		Net	919	930	880	902	911	864	921	901	924	936	1003	1019	1110	1286	1641
	MQTT	Idle	584	629	645	641	581	601	622	699	694	725	716	920	1186	1225	1511
		CPU	912	905	912	917	911	934	945	944	964	1015	1094	1540	1582	1504	1843
		Net	1017	1042	1038	1033	1040	1027	1054	1039	1085	1135	1239	1347	1456	1622	2029
	ROS	Idle	618	628	446	396	398	182	296	319	495	814	698	1007	1273	1486	1823
		CPU	256	320	295	252	190	305	332	317	359	421	563	1077	1114	952	1427
		Net	1017	1014	1009	1012	982	1021	1011	963	1040	1063	1124	1263	1343	1488	1884
	DDS	Idle	726	234	236	243	247	252	251	279	270	314	343	454	495	802	1144
		CPU	624	631	626	625	624	628	622	644	659	442	351	464	489	782	1050
		Net	1091	1086	1070	1111	1063	1088	1102	1099	1123	1114	1184	1190	1270	1432	1806
	TCP	Idle	250	253	284	303	254	255	307	270	289	315	273	823	811	892	1261
	UDP	Idle	241	231	208	213	208	162	168	198	238	193	263	263	268	893	1398
ACK	OPC UA C/S	Idle	383	354	301	294	306	291	293	321	310	305	481	710	681	969	1033
	OPC UA P/S	Idle	185	180	199	207	196	190	180	195	215	279	222	277	298	785	1265
	MQTT	Idle	647	602	556	518	606	626	636	646	650	643	629	575	556	740	800
	ROS	Idle	642	537	321	168	157	248	623	608	437	197	226	871	992	1192	1377
	DDS	Idle	320	228	234	236	241	243	244	242	252	272	275	355	343	514	784
	TCP	Idle	235	243	283	187	239	279	196	208	189	237	58	71	81	167	338
	UDP	Idle	259	223	220	190	155	153	164	169	174	215	359	317	255	502	725

idle state. I used a Preempt-RT Linux kernel where all middleware-related processes are set to the highest priority. MQTT and DDS show a significant slowdown of approximately 300 µs compared to the idle state. This leads to the conclusion, that the Paho MQTT and eProsima Fast RTPS implementations require more CPU power to process the messages and are thus slowed down when the CPU utilization is at 100 percent.

The results of the RTT during high network load (second box column) show that protocols using TCP (OPC UA Client-Server, MQTT) were less influenced by a network interface running at maximum capacity than protocols using UDP (OPC UA Pub-Sub, DDS, ROS). All protocols show a slowdown of more than 400 µs.

The combined view in Figure 4.6a and 4.6b shows the direct comparison of the protocols, including a simple TCP and UDP echo/ACK server implemented in C. The raw UDP implementation is the fastest approach for exchanging messages of small sizes, which are either acknowledged or echoed back, closely followed by TCP, which has a better performance for bigger message sizes. Excluding the results of TCP and UDP, the open62541 OPC UA Pub-Sub and Client-Server implementation described in [Pfrommer et al., 2018] is the fastest middleware for almost all package sizes independent of the ap-

plied system load. eProsima Fast RTPS is in second place, followed by ROS, and then MQTT. The diagrams also indicate that MQTT has the highest number of outliers.

To investigate further which components of a middleware have the highest impact on the RTT, I executed additional tests using the OpenDDS middleware. It features a zero-copy or memory allocation mode for sending back the echo message. This has shown that the way messages are read from the socket and then forwarded to the user code has a high impact on the RTT. Typically, the received message is duplicated and moved in memory multiple times from the operating system's socket until it reaches the implementation. Memory allocations are an expensive operation. Additionally, OpenDDS was significantly slower than Fast RTPS, which shows that the serialization of messages may also lead to higher RTT. The eProsima Fast RTPS and open62541 OPC UA implementations pass constant data pointers to the user code, whereas MQTT and ROS use multiple copy operations to duplicate the data.

Figure 4.7a shows the RTT for sequential requests, where the client waited a random number of seconds (between zero and three) to send one single request (4 bytes payload) and waited for the ACK messages. This test was repeated 100 times to get an averaged value. These results indicate that DDS needs more time to reactivate from the resting state compared to other middleware protocols. It could be the case that the eProsima Fast RTPS implementation is closing the connection sockets after a short period of resting time, and therefore new connections take more time to set up. For OPC UA, MQTT, and ROS, the values are similar to the ones in Figure 4.6b, which shows the ACK RTT.

At last, 500 instances running on the same remote host was evaluated, which exchange a lot of data between each other, as described in Section 4.5.1. For OPC UA Client-Server, 500 OPC UA servers using different enpoint ports were started on the remote host. The client sends 10 simultaneous write requests with a payload of 10 240 bytes and every server forwards this request to the next local server in line, resulting in 50 write requests per package, multiplied by 10 simultaneous streams. The last 10 servers return the value to the client, which measures the overall time. For OPC UA Pub-Sub, MQTT, ROS, and DDS, the same procedure is achieved by using topics with different names and starting multiple processes. One of the biggest issues for this test was the resource usage of ROS and DDS. Starting 500 ROS nodes forces the CPU to work at full capacity and filled 60 GB of the total 64 GB of installed RAM on that host. Using Fast RTPS, it was not possible to start more than 100 nodes, as the CPU was already working at full capacity, and the DDS nodes were unable to discover any topics after some time. To still conduct the tests for DDS, the OpenDDS implementation was used as an alternative, which supports more efficient shared memory. The high load is the result of the discovery process in ROS and DDS: as explained in Section 4.4, these two protocols produce a higher number of packages with more data compared to OPC UA and MQTT. Therefore, the efficiency of the connection setup and teardown implementation has a high impact on the CPU usage, especially if there are multiple instances started on the same host.

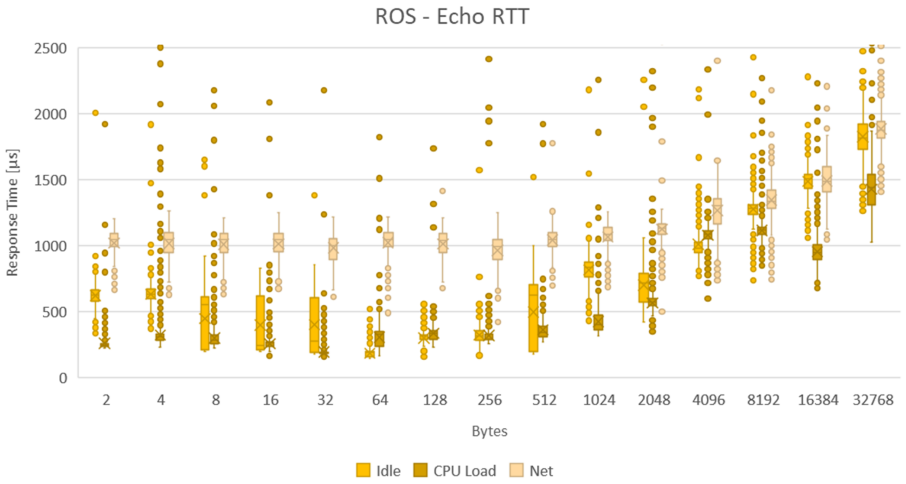
This test shows that the open62541 OPC UA Client-Server implementation is still the fastest protocol and is even faster than the DDS shared memory implementation. The reason for the huge performance gap to MQTT and ROS is the direct TCP connection between nodes in OPC UA, whereas MQTT uses stateless UDP connections, and ROS is slowed down by the high CPU load it produces on the host. In [Maruyama et al., 2016], the authors also state that for small data, shared memory does not improve the latency compared to local loopback. Therefore, shared memory communication does not improve the performance on small data packages, but it may have a significant performance improvement if there is a significant amount of network traffic in parallel.

4.6 Summary & Discussion

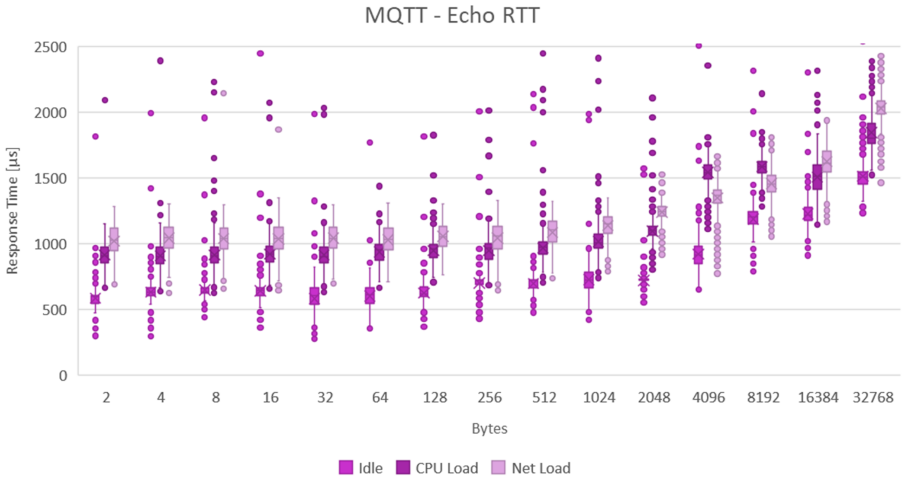
This chapter gives an overview of the features and performance of some well known middlewares in the field of industrial automation and Internet of Things: OPC UA, ROS, DDS, and MQTT. The feature comparison shows that every middleware has its strengths and weaknesses. Therefore, there does not exist a single truth on the best middleware for everything.
OPC UA has its strength in the semantic modeling of information. ROS is mainly used for controlling robots and their modules for research purposes and provides many pre-implemented feature packages. DDS has an extensive set of Quality of Service settings, whereas MQTT mainly focuses on a lightweight Publish-Subscribe protocol.

The performance comparison of the used open-source protocol implementations shows that open62541 for OPC UA and eProsima Fast RTPS for DDS deliver high performance, whereas the MQTT and ROS implementations show a significant slowdown in the RTT of packages sent to the server. It can also be seen that open62541 is based on a very performant implementation, confirming the results published in [Cenedese et al., 2019] where different open-source OPC UA stacks (open62541, freeOpcUa (C++), freeOpcUa (Python)) are compared and open62541 clearly outperforms other implementations.

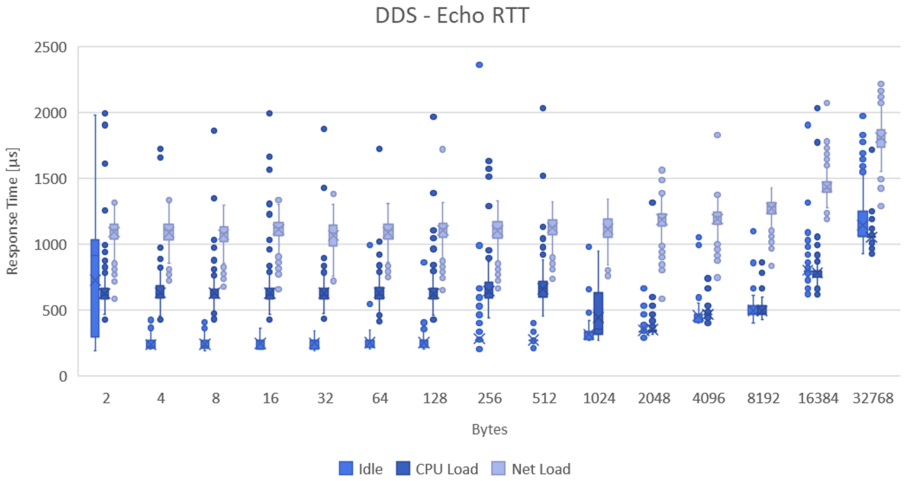
This chapter also answers the initially stated question “Which communication protocol is the best suitable one for the Plug-and-Produce use-case?”. OPC UA, especially its open-source implementation open62541, is a very performant protocol implementation and therefore an ideal candidate for the industrial automation domain. Typically, industrial applications require low-latency communication and securely encrypted data exchange to avoid damages. In addition to its performance, OPC UA includes a rich semantic description model which is essential for Plug & Produce, as the following chapters will show in more detail.



(a) Echo RTT for different data message size using ROS on Idle, CPU Load, and Network Load.

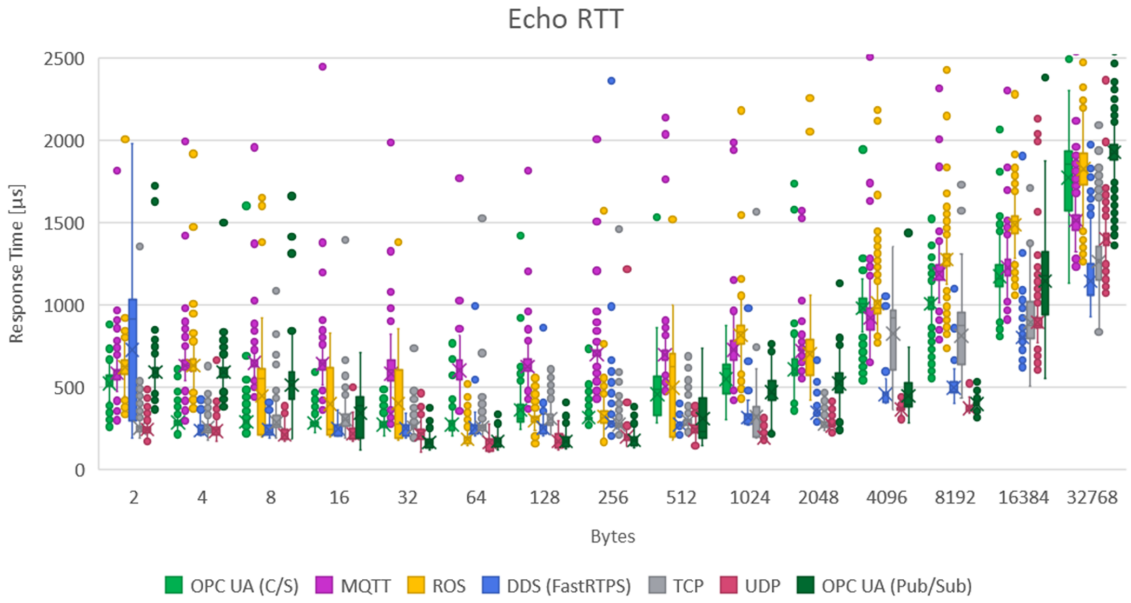


(b) Echo RTT for different data message size using MQTT on Idle, CPU Load, and Network Load.

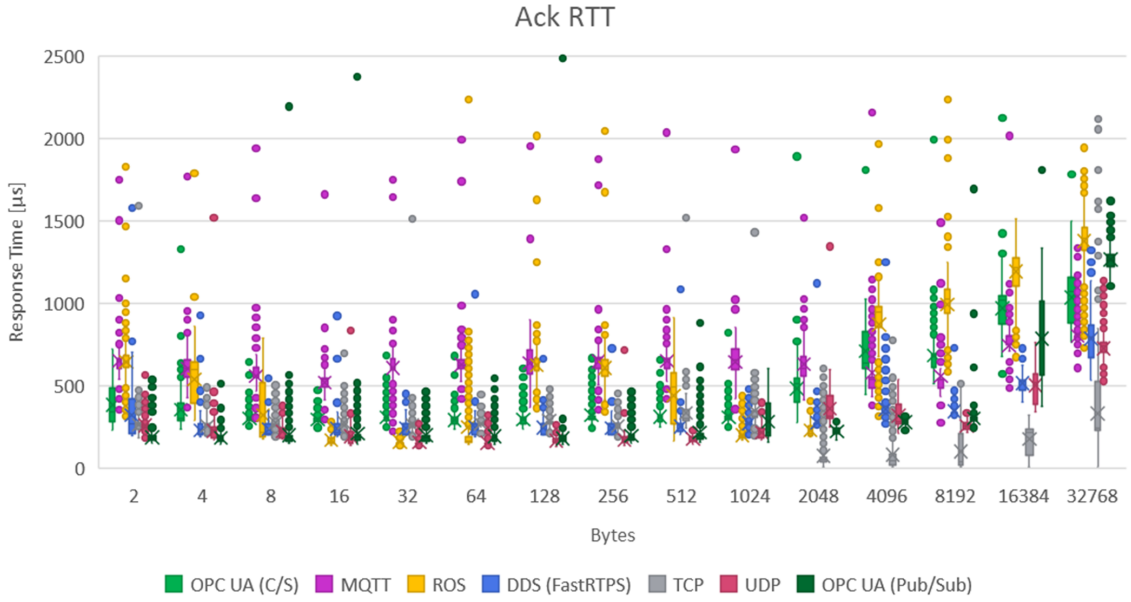


(c) Echo RTT for different data message size using DDS on Idle, CPU Load, and Network Load.

Figure 4.5: Plots showing the echo RTT measurement results for ROS, MQTT, and DDS for an excerpt of the collected data.

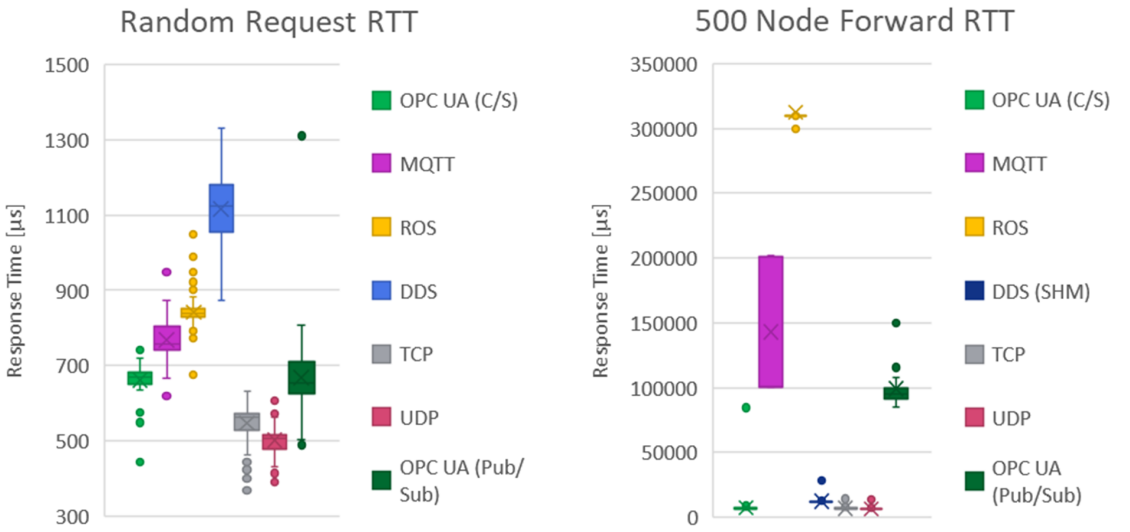


(a) RTT for sending and receiving data messages using different protocols.



(b) RTT for sending data with a simple ACK message using different protocols.

Figure 4.6: Plots showing the direct RTT comparison between the protocols for echo and ACK messages.



(a) Resting RTT for sending a payload of 4 bytes waiting randomly between 0 and 3 seconds.

(b) RTT for sending 10 packages (10240 byte payload) through 500 nodes on another host.

Figure 4.7: Plots showing the resting RTT and the RTT for high data routing on the remote host.

5 Automatic Component Discovery

Partial results of the presented work in this chapter are published in my peer-reviewed publication [Profanter et al., 2018]. I am the main author of this publication, and my main contributions are in further detail described in this chapter. Some figures created and partial text written by me in this publication are directly included in this chapter.

All components within a generic Plug & Produce system should require as little preconfiguration as possible resulting in reduced integration time. Due to this constraint, components need to implement a concept which allows to automatically detect other participants in the system. This is called automatic device discovery.

Figure 5.1 is already known from Section 1.5 and highlights the focus of this chapter: The discovery of components in the system.

In this chapter, I propose a novel hierarchical architecture for a multi-level Plug & Produce system. This architecture is used to automatically discover other I4.0 components in the network and allows optional hierarchical grouping of devices. First, I present different requirements which must be fulfilled by components for such a system. In the previous chapter, I show the comparison of different middlewares, and why OPC UA outperforms feature-wise and performance-wise in this comparison. This chapter is based on another important feature of OPC UA: the discovery service set. I present a deeper insight into the OPC UA discovery service set, which is a basis of the presented novel architecture, followed by a short overview of different open-source OPC UA implementations, and their implementation status of the discovery service set. On top of the features of the service set, I define the hierarchical Plug & Produce architecture and evaluate it. The focus of this evaluation lies on the applicability of the hierarchical architecture for easy integration of new devices. A more detailed evaluation is given in the follow-up chapters, together with the additional concepts described later.

My developed source code, as part of this chapter, is available as open-source on GitHub in various projects and described in more detail in this chapter.

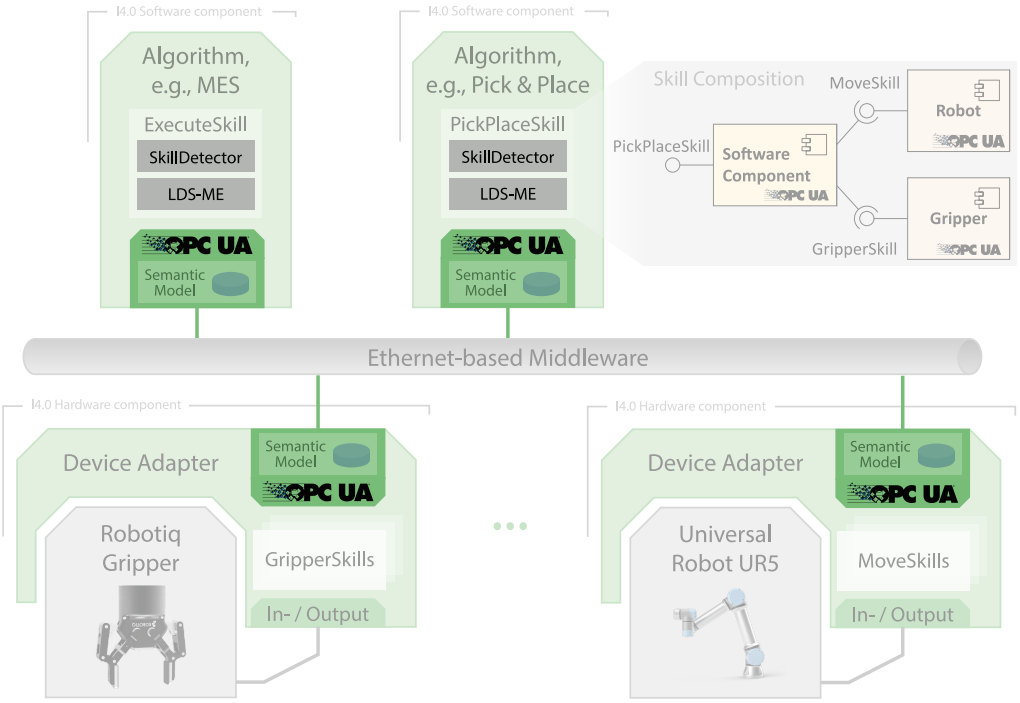


Figure 5.1: Exemplified system setup with different Industry 4.0 components. Chapter 5 focuses on the automatic component discovery based on OPC UA, and the skill detector concept.

5.1 Plug & Produce Component Requirements

The lifecycle of intelligent I4.0 components (e.g., sensors, devices, workstations, software) can be separated into four different phases (see Figure 5.2): *Discovery*, *Configuration*, *Production*, and *Reconfiguration*. To ensure easy integration and interaction of other components with the component itself, every phase must meet different requirements which I present in this section, separated by the mentioned phases.

5.1.1 Discovery Phase

As described in Chapter 2 and defined in [ZVEI, 2015b], Ethernet-based communication is the highly recommended communication method for I4.0 components. It is typically based on IP. Every participant in an IP network generally requires an IP address and a corresponding port number. There exist various automatic and non-automatic methods to assign IP addresses and TCP or UDP ports to devices. The most straight-forward way is to pre-configure every device or component (i.e., machines, devices, sensors) with a specific address, adapted to the specific factory floor and its corresponding network

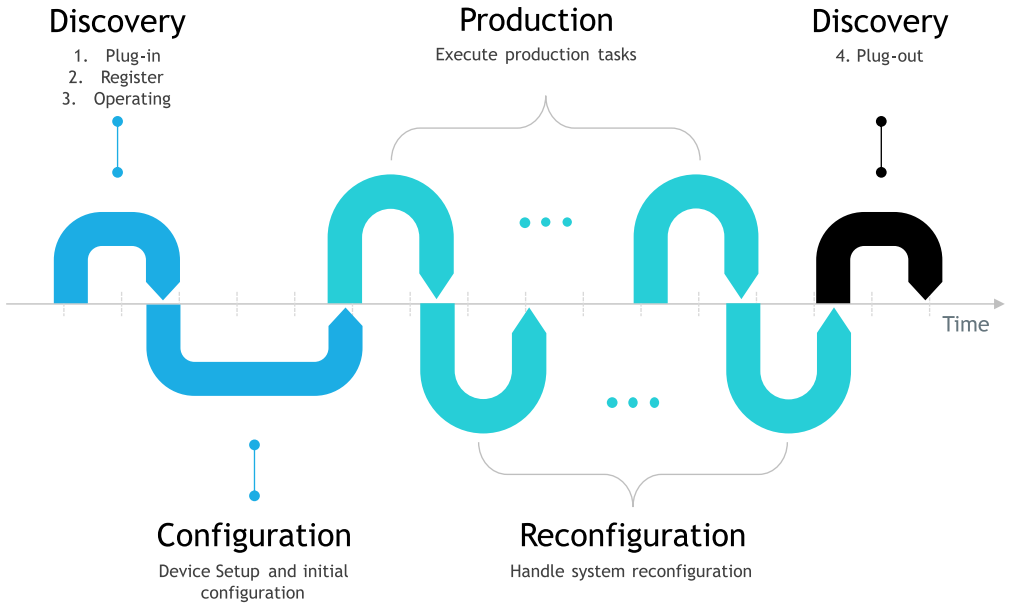


Figure 5.2: Lifecycle phases of an intelligent I4.0 Component (Discovery, Configuration, Production, Reconfiguration) with substages in Discovery

setup. Using this approach, new components cannot be easily integrated into the factory, as not only new components need to be configured, but also the configuration of existing components needs to be adjusted to accommodate the change. This is an error prone process.

To achieve a more Plug & Produce-friendly setup without any factory-specific preconfiguration, the components need to be able to discover other components automatically. This discovery process can be further subdivided into four stages: *Plug-in*, *Register*, *Operating*, and *Plug-out* (see Figure 5.2). One of the few middlewares which can handle these stages is OPC UA. More technical details on the discovery process of OPC UA can be found in the next Section 5.2.

The **Plug-in** stage is the first step, in which the component needs to set up its communication stack, announce its presence to other components, and get a list of all available components in the network. In Ethernet-based communication this is typically achieved via Dynamic Host Configuration Protocol (DHCP), which is used for automatic assignment of IP addresses to devices. The next step for the component is to announce itself by sending out a multicast message to all network devices in the current subnet, e.g., using Zero-configuration networking (zeroconf). Other devices should then respond with a corresponding message containing information on how the network device can be contacted, i.e., its IP address and TCP or UDP port.

After the list of all the network devices is available, the I4.0 component must select the correct device to **Register** itself. By registering itself, the component tells other devices that it is available. The register message also needs to include additional identification data of the component itself. The major issue in this stage is to understand the topological structure of components, so that the component can register with the correct supervising component. Typical industrial setups are hierarchically grouped devices, i.e., the work cells in a production line are controlled by one central entity, every workstation can consist of multiple subsequent devices which are again hierarchically controlled [Brandenbourger and Durand, 2018]. The main goal of this abstraction is to reduce the complexity with every level. The task of the supervising components is to keep track of all enclosed available devices and to connect different I4.0 components on the same hierarchy level with each other. To be able to recognize the type of the component, each Plug & Produce device should have embedded information about its capabilities, skills, data input and output, and the different key performance indicators (KPI) it provides.

After registering, the component is in the **Operating** stage, which is the normal execution mode where the component is periodically checking the current network status and re-registering with the register server to indicate that it is still alive. This ensures that the list of available devices is consistent, e.g., if the network connection is broken, or the device itself is not available anymore it must be removed from the list.

During the **Plug-out** stage the I4.0 component unregisters itself from the supervising component. Unregistering may occur gracefully if the device is shut down normally, or abruptly if the network connection is broken. In the latter case the component itself has no way to notify the Local Discovery Server (LDS) about its plugged-out state. Therefore, the periodic re-register in the operating phase is used as a mean to detect the broken connection. In the worst case, a client trying to connect to an offline component will only then detect that the device is offline, and therefore must wait for the connection timeout. If the device is shut down normally, it first issues an unregister service call and with this immediately notifies other servers that it is shutting down.

5.1.2 Configuration Phase

After the communication channel for the I4.0 component is set up, and it is connected to the corresponding control entities, the component needs to be configured to perform its designated task.

Automatic configuration is still one of the major challenges, and a highly researched topic, as shown in the "Plattform Industrie 4.0"[1] project. Semantic reasoning must be performed to get from complex product specifications and system requirements to the execution of production steps. Additionally, the question must be answered, how a single configuration for a product can be split and deployed to different components or machines to allow hierarchical execution of the necessary production steps. Academic experiments such as MGSyn [Cheng et al., 2012] or F++ [Keddis et al., 2015] have applied game-based reasoning or search-based techniques to create distributed recipes or orchestration plans, based on a predefined skill library of every machine and high-level specification.

In Chapter 6, I present a generic semantic device and component description which allows initial automatic configuration and parametrization of execution functionalities based on the proposed generic skill model. Combined with semantic reasoning a Plug & Play system can be achieved.

5.1.3 Production & Reconfiguration Phase

After the component is configured and all required information is gathered, an I4.0 component needs to provide the services to start the execution and to achieve its task. Each such service, i.e., a basic action of the device, can be defined as a skill. Depending on the complexity of the component, multiple skills or a combination of base skills may be presented to the upper control component. A more detailed description on skill composition is given in Chapter 7.

It is also necessary for a component, to handle different events in the production phase, e.g., if a new job is started, while the old job is still running. Additionally, the component must be able to handle errors which may occur on the hardware or software side during execution of the skill, on the upper level, or if another collaborating component fails to execute its task.

If there is a production running at the moment, while a new component is plugged into the system, all involved components need to be able to handle reconfiguration during run-time correspondingly, in the simplest case by ignoring new devices until a new production cycle is started. The same holds true if a required component is unplugged. Therefore, an intelligent I4.0 component must handle switching jobs when a new production task arrives, including the case where the hardware is altered, or a collaborating device changes its status. Additionally, the communication channel may change (online/offline) and thus needs to be updated accordingly. Moreover, in the reconfigu-

[1] http://www.plattform-i40.de/

ration phase the I4.0 component can receive new parameters for executing its existing skills, e.g., for optimizing the execution speed or power consumption, and new product variants can be also introduced to the system.

5.2 OPC UA Discovery Services

As shown in Chapter 4, OPC UA is a performant and feature-rich protocol. A particularly important feature for the Plug & Produce concept are the OPC UA's Discovery Services specified in Part 12 of the official OPC UA Specification [OPC Foundation, 2015]. Part 12 and partially Part 4 of the specification describe how OPC UA applications can be discovered on a computer or network. It includes the general discovery process, LDS and Global Discovery Server (GDS) concepts, and certificate management within GDS.

In this section, I describe the discovery process and LDS functionality, and summarize the concept of application discovery in OPC UA. This summary provides a basic understanding on what OPC UA Discovery can do, and which additional functionalities are required to fulfill the previously mentioned requirements to implement a real Plug & Produce system. A more detailed technical explanation of the described concepts can be found in the OPC UA specification Part 12.

The discovery process defines how OPC UA Clients can find OPC UA Servers and how such a server can be reached. Therefore, the discovery mechanisms can be separated into two parts: the client discovery and the server announcement.

A client can use different methodologies to find a server. The most basic concept is a list of predefined hard-coded discovery URLs of server endpoints. This is not optimal for adaptable I4.0 components, which are used in different environments since this list needs to be updated manually. A more flexible approach is to contact a locally running server on the same host with a well-known port number. In OPC UA terminology this server is called LDS. A LDS offers specific services like *FindServers* or *FindServersOnNetwork* that can be called to get a list of servers which were discovered by this instance in the same subnet. The LDS is monitoring server announcements on the multicast subnet or is querying the GDS for known servers. A client can even get a list of servers by just monitoring multicast announcements itself, which makes the system even more flexible.

On the server side a corresponding discovery mechanism needs to be implemented:

The *Local Discovery Server (LDS)* is an OPC UA server instance running on a host. It typically keeps track of all OPC UA servers on the same host since they must explicitly register with the LDS, which is by default listening on port 4840. In more complex setups, other OPC UA servers outside of the current host can also explicitly register themselves with a specific LDS.

The Local Discovery Server with Multicast Extension (LDS-ME) is an extension of the LDS. It additionally keeps track of all the servers that announce themselves on the local multicast subnet. Servers on the same host can register themselves directly, whereas other LDS-ME servers are detected by multicast announcements. Server instances on a host can register themselves with the server on localhost port 4840, which then announces the registered server via multicast to other hosts. This avoids the necessity to define specific IP addresses beforehand.

The *Global Discovery Server* (GDS) is another type of discovery server and can be used for discovery among multiple subnets, and in subnets where host names cannot be discovered directly (for the multicast announcements, host names must be discoverable). A GDS may also implement certificate management services for distribution and central trust management of certificates for other OPC UA applications. Certificates are used in OPC UA for encryption and authentication to ensure clients and servers are talking to the expected entity. In typical setups the IP address is statically defined, or DNS is used to determine the IP address of the GDS.

Since my goal is to avoid pre-configuration of components, my focus lies on the case where a client uses the LDS-ME server on the same host to find other servers and other Industry 4.0 components correspondingly so that no component needs to be configured with specific IP addresses beforehand. This is also the basic concept which I use in all the evaluation experiments in later chapters.

Figure 5.3 shows the sequence of the discovery process: In the first step the Server on machine *A* registers with its LDS-ME. When machine *B* is plugged in, its LDS-ME issues a multicast probe on which the LDS-ME on machine *A* responds with a multicast announcement. The LDS-ME on machine *B* now knows all the available servers on the network. Finally, the client on machine *B* queries these servers and selects the desired remote server on machine *A*. Using this information, the connection between both machines can be established without any pre-configuration. The only necessary piece of information is the port on which the local LDS-ME is running, which is normally port 4840, as defined in the OPC specification.

The OPC UA specification already provides basic concepts for device discovery, i.e., how the multicast messages are handled and how the registration on the LDS-ME is done. OPC UA does not include the concept of hierarchical LDS-ME servers, which is one of

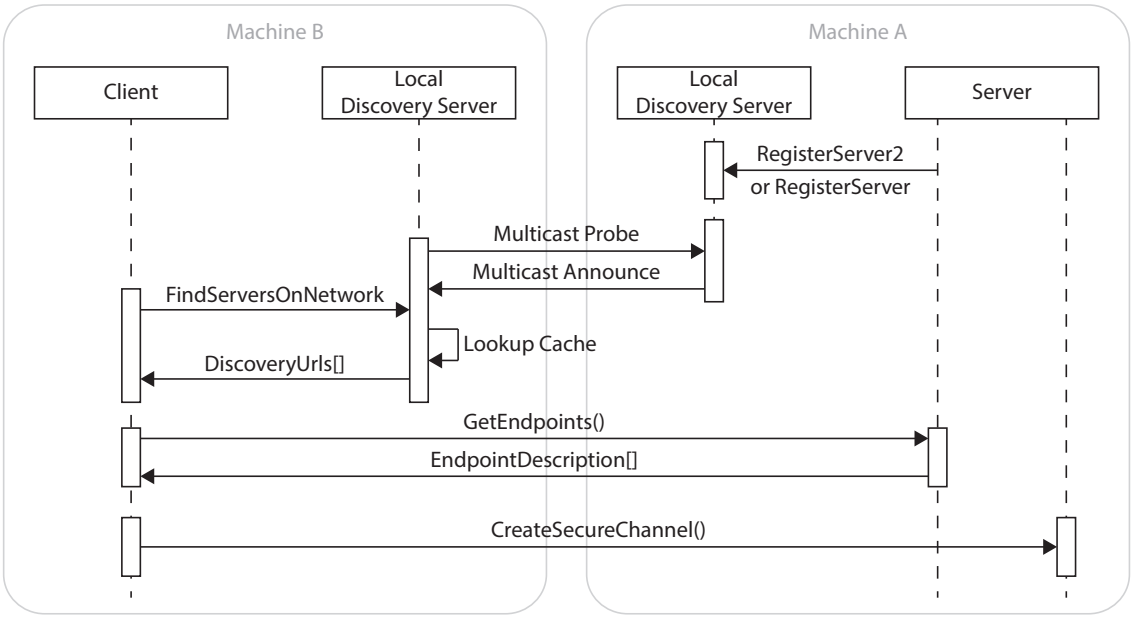


Figure 5.3: By using the Multicast Subnet discovery process, the secondly attached machine *B* queries a predefined or discovered server for all known servers on the network and can then contact the desired endpoint on machine *A*.

the concepts I describe in Section 5.4. Also, the semantic identification as described in Section 5.1.2 is not part of the discovery services and is described as part of Chapter 6.

The next section compares different open-source OPC UA stacks and shows the required additional implementation effort from my side to get the discovery service set up and running in C/C++ and Java.

5.3 Comparison of OPC UA Implementations

To implement Plug & Produce in OPC UA a software stack must be selected which supports the discovery service set. More specifically, it must support the Local Discovery Server with Multicast Extension (LDS-ME). There exist many commercial OPC UA stacks in different programming languages but unfortunately, as of the year 2017 when I investigated this topic, only very few already provided an implementation for LDS-ME. There were even less open-source implementations which support LDS including multicast discovery.

Since my research topic requires the extension of the OPC UA stack, and my goal is to make my research results open to the public world and therefore rely on open-source

software for easy adaption, I was looking for open-source OPC UA implementations. An additional requirement is that the resulting software needs to support microcontrollers to develop device adapters as described in the next chapter. Most industrial-grade microcontroller frameworks only support compiling C/C++ code, while some higher-level control applications rely on Java. Therefore, the focus of the following comparison lies on open-source C/C++ and Java implementations of OPC UA.

There also exist other open-source implementations for C#[2], Python[3], and JavaScript[4].

5.3.1 Comparison of C/C++ OPC UA Stacks

In this section, I compare different open-source OPC UA stacks. Commercial OPC UA stacks are not part of this comparison since I need to extend the OPC UA implementation in the progress of my thesis, which is not possible for closed-source implementations. All the following OPC UA stacks can be compiled under Linux and Windows, some of them also support OS X or microcontroller targets.

open62541 (MPL-2.0, like LGPL, but includes static linking) [Palm et al., 2014]. It provides an API for both, server and clients, and supports nearly all features of the different discovery sets, except (initially) the Discovery Service Set. Now in the year 2020, more than 4 years after initially writing this chapter, Publish-Subscribe and many more features have been added. The implementation is certified with the OPC Foundation Compliance Test Tool (CTT). The information model can be automatically generated out of XML files. This project is actively developed, and new features are constantly being added. It is now one of the most supported and starred open-source stacks. Aside of Linux and Windows many platforms are supported, e.g., OS X, QNX, Android and embedded systems.
https://github.com/open62541/open62541

OpenOpcUA (CECILL-C, like GPL with no fork option). It provides an API for server and client development and allows to dynamically load UA information models from XML files. It is also tested with the CTT. Its main disadvantage is that a one-time fee is requested to access the codebase. The last release was in September 2018.
http://www.openopcua.org

ASNeG OpcUaStack (Apache License, 2.0). It provides an API for servers and clients with only basic functionality, i.e., reading, writing, and monitoring OPC UA Vari-

[2]https://github.com/OPCFoundation/UA-.NETStandard
[3]https://github.com/FreeOpcUa/python-opcua
[4]https://github.com/node-opcua/node-opcua

ables.
http://asneg.de

FreeOpcUa (LGPL). It is a server and client library with support for most of the basic OPC UA service sets, except the discovery service set. Since 2015 contributions to this project rapidly decreased and only a few features have been added. In the year 2019 only 4 commits were made on GitHub.
https://github.com/FreeOpcUa/freeopcua

UAF Unified Architecture Framework (LGPL). It only implements the OPC UA client side and does not support OPC UA servers. Additionally, it is based on the commercial C++ OPC UA Software Developers Kit from Unified Automation, which is required to develop applications.
https://github.com/uaf/uaf

OPC Foundation AnsiC provides an official reference implementation under a dual-license, proprietary for OPC Foundation Members and GPL for everybody else. In March 2017, the LDS-ME server implementation was released as Beta for Windows only. Beginning of 2019 this repository was declared end-of-life since the OPC Foundation decided to focus on their .NET implementation.
https://github.com/OPCFoundation/UA-AnsiC-Legacy

When starting my thesis in 2016, none of the above stacks fully supported the Discovery Service Set, especially the LDS-ME was missing. The OPC Stack from the OPC Foundation included, as of May 2017, a beta release of a stand-alone LDS-ME, but due to its GPL license, and the requirement to disclose the application code built upon the LDS-ME code, it did not meet the necessary requirements. In addition, this official AnsiC reference implementation is since beginning of 2019 declared as discontinued.

Due to its high popularity, open license, and many of the required features already implemented, I decided to use the open62541 stack as the basis for my software architecture. Now, mid of 2020, this decision proved to be the right one: it is still very actively developed with a lot of commercial companies backing it.

One of the most important features, the LDS-ME Server, was in 2017 not yet included in the stack as it was the case for all other open-source stacks. My steps for implementing LDS-ME is further described in Section 5.5.1.

5.3.2 Comparison of Java OPC UA Stacks

Aside of the most used commercial stacks from Unified Automation, Prosys, and Asco-Lab there are only a few open-source implementations of OPC UA in pure Java. It is possible to encapsulate a C stack from the previous subsection using Java Native Interface (JNI), which would result in a less portable but more performant data serialization.

Here I compare the only two well-known open-source stacks, which use pure Java and implement most of the OPC UA features.

OPC Foundation Java is the official implementation by the OPC Foundation and released under a dual-license, proprietary for OPC Foundation Members and GPL for everybody else. It provides the basic tools to implement OPC UA servers and clients, but only included example code for the Nano profile which does not support the discovery service set. Like the AnsiC implementation of the OPC Foundation, the development of the Java stack has been discontinued beginning of 2019 with a reference to the official .NET implementation.
https://github.com/OPCFoundation/UA-Java-Legacy

Eclipse Milo is a project under the Eclipse Foundation and therefore licensed under EPL-1.0. It includes a fully functional stack, client, and server SDK, however it is missing certain functionality, especially LDS-ME.
https://github.com/eclipse/milo

Since the official OPC Foundation reference implementation for Java has been discontinued, the only major open-source Java implementation is the Eclipse Milo project. Therefore, I decided to use the Eclipse Milo for my implementation and testing of Plug & Produce for operating system independent Industry 4.0 components. Aside of the presented open-source projects, there are also a few commercial stacks available, which support Java.

5.4 Hierarchical Component Discovery

My proposed architecture for automatic device discovery using OPC UA with LDS-ME consists of an intelligent Manufacturing Service Bus (MSB) which is responsible for detecting other I4.0 components on the network, and to configure the component when it is plugged in. Such a component may be an intelligent workstation which is embodied by an administration shell to provide the needed services for discovery. This workstation also contains an OPC UA Server responsible for discovering newly plugged-in devices on the workstation itself and providing this information to the MSB. The general architecture is shown in Figure 5.4.

The MSB acts as the centralized communication mean, which makes it possible to move from the automation pyramid to a more vertically and horizontally integrated automation platform by using the same middleware on all layers, as shown in Figure 3.1. Its responsibility is also to tell a requesting workstation which other workstations are currently available so that the two workstations can directly communicate with each other. To detect other components on the network, the MSB must implement LDS-ME.

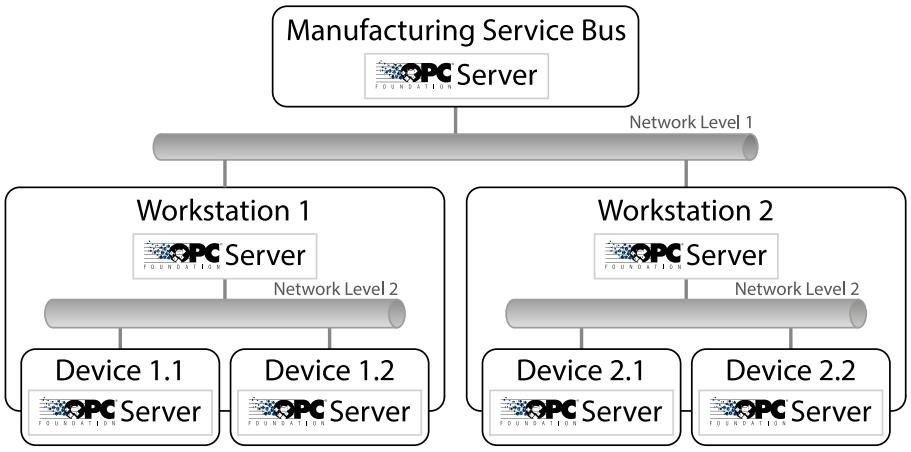


Figure 5.4: Proposed architecture for hierarchical Plug & Produce. The Manufacturing Service Bus (MSB) can discover workstations. Each workstation provides discovery functionality for devices within the workstation.

A workstation is a super component which encompasses other components in logical terms to act as a unit, and to abstract the underlying components for a higher level. As shown in Figure 2.2 and described in RAMI 4.0 [ZVEI, 2015b], nestability of I4.0 components requires such components to have more than one communication interface on different abstraction levels and a component management for subcomponents. The interface on the upper level is connected to the MSB and is used for registering the workstation, and to receive corresponding configuration data from the MSB. The component management is implemented as an OPC UA Server which can receive control commands from the MSB, and if necessary, forward these commands to the subcomponents, i.e., devices. To be detectable by devices which belong to the workstation, the workstation also needs to implement its own LDS-ME server that listens on the lower level interface for new device announcements.

A device can be a single actor, like a motor, or a sensor which delivers specific measurement data for the higher level or other devices on the same workstation. In a more complex system structure, a device could be a workstation with subcomponents. For the sake of simplicity, I focus in this chapter on the case where a device is an actor or a sensor without subcomponents. Still, this more complex use-case is supported by the presented architecture as shown in later chapters.

In the following paragraphs, I present the hierarchical discovery process which is part of my novel hierarchical Plug & Produce architecture. The discovery process is responsible for automatic device detection inside the network, which avoids pre-configured IP addresses.

An exemplified discovery process with multiple levels of hierarchy is shown in Figure 5.5 and explained in more detail in this section. This setup consists of the MSB and two workstations with two devices each resulting in two levels of hierarchy. In general, an OPC UA Server can support multiple service sets at the same time, therefore it can be a standard OPC UA Server which provides methods and data to clients and at the same time offer the LDS-ME functionality. Additionally, an OPC UA Server cannot call a method on another OPC UA Server directly and thus needs to use an OPC UA Client to call the corresponding configuration methods on the lower-level server. Therefore, the green boxes in Figure 5.5 represent an OPC UA server with additional LDS-ME functionality and built-in clients.

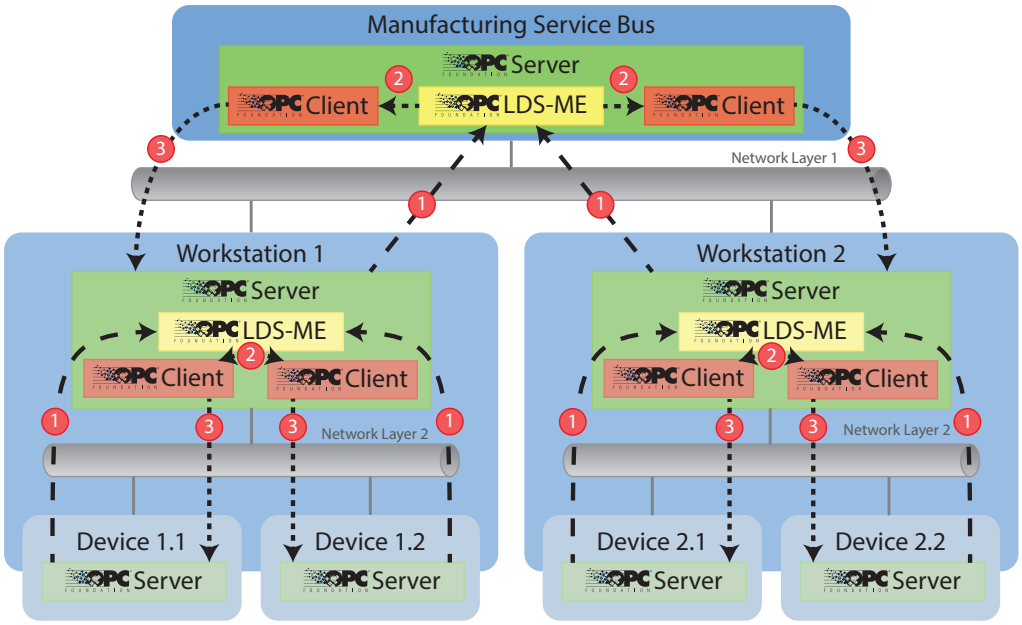


Figure 5.5: Discovery process with multiple levels of hierarchy. The Manufacturing Service Bus controls Workstations which in return control Devices.
① The component's OPC UA server detects the LDS-ME server through multicast and registers itself with the LDS.
② The LDS-ME Server creates a new client to communicate with the OPC UA Server.
③ The Client calls the configuration method on the Server and controls its actions.

The detailed process for automatic device discovery as shown in Figure 5.5 is as follows:

1. When a workstation, device, or component is plugged in, it gets assigned an IP address and the corresponding subnet using DHCP. After network initialization, the OPC UA server on the plugged-in device issues a multicast probe (see Figure 5.3). All LDS-ME servers within the same subnet respond with a multicast announcement, including information how the LDS-ME server can be contacted. With this information the newly plugged in OPC UA server can register itself with

the LDS-ME server. This step completes the Plug-In and Register stages described in Section 5.1.
2. The new server is stored in a list and an OPC UA client is created to be able to connect to the newly registered server. An OPC UA server cannot directly communicate with another OPC UA server.
3. The newly instantiated client then calls a predefined configuration method on the server with configuration parameters, e.g., how often status updates should be sent (Configuration phase). This client instance is also used in the Operating phase for controlling the workstation or device, respectively.
4. If a device or workstation is gracefully shut down, it must unregister itself from the upper LDS-ME server. If it is disconnected by a network or software fault, the controlling server either detects the downtime through its controlling client or through the LDS-ME server which checks if the underlying component periodically re-registers itself. An additional heartbeat implementation can be used to detect component downtime even faster.

If the network link is down or the LDS-ME server is not running when a new component is plugged in, the multicast probe is sent after a short retry interval to make sure the registering succeeds as soon as possible.

5.5 Implementation & Evaluation

This section shows the necessary parts of code I had to implement in both open-source OPC UA stacks to evaluate my proposed architecture. The implementation of the hierarchical discovery architecture is then evaluated.

5.5.1 Implementing LDS-ME

Since neither open62541, nor Eclipse Milo did support the full LDS-ME, I first had to implement this feature set according to the OPC UA Specification Part 12 [OPC Foundation, 2015], to be able to create configuration-less Plug & Produce Industry 4.0 components. The features of the discovery service set are described in Section 5.2.

In the open62541 C/C++ stack, the services for LDS without multicast were already implemented. This includes *RegisterServer* to allow other server instances to register themselves with the LDS, *FindServers* to allow clients to get a list of registered servers, and *GetEndpoints* which returns the list of available connection endpoints of the LDS.

For my specific architecture proposal, the complete LDS-ME functionality is required, especially the multicast announcement.

The first step to get LDS-ME into open62541 was to evaluate different open-source Multicast Domain Name System (mDNS) implementations, which can be used within the open62541 project in consensus with the MPL-2.0 license. The corresponding discussion with the core developers of open62541 can be found on GitHub[5]. For easy integration and support on embedded devices the mDNS library must be self-contained without any external dependencies. The only fully functional library which meets this requirement and is less restrictive than the MPL-2.0 license is the *mdnsd* library, which I adapted to be compilable under Linux, Windows and OS X and fixed various bugs. The improved library source code is available on GitHub[6]. Based on this library, I implemented the multicast mechanism for automatic detection of other running instances within the same subnet as shown in Figure 5.3. This mainly consisted of integrating the *mdnsd* library into the core and handle multicast messages. Finally, I implemented the *RegisterServer2* service, which allows other instances to register with additional multicast information, and *FindServersOnNetwork* which returns not only explicitly registered OPC UA servers, but also the ones detected through multicast messages. These changes were compiled as a pull request with more than 3000 lines of code modifications and contributed to the base repository of open62451[7], where it is now integrated into the master branch and latest releases.

With this significant contribution I was nominated as a core developer of the open62541 stack where I contributed many new features and bug fixes during my free time over the last few years.

For the Eclipse Milo project, the same amount of work had to be performed: After I extended the basic methods *RegisterServer*, *FindServers*, and *GetEndpoints*, the *jmDNS* library was used to add mDNS support to Java. This enables Milo to detect other multicast enabled OPC UA instances on the network and to keep a list of known OPC UA servers. Additionally, the *RegisterServer2* and *FindServersOnNetwork* were implemented, and then again compiled into a pull request with around 3000 lines of code modifications, to be submitted to the Eclipse Milo project on GitHub[8]. Unfortunately, this pull request is after four years still not completely merged into the master branch and is still pending.

[5]https://github.com/open62541/open62541/issues/701
[6]https://github.com/pro/mdnsd
[7]https://github.com/open62541/open62541/pull/732
[8]https://github.com/eclipse/milo/pull/89

5.5.2 Hierarchical Discovery Evaluation

This section presents a preliminary evaluation of the basic discovery features. A more detailed evaluation of the whole discovery process is included in later chapters of this thesis.

OPC UA defines how data should be serialized on the wire, therefore different OPC UA stacks are able to directly communicate with each other. This is also the case for the discovery services: LDS-ME servers from different vendors should be able to discover each other and to exchange data. I evaluated the compatibility of my presented implementations in open62541 and Milo, including the applicability of the presented hierarchical architecture, by creating dummy workstations and devices which need to register with the corresponding LDS-ME server. Different combinations of open62541 and Milo are used to show their compatibility. Additionally, both implementations are tested against the official OPC Foundation reference implementation of the LDS-ME server written in C[9].

For this setup I am using a standard Desktop PC with Ubuntu Linux where the LDS-ME server is directly started. Additionally, two Virtual Machines are set up, which simulate the two workstations from Figure 5.5. Each workstation contains a simulated device by directly starting an application within the virtual machine. The plugging in of a component is simulated by starting the application. In real-world setups a device either just needs to establish a network connection, or it still needs to boot up first. Since this does not influence the functionality of a LDS-ME, these different cases are neglected.

The device will then query the LDS-ME server on the workstation for other known OPC UA Servers. This query should return the OPC UA Server from the second device on the same workstation, so that both devices on one workstation are connected with each other. In my test the device will then just call a method to trigger the time measurement for the performance evaluation.

Executing these steps with different combinations of my LDS-ME implementations and the official LDS from the OPC Foundation have shown that all implementations are able to discover each other. In open62541 the discovery process is significantly faster, as it takes less than a 500 milliseconds for the workstation and device to find the LDS-ME servers, register with them and then query for known OPC UA servers, to finally call the method on the other device. In Eclipse Milo the multicast probe is slower as it takes up to 7 seconds until the device or workstation detects its counterparts LDS-ME. This is because the *jmDNS* implementation is first checking if there is already a DNS-SD service announcement with the same name, using a fixed timeout value of 6 seconds, which is hardcoded inside the *jmDNS* implementation. Only afterwards the mDNS probe is sent

[9] https://github.com/OPCFoundation/UA-LDS

out and the response from the LDS-ME server is processed. In open62541 such a name conflict is handled by immediately announcing itself, and if at the same time another server with the same name exists, the mDNS implementation within open62541 detects it and can change its name to a more unique one to retry. Since the official LDS-ME implementation of the OPC Foundation only provides an LDS-ME server without the possibility to easily add user code, its performance was not evaluated.

If different network segments are used, as shown in Figure 5.5, a device on one workstation cannot simply connect to a device on another workstation. Either the network segments must be connected and corresponding routing tables must be set up, or servers on a higher level must act as proxies, e.g., a device must call its parent workstation server, which calls the server on another workstation, which then can call a method on its underlying device. This introduces additional delays, and it is desirable to keep communication of devices isolated on the same workstation. If additional data is required from other workstations or subcomponents, this data should be delivered through the MSB or via data aggregation of higher-level OPC UA servers.

If there are multiple LDS-ME servers running within the same subnet, additional intelligence and pre-configuration must be added to devices. If a device receives multiple LDS-ME announcement messages, it must select the correct LDS-ME to register itself. To solve this issue, the device can be configured to only connect to LDS-ME servers with a specific ID, or the main LDS-ME server has some specific nodes within its information model, which a device can query and then decide if it is the LDS-ME it is looking for.

In [Madiwalar et al., 2019] it is shown how the combination of Software-Defined Networking (SDN) in combination with intelligent switches can be used to cope with this problem. Before the switch forwards the multicast message to other components in the network, it can use the server's semantic description to infer the correct parent component. The SDN switch then selectively forwards the mDNS message directly to that component. With this approach the implementation of workstations and their devices does not need to be changed, as the decision logic is inside the network and can dynamically be adapted.

5.6 Summary & Discussion

In this chapter, I first list requirements which must be fulfilled by components to achieve a real Plug & Produce architecture. This is followed by a more detailed explanation of the OPC UA discovery service set, and a comparison of different open-source OPC UA implementations. The proposed hierarchical discovery architecture is the main contribution of this chapter, including the necessary implementation efforts to get the dis-

covery functionalities into the open-source stacks. The chapter is concluded by a short evaluation of the implemented functionalities. A more detailed evaluation of the discovery functionality is given in the next chapters, especially in Chapter 8.

The main take-away message of this chapter is, that it is possible to implement automatic component discovery for the Plug & Produce concept on the network side using OPC UA Local Discovery Servers with Multicast Extension (LDS-ME). The discovery service set allows easy integration of new devices into the network without any network-specific pre-configuration. Using multiple hierarchies of LDS-ME, it is possible to create modular workstations which abstract underlying devices to the middleware. Still, a common semantic interface between these components is required.

As compared in this chapter, there are many open-source implementations for OPC UA, whereas open62541 for C/C++ and Eclipse Milo for Java provide a manufacturer-friendly license with most of the OPC UA features already implemented.

The next chapter is describing in more detail the device adapter concept which adds the semantic abstraction of underlying device functionality and uses the discovery functionality described in this chapter.

6 Generic Skill Concept & Device Adapter

Partial results of the presented work in this chapter are published in my peer-reviewed publication [Profanter, Breitkreuz, et al., 2019]. I am the main author of this publication, and my main contributions are in further detail described in this chapter. Some figures created and partial text written by me in this publication are directly included in this chapter.

Figure 6.1 is already known from Section 1.5 and highlights the focus of this chapter. In Section 1.1, I described the necessity to have an abstraction of low-level device functionality and its proprietary interfaces, to be able to integrate them into the existing environment. In this chapter, I present a solution for such device adapters based on my generic skill concept. I show that this generic skill concept can be used to adapt any device components, and even software components, to make them Plug & Produce ready.

6.1 Capability and Skill Definition

Before I describe the generic skill model, I give a definition of the terms *Capability* and *Skill*. The following definition is based on the publication[1] [Perzylo, Grothoff, et al., 2019]:

> In general, a skill provides the (executable) capability of something to cause an effect on something. [...] a skill may be described by a set of properties. Thus, a skill can be considered a property carrier. [...] a skill can be characterized by its input, output, and transient conditions.

This quote already contains a very compact definition of a Skill and is similar to other relevant publications, like [Bedenbender et al., 2019].

[1] A. Perzylo was a colleague at fortiss where I conducted most of my research for this thesis.

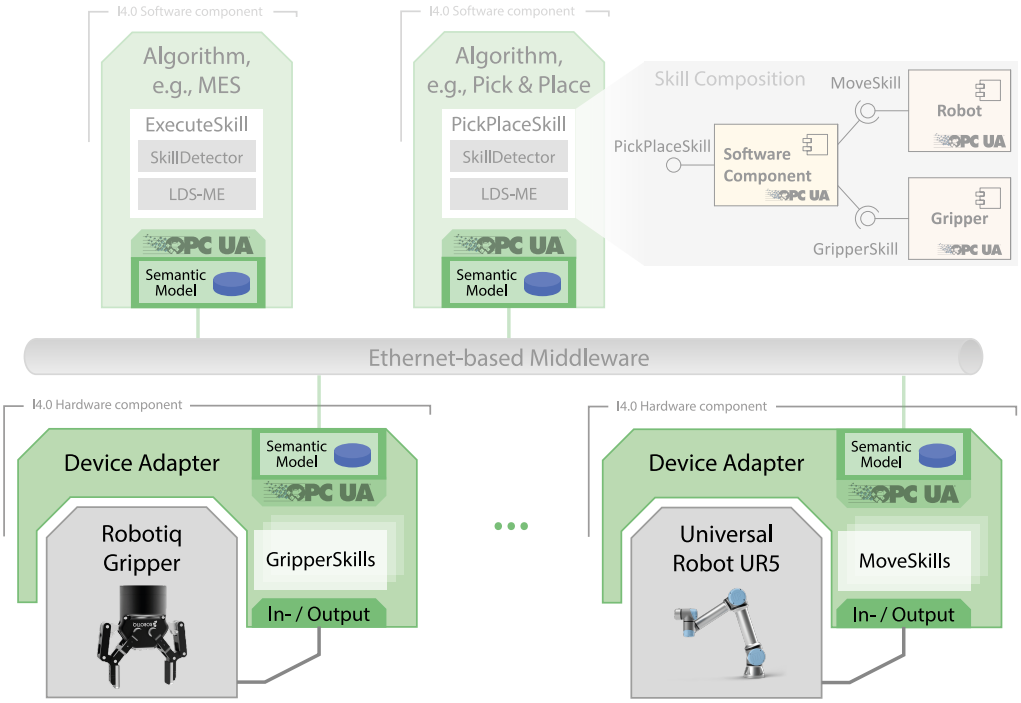


Figure 6.1: Exemplified system setup with different Industry 4.0 components. Chapter 6 focuses on the abstraction of hardware components using device adapters, and the definition of a generic semantic skill model.

A **Capability** is defined as the ability to execute a specific action which causes a specific effect. This effect can be a physical effect (e.g., object position changed), or a non-physical effect (e.g., detecting an object in a vision system). Note that a capability only describes the action and its effect, but not the necessary parameters and execution steps to perform the action. Some example capabilities are: *AttachObjectCapability* (effect: object is attached to gripper or device), *PickAndPlaceCapability* (effect: Object is moved from one position to another), *DetectObjectCapability* (effect: A specific object is detected in the current camera image). The list of capabilities is endless and there is, to the best of my knowledge, not yet a comprehensive established standard to describe different capabilities required for real-world applications.

A **Skill** provides the execution of a specific capability. The capability itself does not specify, how the effect is reached, compared to the skill which consists of one or multiple steps to achieve the desired effect. The skill typically requires a set of parameters to trigger the execution, and provides a set of transition events (e.g., start, halt, suspend, resume) during its execution. The result of the skill execution is either success if the described capability effect is achieved, or an error state if the effect cannot be reached. To further specify the execution result, additional output parameters characterize a skill. An example for a skill is the *PickAndPlaceSkill* which implements the previously mentioned *PickAndPlaceCapability* on a specific robot and gripper combina-

tion. Typical input parameters for such a skill are the pick and place positions, and the required gripper span.

By using capabilities as a reference inside a skill, it is possible to have different specific implementations for the same capability (or effect) inside a system. The capability description can be extended semantically and annotated with additional properties. This is not further described in this thesis as I mainly focus on the execution, parameter mapping, and combination of skills. The skill definitions and examples in this thesis can easily be extended with a reference to a specific capability. The interested reader is pointed to [Perzylo, Grothoff, et al., 2019] which shows a possible approach to semantically annotate capabilities.

6.2 Generic Skill Model

In the previous section, I defined the terms Capability and Skill. To get from the theoretical definition to the implementation of a skill, I define a generic skill model in this section. This model describes the basic requirements of such a generic model, and the interfaces to interact with the skill.

6.2.1 Requirements & Definition

The generic skill model is the basis for having generic hardware-independent component skills. I define the following requirements to be necessary for a generic interface:

- A skill is classified by a corresponding **type definition** and reference to a capability description
- It has a **set of base properties** which identify and describe the skill
- It has an optional **set of input parameters** which configure the skill execution
- It has an optional **set of output parameters** which represent the result of the skill execution
- It can be in various **states** depending on the underlying hardware or the execution result. There is a minimum set of states which every skill must support
- There are methods which trigger **state transitions** between these predefined states
- The skill implementation may trigger a state change internally, e.g., if the execution is finished, or an error occurred

As noted in previous chapters, I am using OPC UA as communication platform and basis for semantic models. Therefore, I describe the generic skill model semantically inside the OPC UA address space. Using OPC UA allows having a well-known and easy to integrate interface to the skill. Since OPC UA is already employed in many industrial shop floors, this even lowers the integration effort of the new skill model. While defining the skill model, I am also focusing on building it on already existing well-defined concepts instead of reinventing the wheel. The final model is released on GitHub as open-source[2] and excerpts of the resulting model description are shown in Appendix B.

One could think about a simple model where the skill functionality is simply wrapped by a method call. According to the official specification, OPC UA methods are meant for short-running tasks with a limit of 10 seconds while OPC UA Programs can model more complex long-running tasks [OPC Foundation, 2019b]. In addition, methods are stateless and do not provide any feedback while they are active. Due to these drawbacks, I am using the OPC UA Specification Part 10 "Programs" as a basis of my skill model. A program in OPC UA represents a state machine which provides basic methods to trigger state changes and includes input and output variables for the client. It also defines a basic set of states which every program must support: Halted, Ready, Running, and Suspended. For every state change the OPC UA server emits an event which may contain additional information on the state change itself. Using this approach, it is possible to model long-running processes using the OPC UA information model, while adhering to the modeling principles of OPC UA.

An OPC UA Program already fulfills most of the previously listed requirements. I introduce the *SkillType* as a subtype of the OPC UA *ProgramStateMachineType* and extend it with a set of base properties. The *SkillType* model, including its inherited properties, is shown in Figure 6.2 using the OPC UA notation (see Appendix A). This *SkillType* is the base type of all the skills offered by any system component. It extends the definition of OPC UA Programs with a mandatory *Name* property. Since this type is the base type for all skills, a client can easily browse the whole namespace of an OPC UA server and find all skill instances (see Section 7.2). The *SkillType* inherits the program state machine which provides methods for triggering state changes in the state machine. For every state change, an OPC UA event is emitted by the server, independent if the state change was triggered by a client or internally. A client can create monitored items to receive event notifications and therefore be immediately notified of these state changes. Since an OPC UA Program is based on the FiniteStateMachineType of OPC UA, the list of predefined states is fixed. However, it is still possible to create sub-states for these predefined states, which makes this state machine very flexible.

[2] https://github.com/pro/opcua-device-skills

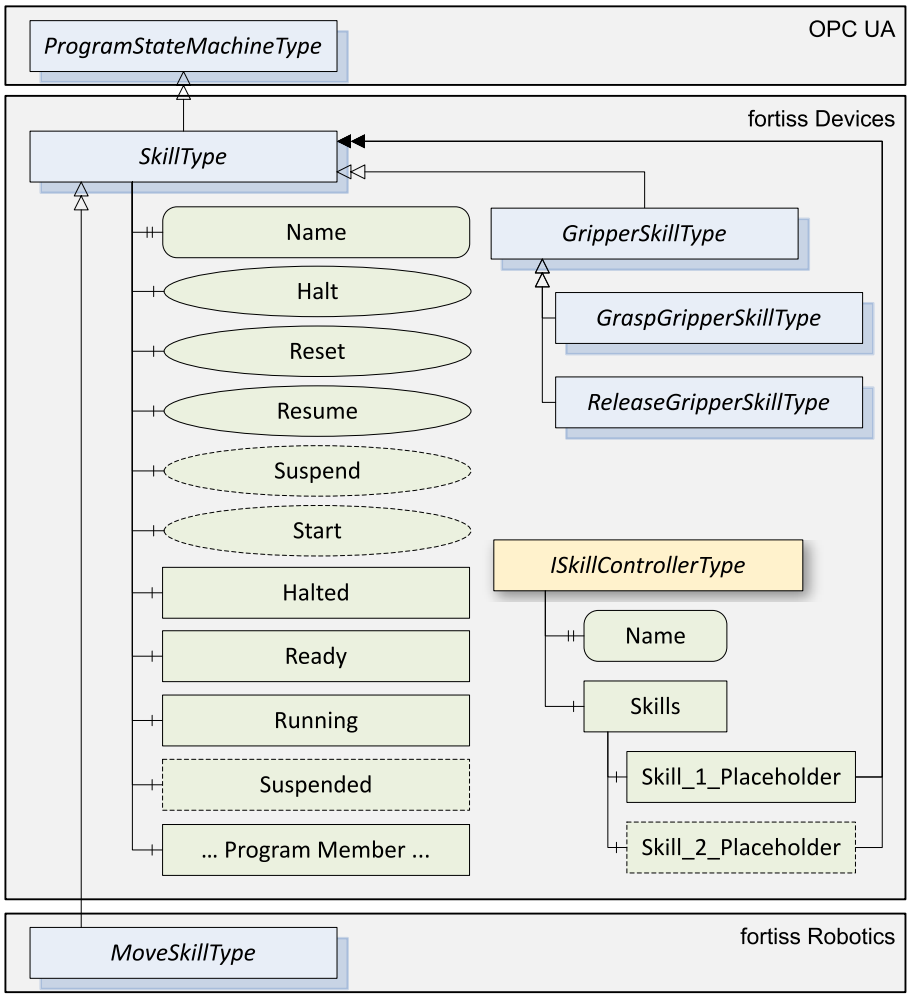


Figure 6.2: OPC UA model of the *SkillType* in OPC UA modeling notation and its subtypes (blue). *SkillType* is a subtype of a *ProgramStateMachineType*. It adds additional parameters (green) to the inherited children. *ISkillControllerType* is an Interface (yellow) grouping all the supported skills of a component inside the *Skills* object. The *GripperSkillType* is an example for a specific skill subtype. Further details on the OPC UA notation can be found in Appendix A.

Additionally, I define the *ISkillControllerType* interface. Every component which implements skills must implement this interface and list the supported skills at least inside the *Skills* object. Thus, a skill instance must be listed inside the *Skills* object, and optionally also as a child of other nodes in the address space. An OPC UA interface adds additional mandatory and optional modeling rules to the implementing object node, similar to object-oriented interface programming. This ensures that all the skills are listed inside a common well-known *Skills* object node. This allows any client to immediately get a list of all the available skills without browsing the whole information model. Inside the

OPC UA address space a node can have multiple hierarchical references, and therefore the same node instance can appear in multiple places.

The state change methods to control any instance of the *SkillType* always remain the same for all the specific skill implementations. This provides a generic interface for all clients. The evaluation in Chapter 8 shows that the same interface is supporting many different skill implementations. A skill can be parameterized using the *ParameterSet* object as shown in the example in the next section. The *ParameterSet* contains variables which need to be set by the client. For example, the target joint values for moving a robot are set for the corresponding variable inside the parameter set. The client then calls the *Start* method which starts the execution of the skill which internally reads the parameters and performs the movement. Every skill type has predefined states and transitions. Starting the execution of the skill is only allowed if the skill is in the corresponding ready state.

Using this concept, it is possible to easily model a huge set of generic functionalities as a skill. Interface-wise, only the parameter set changes. The skill type can be used for various hardware: robots, tools, cameras, manufacturing machines, and many more. My model is not limited to only hardware but can also be used for software components: A software component can provide more complex functionality by reusing skills of other components. For example, this could be an object detection software component which takes a camera image as input, and outputs the detected objects. With this, one can simply exchange the underlying implementation while keeping the same interface to the algorithm.

In addition, it is possible to hierarchically compose skills, as described in more detail in Section 7.3. A robot controller can implement a movement skill, and an attached gripper enclosed by a separate OPC UA server can provide a corresponding gripper skill. The robot and gripper skills are completely independent. A new software component references to the *PickPlaceSkillType* to implement a basic pick-and-place functionality, by synchronizing and controlling the lower-level skill state machines of the robot and the gripper. This software component can be reused for any combination of robot and gripper hardware if both are using my generic skill model for moving and gripping. A robot which already combines a gripper with motion axes can directly provide the Pick-and-Place skill in addition to the other move skills. More details on skill composition is given in Section 7.3.

6.2.2 Application of the Generic Skill Model for Industrial Robots

In this section, I present an example application of the generic skill model for typical industrial robots. Industrial robots in general support two types of target position notation: either in cartesian space or in joint space. In addition, the type of movement from the current position to the specified target position can be a linear movement (LIN, straight line) or a point-to-point movement (PTP, typically faster but not a straight line). Some robot manufacturers support more types of movements, e.g., circular movements, where a third point is specified, and the movement is calculated along a circle. Since the presented model is generic in such a way, that other skill types can easily be added, I will focus on the LIN and PTP movements.

In Figure 6.3 the skill type hierarchy for the most basic robot skills is shown. All skills moving the robot based on a specific tool frame must be subtypes of the *MoveSkillType*. The *MoveSkillType* defines the mandatory *ToolFrame* parameter, which indicates the frame on the robot, which should be used to reach the target. Most robot manufacturers allow defining different frames for different tools.

Point-to-Point (PTP) movements are defined as the abstract *PtpMoveSkillType*, linear (LIN) movements defined as the abstract *LinearMoveSkillType*. Since a PTP movement is based on the robot axes, the required parameters are an array of maximum acceleration and maximum speed values for each axis. The LIN movement requires six acceleration and six speed parameters: the first three for the position, the last three for the orientation. The abstract PTP and LIN move skills are again sub-typed into cartesian and joint movements, where the client can either define the new pose in cartesian space, or by setting the new joint angles. This results in four more skill types: *CartesianLinearMoveSkillType*, *JointLinearMoveSkillType*, *CartesianPtpMoveSkillType*, *JointPtpMoveSkillType*. To reuse the parameter definition for both cartesian skill types and joint skill types, I am using the concept of interfaces: The *ICartesianMoveSkillParameterType* adds two Parameters: *TargetPosition* as the goal position in cartesian space (ThreeDFrameType) and an optional *AxisBounds* two-dimensional array which can limit the solutions of the inverse kinematics calculation. *IJointMoveSkillParameterType* adds the *TargetJointPosition* array which gives the absolute target joint angle for every axis.

The previously mentioned skill types currently take absolute coordinates as input. It is possible to extend the model to support relative coordinates by adding a Boolean parameter to the *MoveSkillType* which allows the distinction between absolute and relative coordinates. Another approach is to add more skill types, one for absolute, and one for relative movements each.

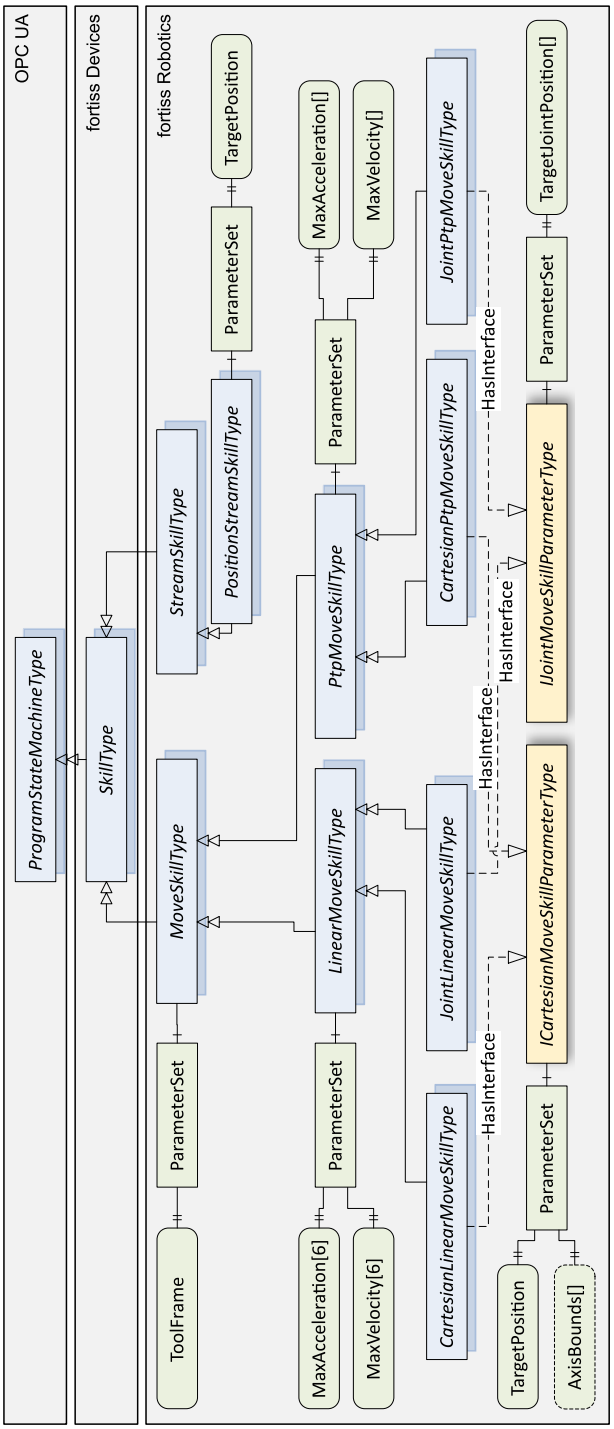


Figure 6.3: Robot skill types which could be implemented by a robot (blue). The required parameters (green) are inherited from the corresponding supertype. Cartesian and joint skill types inherit the interface (yellow) to avoid duplication of parameter sets. The OPC UA modeling notation is used, as described Appendix A.

Some robot manufacturers provide real-time position streaming interfaces. In this case, a remote client is sending new joint positions in a specified control frequency. To support this kind of robot movements, I define the *PositionStreamSkillType* which takes the target joint position as a parameter. As soon as this skill is started, the execution will continuously check for new values in the target position parameter and instruct the robot to move to that joint position. This position streaming skill can be composed by a software component which provides higher-level more complex robot movements. The concept of re-using existing skills is called skill composition and explained in more detail in Chapter 7.

For a more complex setup, where two robots need to move synchronized, position streaming in combination with real-time capable communication is required [Huang et al., 2019]. This is not the focus of this thesis. The interested reader is referred to the VDMA OPC UA working group SOArc (service-oriented architecture and real-time control), where I am also contributing the results of this thesis with the common goal to define a generic real-time capable skill execution interface based on OPC UA Publish-Subscribe in combination with TSN.

My definition of skill types only defines the parameters which are required for the specific skill type. It does not define if and how the robot manufacturer needs to implement the functionality internally and therefore acts as an abstraction layer of the underlying implementation. A robot manufacturer may also decide to only support a smaller subset of the skill types. Every skill type can also be extended by manufacturers to include additional properties such as the control frequency or additional movement types. If these new skill types are part of a well-known standard, clients can automatically use them without the need of time-consuming re-programming.

To be able to evaluate my new skill concept, I extend the newly released OPC UA Companion Specification for Robotics [OPC Foundation, 2019a]. This specification is only supporting read access to robot data. A control interface is still missing and as of August 2020 work in progress, at least on the task level. I am also contributing to that working group to integrate the results of this thesis and to accelerate the development of a common standard.

My extension of the Robotics Companion specification defines a *SkillMotionDeviceType* which implements the *ISkillControllerType* interface in addition to the base objects of the *MotionDeviceType*. The *Skills* object contains instances of the *SkillType* definition and lists all the available skills the robot provides as shown in Figure 6.2.

The additional skill types for a simple gripper are shown in Figure 6.2: *GraspGripperSkillType* and *ReleaseGripperSkillType* define a skill where the gripper is supposed to grasp an object (e.g., close the fingers) or release it (e.g., open the fingers).

The extended skill model and the robotics Companion Specification are available as a NodeSet2.xml file on my GitHub repository[3].

6.3 Device Adapter Concept

In current shop floor setups, different devices from various manufacturers are used, each with several parameters to tune and with proprietary control interfaces. Integrating such devices with varying interfaces into a standardized Industry 4.0 shop floor is called **brownfield integration**. In comparison, **greenfield integration** assumes that all devices are controllable through the same interface specification.

Brownfield integration can be achieved by developing an adapter which abstracts the manufacturer dependent interfaces and provides a standardized interface, as shown in Figure 6.4. In this section, I give a brief overview of device adapters based on my previously presented skill model. A device adapter is implemented in software and can either be deployed directly on the device, or for resource-limited devices on an external controller connected via input/output ports directly to the device. It consists of a low-level proprietary interface to connect to the hardware, and a high-level interface. This high-level interface is using an OPC UA server which contains the semantic model for describing the provided skills and the device properties.

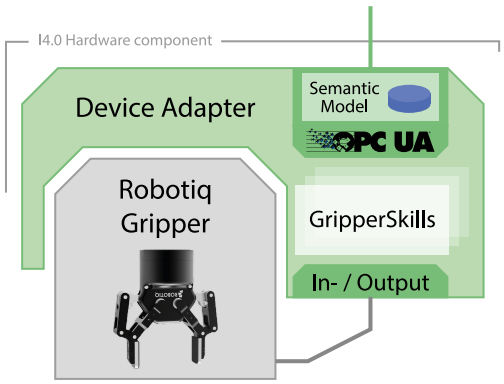


Figure 6.4: A device adapter provides the functionality of a device through adapting its proprietary interface using the semantic OPC UA skill model.

The functionality of the device is offered through semantic skills as defined in Section 6.1. These skills can then be re-used by other components or directly controlled by clients. In my setup the device adapter is responsible for abstracting the lower-level device functionality through skills. Such a device adapter can also be the implementation

[3] https://github.com/pro/opcua-device-skills

subset of the AAS (see Section 2.2.3). The AAS typically includes more functionalities, like component properties or geometric descriptions, which is not in the scope of my thesis.

6.4 Skill Model & Device Adapter Implementation

After the definition of the Skill model and introducing the concept of Device Adapters, I describe in this section how the skill model definition can be transformed into a real-world execution experiment. I also describe the necessary contributions to the used open-source OPC UA stack, to achieve this goal.

In Chapter 4, I have shown that the open62541 open-source implementation of OPC UA is one of the best performing open-source protocol implementations compared to implementations of MQTT, ROS, and DDS and other OPC UA implementations. Therefore, I am using open62541 as the basis for my developed applications. open62541 is a software library developed in C99 while I am using C++ for my experiments and integrating open62541 as a library component.

6.4.1 Information Modeling Pipeline

OPC UA organizes all the nodes inside an address space. This address space can be extended by custom information models as shown in Figure 4.2.

Every custom information model or Companion Specification is released in the official NodeSet2 XML format. This file contains all the nodes and references between the nodes inside this specific information model. For more complex information models, typically the more intuitive OPC UA ModelDesign format is used. It contains the same information as the NodeSet2 format but is less verbose and simpler to read and write. Listing 6.1 shows an excerpt of a ModelDesign XML file, and Listing 6.2 the resulting NodeSet2 XML definition with automatically generated node ids. More complete excerpts of the defined ModelDesign files are attached in the Appendix B.

All major OPC UA implementations support loading NodeSet2 files and initializing their address space correspondingly. Figure 6.5 shows the information modeling pipeline to get from a custom information model to a running OPC UA server on the example of the open62541 stack.

```
<ObjectType SymbolicName="FOR_DI:SkillType" BaseType="OpcUa:ProgramStateMachineType"
    IsAbstract="true">
  <Description>A skill type</Description>
  <Children>
    <Property SymbolicName="FOR_DI:Name" DataType="OpcUa:String" ValueRank="Scalar"
        ModellingRule="Mandatory">
      <Description>Name of the skill</Description>
    </Property>
  </Children>
</ObjectType>
<ObjectType SymbolicName="FOR_DI:GripperSkillType" BaseType="FOR_DI:SkillType" IsAbstract="
    true">
  <Description>A gripper skill type</Description>
</ObjectType>
```

Listing 6.1: Excerpt of a ModelDesign XML Example defining a very basic *SkillType* with a *Name* property and a *GripperSkillType* subtype.

```
<UAObjectType NodeId="ns=1;i=15034" BrowseName="1:SkillType" IsAbstract="true">
  <DisplayName>SkillType</DisplayName>
  <Description>A skill type</Description>
  <References>
    <Reference ReferenceType="HasProperty">ns=1;i=15100</Reference>
    <Reference ReferenceType="HasSubtype" IsForward="false">i=2391</Reference>
  </References>
</UAObjectType>
<UAVariable NodeId="ns=1;i=15100" BrowseName="1:Name" ParentNodeId="ns=1;i=15034" DataType="
    String">
  <DisplayName>Name</DisplayName>
  <Description>Name of the skill</Description>
  <References>
    <Reference ReferenceType="HasTypeDefinition">i=68</Reference>
    <Reference ReferenceType="HasModellingRule">i=78</Reference>
    <Reference ReferenceType="HasProperty" IsForward="false">ns=1;i=15034</Reference>
  </References>
</UAVariable>
<UAObjectType NodeId="ns=1;i=15101" BrowseName="1:GripperSkillType" IsAbstract="true">
  <DisplayName>GripperSkillType</DisplayName>
  <Description>A gripper skill type</Description>
  <References>
    <Reference ReferenceType="HasSubtype" IsForward="false">ns=1;i=15034</Reference>
  </References>
</UAObjectType>
```

Listing 6.2: Excerpt of NodeSet2 XML generated from the previously shown ModelDesign example.

The open62541 stack provides a C Application Programming Interface (API) which is used to add specific nodes and references to the server instance. Looking at the pipeline from right to left, one can manually write such code © by transferring the graphical

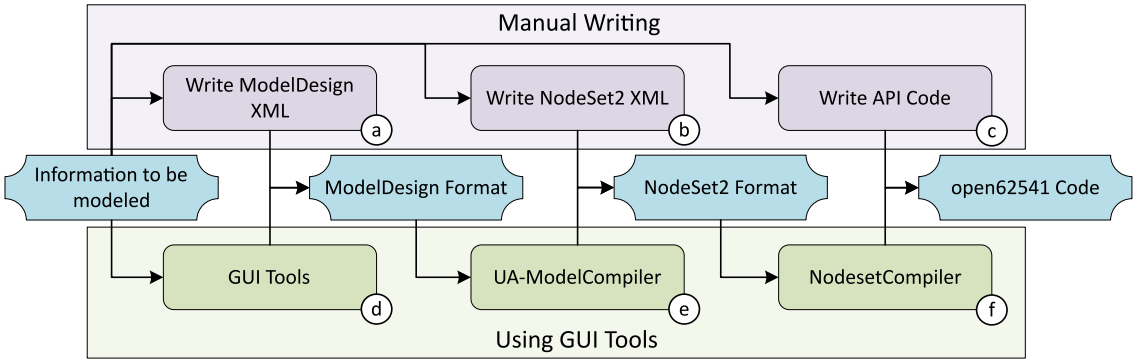


Figure 6.5: Different options for the information modeling pipeline. Starting with the information to be modeled, it is possible to manually write each intermediate format with increasing complexity (ⓐ,ⓑ,ⓒ). Different tools can be used to reduce the modeling effort (ⓓ,ⓔ,ⓕ).

drawings into C code. The maintainability and transferability in this case is extremely low and there is a high risk that the code is inconsistent.

A more generic approach is to manually write the NodeSet2 XML file ⓑ based on the graphical model. This file is then transpiled with the open62541 nodeset compiler ⓕ to the corresponding C API code. The resulting code is used during compile time to initialize the server. Official Companion Specifications are always released at least in the NodeSet2 XML format.

Since the NodeSet2 XML format is very verbose, the OPC Foundation defined the ModelDesign XML format and provides the open-source UA-ModelCompiler[4] ⓔ to automatically generate NodeSet2 XML files out of ModelDesign. Compared to NodeSet2, ModelDesign is more object-oriented and easier to write and read. There are very few graphical GUI tools which support the export as ModelDesign format ⓓ and these tools only support a limited set of features.

Therefore, the most recommended and generic approach to get from a graphical information model to the final open62541 code is to manually write the ModelDesign XML file ⓐ, and then use the UA-ModelCompiler to generate NodeSet2 files. These files can then be used in other OPC UA stacks, or as an input to the open62541 model compiler to generate open62541 source code. This is also the approach I am using in this thesis.

The open62541 nodeset compiler is written in Python and converts NodeSet2 XML files to compilable C code, which directly initializes the OPC UA server. At first, the open62541 stack only supported a limited set of NodeSet2 definitions. As part of this thesis I extended the open-source implementation of the nodeset compiler to support the

[4]https://github.com/OPCFoundation/UA-ModelCompiler

full set of NodeSet2 features. This was necessary as my generic skill model is released in the NodeSet2 format and uses different features of that format. The pull requests leading to the current almost complete functionality of the nodeset compiler can be found on GitHub[5] and are included in recent releases of the open62541 stack.

My generic skill model is written in the ModelDesign format. The UA-ModelCompiler is then used to generate the NodeSet2 XML. To simplify the full pipeline, I integrated various CMake macros into the open62541 repository. During compile time, NodeSet2 files are automatically transpiled and no manual step is needed. I also developed a Docker Container for UA-ModelCompiler to run it with a single command. A full tutorial on how to create custom NodeSet2.xml is available online on my web page[6].

6.4.2 Generic C++ class model

The nodeset compiler of open62541 converts the NodeSet2 XML format into C source code which automatically creates all the defined nodes and references in the OPC UA server. This means that for every instance of a SkillType, all the mandatory nodes are created, but the functionality, especially the handling of the state machine and the skill functionality is still missing.

This needs to be implemented in addition to the generated code using the open62541 server API. To reuse as much code as possible for different device-specific implementations, I am using object-oriented programming with class inheritance in C++. Figure 6.6 gives an overview of the defined classes.

I define a generic abstract `Program` class which registers the method callbacks for the state transition methods of an OPC UA Program using the provided server API and handles the event triggering for state transitions. The `SkillBase` class extends the `Program` class and acts as a basis for all skill implementations. Due to this abstraction, a specific skill implementation only needs to implement the hardware interface and does not need to handle the OPC UA specific configuration. This is achieved by using lambda callback functions, which are a feature of the C++11 standard. The first step of a client is to set the parameters of a skill and then call its start method. This method call is triggering the lambda callback function in `SkillBase` and forwarded to the concrete skill implementation. Skill parameters are statically defined using template methods, and transparently initialized in the `readParameters` method to be used in the skill implementation. State transitions and event handling are done in the `Program` class based on the return value of the lambda callback.

[5] https://github.com/open62541/open62541/pulls?q=is%3Apr+author%3APro+nodeset+
[6] https://opcua.rocks/custom-information-models

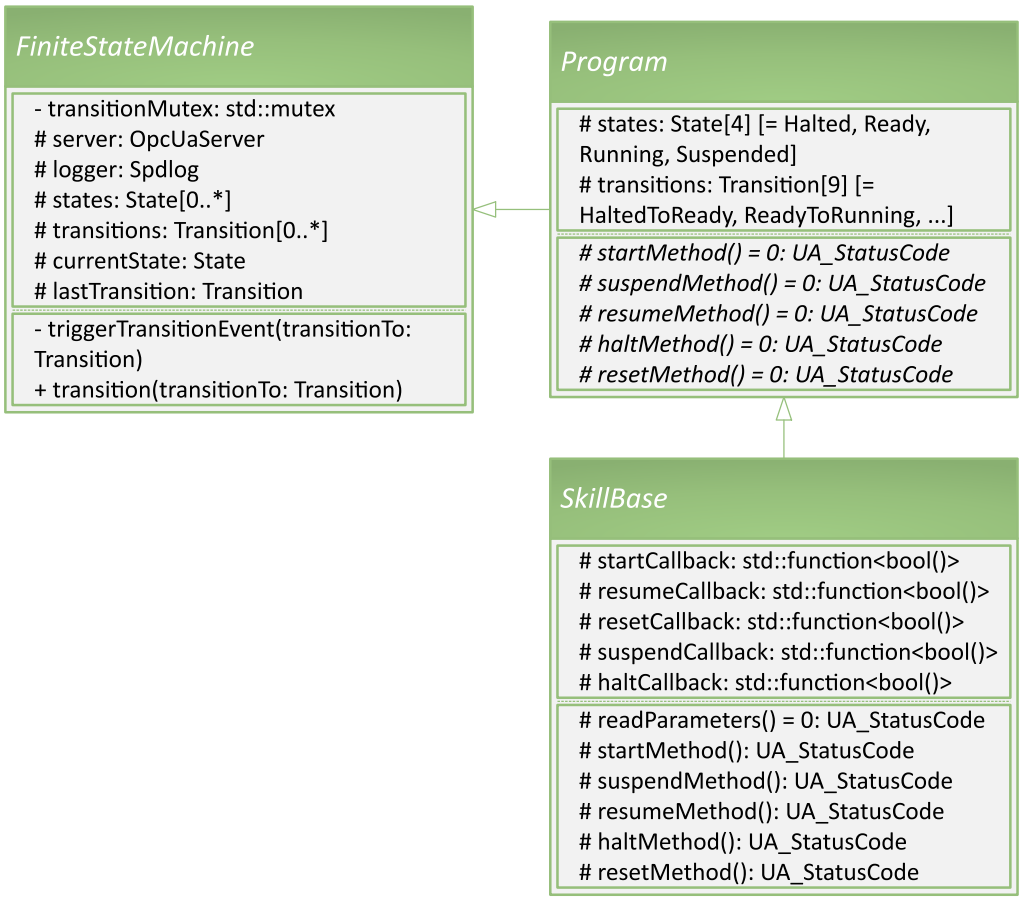


Figure 6.6: UML Class Diagram for a selection of C++ classes and their class members for the generic skill model implementation. All classes are abstract. A specific skill type inherits from *SkillBase* and sets the corresponding method callbacks.

Event notifications are an important feature of OPC UA Programs which was missing in the open62541 OPC UA Stack at the time I started my implementation. Therefore, I supervised the development of the events support for open62541 together with a student as part of his bachelor thesis [Breitkreuz, 2019], and extended the features of the nodeset compiler to be able to include any NodeSet2 XML into the open62541 stack. These improvements are available to the open-source community as part of the open62541 stack and are alread included in the master branch[7].

[7] https://github.com/open62541/open62541/pull/1739

6.5 Summary & Discussion

In this chapter, I present the concept of a device adapter. Device Adapters are necessary for brownfield integration where proprietary interfaces need to be adapted by an additional software to a standardized interface. The device adapter uses a generic skill model which allows easy integration of system components while keeping the same interface description. My skill model is described as an OPC UA nodeset, while the implementation is done using the open-source OPC UA stack open62541 and my own C++-based skill implementation. In the next chapter I present the software components which are using the skills of the hardware devices and compose more complex skills based on the device's functionality. It allows to re-use skills inside other skill implementations and therefore achieve plug & produce for different components.

One of the major issues of OPC UA throughout the industry is that there are currently only a few devices available, which provide an OPC UA interface, even fewer support specific OPC UA Companion Specifications. My proposed structure and NodeSet2 definition keeps the specific implementation and controlling of hardware intentionally open, so that the effort to implement this interface is kept as low as possible for commercial and open-source OPC UA hardware vendors. Still, adapting new interfaces especially in the domain of industrial automation is a slow process and manufacturers need to be convinced to adapt such proposed interfaces. To achieve this goal, I am also participating in two VDMA OPC UA Companion Specification working groups: OPC UA for Robotics and OPC UA for Service-oriented Architectures and real-time control[8]. While the first one aims at standardizing the control interface for robots, the latter is focusing on a skill interface for various hardware and software components.

[8] https://opcua.vdma.org/

7 Plug & Produce

Partial results of the presented work in this chapter are published in my peer-reviewed publication [Profanter, Breitkreuz, et al., 2019] and my publication submitted for review [Profanter et al., 2020]. I am the main author of these publications, and my main contributions are in further detail described in this chapter. Some figures created and partial text written by me in these publications are directly included in this chapter.

Flexible component integration is one of the major challenges in Plug & Produce production environments. The main idea behind the Plug & Produce concept is derived from the well-known Plug & Play concept in the domain of computer systems: A USB device can be plugged into a computer and is immediately available to be used without the need to manually program a driver for it. Achieving the same level of automated configuration and interface description in manufacturing shop floors is still a major challenge. The Multi-Annual-Roadmap (MAR) of the EU SPARC programme [SPARC, 2020] especially identifies configurability as one of the key system abilities of Plug & Produce systems. Standardized interfaces and the identification of system interconnection points are basic requirements to achieve automatic configuration. By implementing Plug & Produce in the domain of industrial automation, more flexible production shop floors, short setup times, and easy reconfiguration are achieved.

In Chapter 4, I compare different middlewares and show why OPC UA is a very performant protocol, and an ideal basis for a generic Plug & Produce implementation. The discovery capabilities of OPC UA are described in Chapter 5 and allow automatic discovery of components in the network. Integrating proprietary hardware devices is achieved via device adapters which provide an adaption from specific interfaces to a more generic skill interface (Chapter 6). In Chapter 6, I also show the typical information modeling pipeline for OPC UA and the C++ class model for implementing such skills. In this chapter, I mainly focus on the software components which re-use skills of other components and either compose new skills out of the base functionality or coordinate the execution of lower-level skills. Since only the skill implementation itself changes, but not the interface, a Plug & Produce system can be achieved. Figure 7.1 is already known from Section 1.5 and highlights the focus of this chapter.

Standardized skill interfaces and parameters facilitate easier component exchange and automatic parametrization with a focus on reusability of skills across different platforms

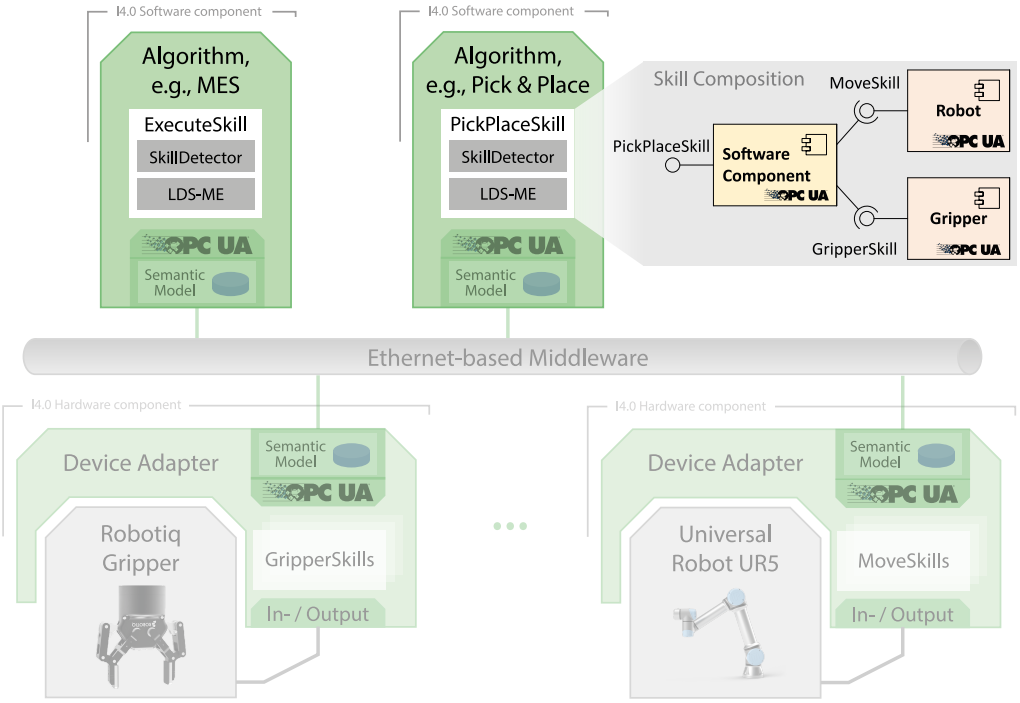


Figure 7.1: Exemplified system setup with different Industry 4.0 components. Chapter 7 focuses on the software components in the system and the composition of skills.

and domains. The hierarchical composition of such skills allows additional abstraction and grouping of functionalities.

7.1 Generic Skill Client

In previous chapter, I define the generic skill model and show its usage inside device adapters. These device adapters transform proprietary interfaces to the generic skill model. The defined skill model includes a state machine representing the current execution state of the skill, and a list of input and output parameters. To control the skill execution, methods of this state machine need to be called. State transitions are reported through the OPC UA event subscription mechanism. Parameters are set through OPC UA write requests. Figure 7.2 shows the necessary interaction between a client and the skill server to parametrize a skill, execute it, and monitor its execution state.

The whole control interface of the skill is accessible through standard OPC UA functionality and can therefore be controlled by any OPC UA implementation. Since my proposed Plug & Produce system is mainly developed in C++, I decided to create a generic

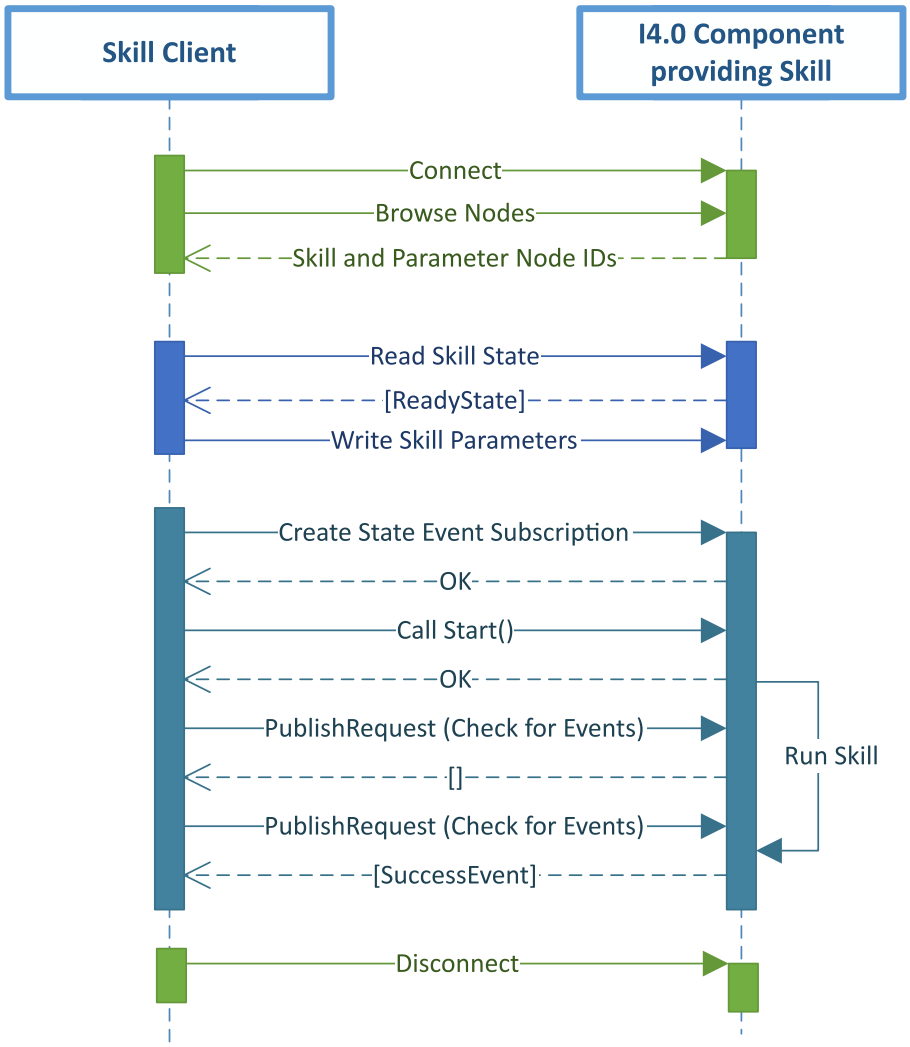


Figure 7.2: Necessary interaction steps of a client to control a skill implementing the proposed generic skill interface.

Skill Client Class in C++ which provides convenience methods to the programmer and abstracts the complexity of the OPC UA protocol.

It is used to initialize the OPC UA TCP connection to the server, call the state control methods (`Start`, `Halt`, `Reset`, `Resume`, `Suspend`), subscribe to state transitions and trigger a C++ callback, to set a parameter with a given name to a specific value, and to get the result data of the skill execution. Figure 7.3 shows the UML Class Diagram of the generic skill client class.

`SkillClient` wraps the OPC UA client implementation with additional helper functionality, i.e., method calls for state transition trigger, monitoring of the skill state, or

setting skill parameters. `GenericSkillClient` extends `SkillClient` with additional convenience methods, such as setting a skill parameter based on its name.

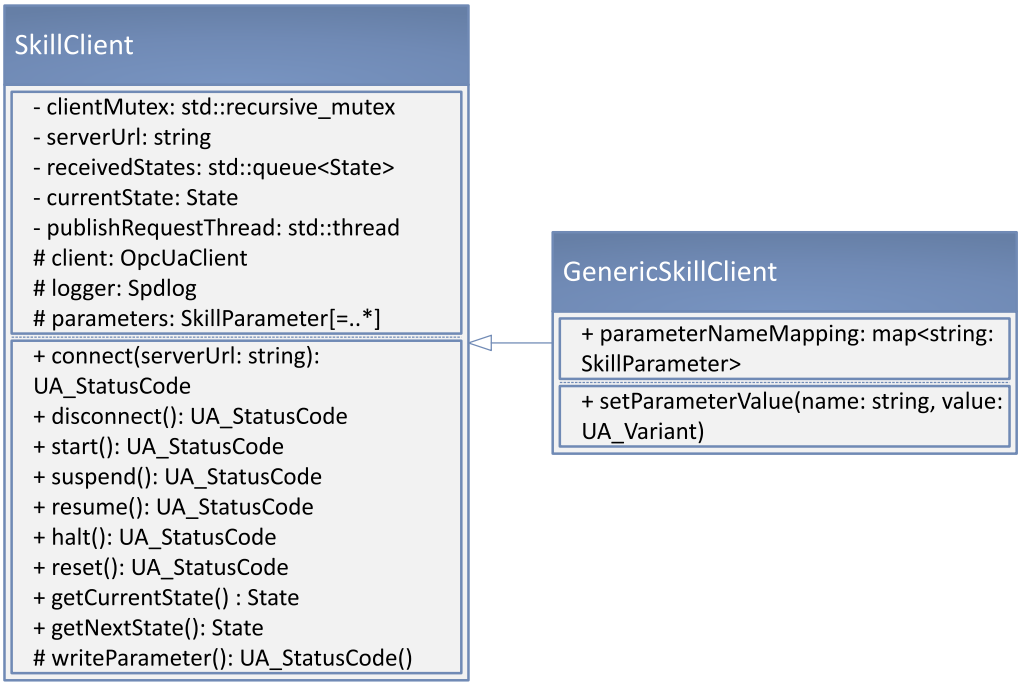


Figure 7.3: Partial UML Class diagram of the generic skill client and the provided functionality.

This skill client is used in the software components presented subsequently to control any device or component which supports the generic skill interface.

7.2 Automatic Skill Detection

In Chapter 5, I describe the OPC UA discovery mechanism, which can be used to find instances of OPC UA servers in the network. In my proposed system architecture, every component (hardware or software) is accessible through the interface of such a server. The OPC UA discovery mechanism only detects server instances, i.e., I4.0 components in the network, but not the specific skills which are provided by that component.

Therefore, every component which is using skills from other components in a Plug & Produce system needs to include a Skill Detector. The Skill Detector is a separate C++ class and manages the listening to server announcements, and the detection of available skills on newly plugged devices. In addition, it also handles component disconnect by listening for the corresponding mDNS messages and removing the skills of that

component from the available skills list. The remainder of this section describes the functionality of the skill detector, and how it is used by the component.

In Section 6.2, I define the *ISkillControllerType* interface. A component, which provides skills according to my generic skill model, needs to implement this interface. Implementing an inferface in OPC UA is achieved by adding a *HasInterface* reference from the main device node to the *ISkillControllerType*.

The implementation of the *ISkillControllerType* tells a client that this device supports the generic skill model, and all available skills of that component are listed under the *Skills* node. The main device node can be in any place inside the OPC UA address space below the *Objects* folder. In addition, a component can have multiple device nodes, i.e., if it consists of multiple subcomponents with different skills, although the typical use case is only one main device node.

Finally, to get a list of available skills provided by a component, the skill detector performs the following steps:

1. Connect to the newly discovered server
2. Check the namespace array if it includes the predefined namespace. If not, the server is not implementing any nodes presented in this thesis.
3. Recursively browse the *Objects* folder for any nodes which include a reference to the *ISkillControllerType*
4. For all such nodes read all children inside the *Skills* node and get the referenced skill type
5. Return the list of found skill nodes to the calling client, i.e., the skill detector.

As soon as a new OPC UA instance announces itself, the skill detector is triggered and starts browsing the remote server recursively for implemented skill types, as shown in Figure 7.4. Based on that information it updates its internal hash map which maps specific skill types to OPC UA server instances and node IDs. The component will use this map to check if a specific skill is currently available in the system.

In OPC UA every object node has a specific associated object type. This object type can be extended by subtyping, similar to object-oriented programming. Section 6.2.2 shows an example for specific skill object types.

The list of skills which is reported by the steps above contains the type reference for every skill. In other words, it contains the referenced skill type, e.g., the *CartesianPtpMove* skill is an instance of the *CartesianPtpMoveSkillType*. With this additional piece of information, the higher-level client code immediately recognizes the semantic meaning and functionality of the found skill. If there are multiple skills of the same type available, it

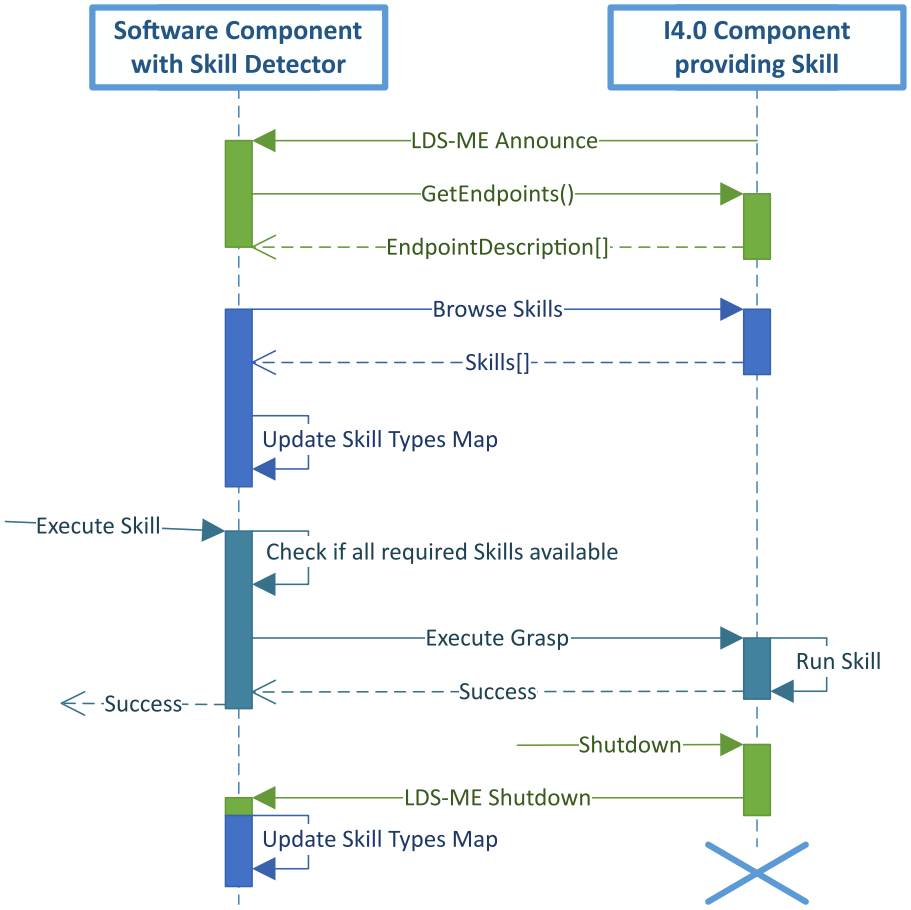


Figure 7.4: Communication sequence between a software component using a skill detector and an I4.0 component providing a skill. Sequence: server announcement, skill detection, skill execution, and component shutdown. The skill detector always keeps an up-to-date map of available skills.

is up to the higher-level client to gather additional information on that component and decide based on this which skill to choose.

Since the interface for all skills is identical, and only the parameters change, the client can control any skill it finds (see Section 7.1). On the other hand, it does not make sense to control a skill where the client does not know the semantic meaning and its physical effect. Thus, the client implementation should use the associated skill type to infer its meaning and ignore unknown types. As mentioned in Section 6.2, the base type of every skill is the *SkillType*. Companion specifications extend the skill model with additional, more specific, skill types as shown in Section 6.2.2. These skill types assign a specific semantic meaning to the skill and define the required and optional parameters.

This implies that a specific set of supported skills is integrated in the client application, and the client can only handle these types. Due to the fact that skill types are built up hierarchically, and it is defined that more specific skills such as *ForceCartesianPtpMove*, which is a subtype of *CartesianPtpMove*, can also be executed through its more generic type, the client does not necessarily need to know the most specific skill type. It can also just use the more generic interface of a higher type to control the skill. The implementation of the skill then must make sure it is executing its functionality according to the defined skill type specifications, which is in more detail described in Section 8.2.

In more complex setups, the skill type can be associated with additional knowledge modeled in an external knowledge database. A generic skill client is then just instructed by the higher-level implementation to execute a specific skill with a specific set of parameters. This use-case is also shown as part of the evaluation in the next chapter.

7.3 Skill Composition

An I4.0 component typically provides one or multiple skills to higher-level components. Such a higher-level component can either directly control the used skills based on its implementation, or it can provide a higher-level skill functionality by using lower-level skills and combining their functionality. I define this combination of lower-level skills to form higher-level functionality as the concept of skill composition.

Figure 7.5 gives an example for skill composition. On the right-hand side there are three types of robot and tool combinations, each offering a specific set of skills:

- Robot with built-in Gripper: This is a robot, where the gripper is directly attached to the end of the robot arm and cannot be easily removed. An example for such a robot is the Comau e.DO. Its controller should provide the base robot move skills (PTP and linear movements with cartesian or joint target), and gripper skills, since the gripper is directly controlled by the robot controller. Combining the move skills and gripper skills results in a pick and place skill, which can also be directly provided by the controller.
- Robot with separate gripper component: the robot controller only provides the move skills; the gripper controller only provides gripper skills. Both do not communicate directly with each other.
- Robot with low-level position streaming interface

At the end, the goal is to get a *PickPlaceSkill* independent of the low-level robot type. In the first case, the robot's device adapter directly provides the PickPlaceSkill. In the second case, an additional software component is required which uses the move and

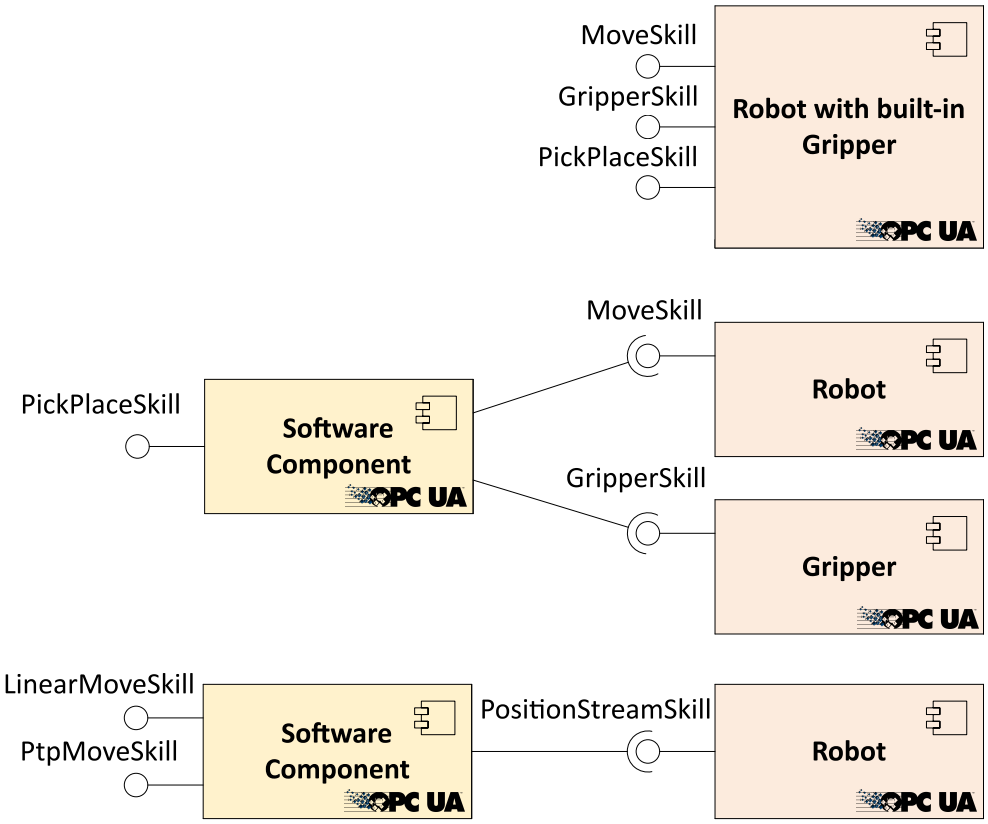


Figure 7.5: Hierarchical composition of skills. Different robot types can offer different skills. A robot with built-in gripper can directly provide a *PickPlaceSkill*. If there is a separate robot and gripper component, their functionality can be reused by a software component which provides the same *PickPlaceSkill*. A basic robot controller may only offer a *PositionStreamSkill*, which is used by a generic software component to offer higher level skills. These skills can then be reused, e.g., by the PickPlace Software Component.

gripper skills of the lower-level components and provides a PickPlaceSkill by internally synchronizing the execution of the used skills. For the third case, first a software component is required which uses the position streaming skill to provide higher-level robot move skills. This component needs to implement the corresponding forward and inverse kinematics of the robot, and with that also needs to know more about the robot model itself. This software component then can be used instead of the robot component in the second case, and therefore a PickPlace skill is available by composing the skills of multiple other components.

The example above shows how powerful the standardized skill model is. In the composition shown in the middle, the software component can be used with any robot and gripper combination, which implement my generic skill model and the specific skill types, and offer a standardized PickPlaceSkill. The parameters of the offered PickPlaceSkill are

again standardized, to allow further composition to reach even higher complexity in the skill functionality and abstract the low-level functionality.

It is important to mention that my skill model does not define, how the implementation of a skill must look like. It only defines the required mandatory and optional parameters and the semantic meaning or physical effect, which a client can expect when executing the skill.

7.4 Plug & Produce Architecture Definition

Based on the concepts described in previous sections and chapters, this section defines the overall Plug & Produce architecture and its components. The specific system setup for a Plug & Produce system based on generic skill models may vary depending on used components and the application domain. As a result, there might not be the one and only solution. The presented architecture proved to be robust and flexible enough for various use cases, as shown in the evaluation in Chapter 8. Figure 7.6 shows an overview of the required modules for a robotic Pick-and-Place system.

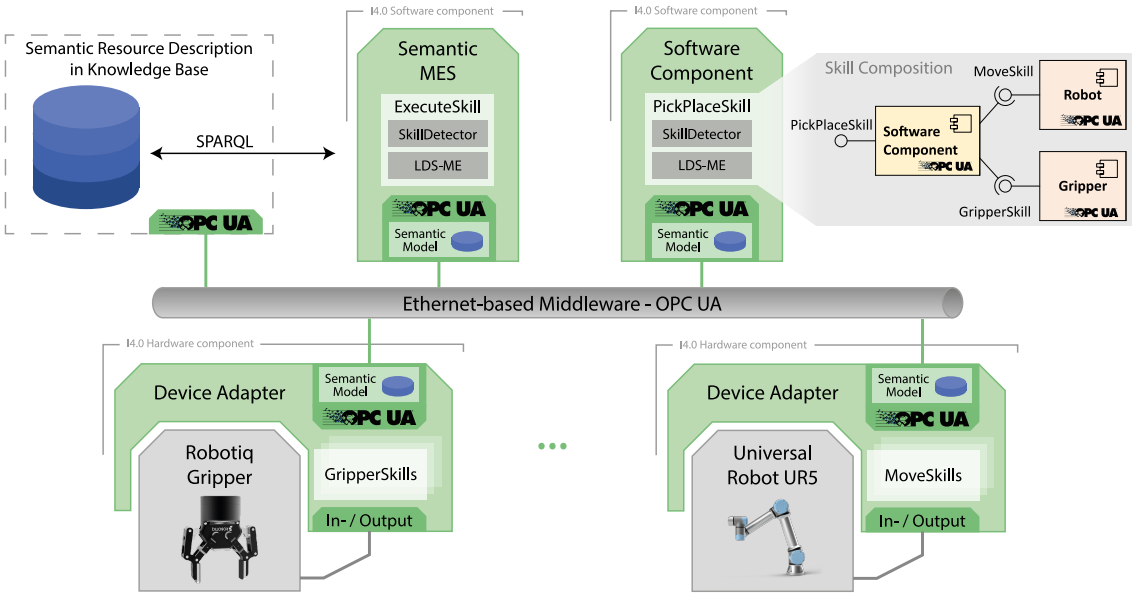


Figure 7.6: Possible system architecture to achieve a Plug & Produce System with generic device skills.

Almost all modules are described in previous sections: OPC UA as Ethernet-based Middleware (Section 4.2.1), component discovery with LDS-ME (Section 5.2), the generic semantic skill model (Section 6.2), device adapters (Section 6.3), the skill detector (Section 7.2), and the composition of skills (Section 7.3).

Based on these presented achritecture modules, software components provide higher-level skill implementations, e.g., to produce a specific part, or in the basic example above, to provide a Pick-and-Place skill. Lower-level components in the resulting Plug & Produce system can be easily exchanged without the need of adapting the higher-level control applications.

To deploy a specific manufacturing process to this system, an additional module is required which is responsible for parametrization of high-level skills based on additional knowledge on the process, product, and resources. Typically, such a module is called Manufacturing Execution System (MES).

In general, a MES is responsible for managing and monitoring the execution of tasks on relevant devices on a shop floor. My definition of a semantic Manufacturing Execution System (sMES) extends this functionality by using semantic information available in the system, especially in a connected Knowledge Base (KB) as shown in Figure 7.6. In a generic Plug & Produce system the sMES is the main component, which orchestrates and triggers the top-level skills. The sMES itself is embedded in a superordinate system that takes care of the planning and scheduling on the factory level. For the invocation of the sMES, a high-level process description is used for parametrization, which includes a set of partially ordered subtasks and a semantic description of requirements that potential target components have to meet. The low-level skills are dynamically selected based on their suitability and availability in a particular production environment.

The KB contains models of the used high-level skill types and their parameters for the process or product and is responsible for persistently storing all relevant knowledge of the production system. It provides both, an OPC UA-based and a REST-based interface for enabling other components of the Plug & Produce system to interact with the KB through SPARQL[1] queries and update requests. The knowledge representation itself uses ontology-based semantic description languages that are specified with the help of OWL2[2]. The KB further provides means to interpret the semantic models in order to check for logical inconsistencies and to automatically infer implicit facts from explicitly represented knowledge.

In particular, the sMES makes use of the KB in order to perform the deployment of individual tasks of a manufacturing process to the skills provided by hardware and software components. With this approach a client just needs to send a manufacturing process' identifier to the sMES, which then queries the KB and gets a sequence of skills (types) and associated parameters that are required for successful execution. Since the functionality of the sMES and the KB connection is not the focus of this thesis, further details

[1] https://www.w3.org/TR/rdf-sparql-query/
[2] https://www.w3.org/TR/owl2-overview/

of this concept were developed together with my colleagues at fortiss and published in a research paper [Perzylo et al., 2020].

In that publication we describe a system for fuse insertion which is built upon my proposed Plug & Produce architecture to integrate different system components. Figure 7.7 shows a picture of the resulting work cell.

Figure 7.7: System setup for Knowledge-based fuse insertion build upon my proposed Plug & Produce architecture.

The robot and the gripper provide move and gripper skills. The KB includes semantic knowledge on the process and the resulting skill parameters, i.e., current fuse positions

and target frames. The sMES uses this information to automatically parametrize the skill execution and monitor the skill state. Due to the abstraction introduced by my proposed architecture, system components like the gripper or robot can be easily exchanged with different hardware as long as they provide the same skill interface.

7.5 Plug & Produce Architecture Implementation

Based on the concepts described in previous sections and chapters, I describe in the following paragraphs how one can implement the proposed Plug & Produce architecture using generic device skills.

Device Adapter - Proprietary interface adaption All components in the system need to implement the presented skill interface to be considered for task deployment. The current market situation shows that there are not many robot systems on the market which support OPC UA natively. Robot manufacturers are currently still in the process of implementing basic OPC UA functionality such as reading status variables. Standardization and adoption of more complex interfaces will take some time. Therefore, a temporary solution is needed to integrate non-compatible devices. In general, this can be achieved by implementing wrapper components, which wrap proprietary interfaces to provide the proposed skill interface.

It is similar to develop a new USB mouse. Either a generic USB controller can be used in the mouse, which is already recognized by common operating systems, or a custom driver needs to be implemented that is based on the USB specification and may support special features.

The first step to create a new skill implementation is to decide on its parent type based on already existing skill types, and to define a set of parameters for providing the represented functionality. Every skill needs to be based on my proposed OPC UA *SkillType* to be automatically detectable and controllable by the generic skill client. The created definition then needs to be transferred to an OPC UA NodeSet (see Section 6.4.1), which is the basis for the resulting address space model in the OPC UA server.

Based on my provided source code, which already abstracts away the skill state machine and OPC UA initialization (Section 6.4.2), one only needs to implement the specific control of underlying hardware or communication with other skills. My provided code also includes a generic implementation of a skill client to be used inside a skill implementation (Section 7.1).

In the evaluation chapter I give more specific implementation examples for such device adapters.

Local Discovery Services (LDS) To achieve a real Plug & Produce system, newly plugged-in components must be detected automatically. In Chapter 5, I show how the OPC UA LDS-ME can be used to automatically find OPC UA servers in the network.

As part of this thesis I added support for LDS-ME in the open62541 C++ stack, and the Eclipse Milo stack. Therefore, both OPC UA implementations can be used to implement the device adapters, and on-the-fly support the LDS-ME discovery services. The only step which must be performed is to connect the discovery callback of the stack with, e.g., the skill detector. An alternative solution would be, that components are configured with static IP addresses, which is not practical in flexible setups.

Skill Detector The skill detector is a custom software implementation inside a component which is responsible for finding all available skills offered by other components as explained in Section 7.2. This functionality is also provided as a C++ class which can easily be integrated into custom component implementations.

Typically, low-level components do not need to implement the Skill Detector functionality as they do not depend on other skills. As soon as a component uses one or multiple other skills, it should integrate the Skill Detector to ensure that the available skills of other components are detected. Further details on the skill detection are described in Section 7.2. In Figure 7.6, examples for such composing software components are shown on the upper part, namely the *Semantic Manufacturing Execution System* and the *Pick-and-Place Software Component*. Both re-use skills of other components.

Software Components for Skill Composition Device adapters of hardware components typically provide basic skills for the adapted component only, e.g., a robot device adapter provides robot move skills, while a gripper adapter provides gripper skills. Higher-level components can directly use these skills and implement the corresponding desired functionality. To increase the flexibility and adaptability of the system, it is recommended to further abstract skills by composing new more abstracted skills. An example for such skill composition is shown in Figure 7.6 as the Pick-and-Place Software component. It uses the skill detector to find gripper and move skills in the system and provides a higher-level Pick-and-Place Skill with more abstracted parameters. The implementation inside the Pick-and-Place Skill predefines the sequence of the executed lower-level skills.

Semantic Manufacturing Execution System (sMES) The sMES is orchestrating the execution of the higher-level skills. Its generic implementation is not process specific as it queries the KB for the skill execution sequence, the expected skill types, and the corresponding parameters. Therefore, the sMES implementation can be used for any system, and does not need to be adapted, as long as the KB contains the specific process definition.

7.6 Summary & Discussion

In this chapter, I define additional modules which are required to achieve a Plug & Produce architecture. The generic skill client is used inside a component to parametrize and control any skill offered by other components, given that they adhere to the presented skill model interface. Using the presented skill detector concept, skills offered by components are automatically detected and can be composed to higher-level functionality, which introduces an additional abstraction layer and more flexibility. The sMES in combination with the KB is used to parametrize high-level skills based on product, process, and resource models defined in the KB.

The cost of this flexibility and configurability is performance. The more flexible a system is, the slower it typically performs since the components are and cannot be optimized for one specific use-case. Still, the presented Plug & Produce architecture allows easy exchange of hardware and software components, and is therefore very suitable for small lot size productions where the system setup needs to be adapted to the varying products.

8 Evaluation

Partial results of the presented work in this chapter are published in my corresponding papers mentioned in previous chapters. I am the main author of these publications, and my main contributions are in further detail described in this chapter. Some figures created and partial text written by me in these publications are directly included in this chapter.

In previous chapters, I defined the various concepts necessary to achieve a Plug & Produce system with generic component skills. In this chapter, I am evaluating the presented concepts and techniques on different real-world robot demonstrators with different tools and requirements.

The evaluation is separated into multiple parts. In the first section, I show the evaluation of generic robot move skills: three robots from different manufacturers are controlled through the same skill-based interface to follow the same trajectory along a square. This is followed by an extension of my skill model with new skill types, especially for software components and skill re-usage. The evaluation is concluded by a complete Plug & Produce system evaluation with different software and hardware components (Universal Robot UR5, Parallel Gripper, Vacuum Gripper, Kelvin Tool Changer) with the goal to evaluate the applicability of the Plug & Produce concept in the industrial context while different tools are exchanged via an automatic tool changer.

8.1 Generic Skill Interface for Industrial Robots

In Section 6.2.2, I defined specific skill types for industrial robots. Figure 8.1 is copied for convenience and shows again the skill types and their parameters.

All included skill types can in general be performed by any 6-DOF (degrees of freedom) robot with rotational joints. Linear (LIN) move skills move the robot from one defined position to another in a linear motion, while point-to-point (PTP) movements are along an arbitrary path. Cartesian movements expect a target position in cartesian coordi-

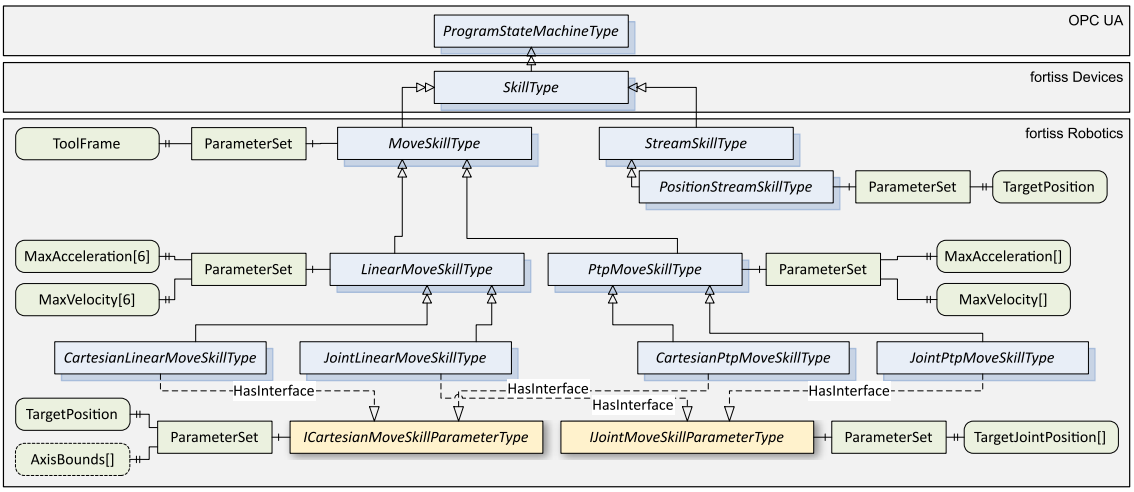


Figure 8.1: Robot skill types and their parameters. Identical to Figure 6.3.

nates, while joint movements expect the target position in the robot's joint angles for each joint.

These skill types only describe the interface and are not robot manufacturer specific, nor define how the skill is implemented for a specific robot. Therefore, I give a specific implementation example of the generic skill model for multiple industrial robots (Universal Robots UR5[1], Comau e.DO[2], KUKA iiwa[3]).

Every specific robot implementation has its own nodeset definition which extends my OPC UA Companion Specification for robots (*fortiss ROB*) by defining specific object instances of the supported move skill types and motion devices, which are defined in the official OPC UA Robotics Companion Specification Part 1 (*OPC UA ROB*, [OPC Foundation, 2019a]). In my implementation I created three additional nodesets for every robot I am using in the evaluation: **https://fortiss.org/UA/iiwa/** (fortiss IIWA), **https://fortiss.org/UA/edo/** (fortiss EDO), **https://fortiss.org/UA/universal_robots/** (fortiss UR). Note that these URLs are the unique namespace URI of the nodeset and not necessarily URLs which can be accessed online.

Currently, in the year 2020, industrial robot controllers do not provide any OPC UA interface which can be used to control such a device. Robot manufacturers provide a non-standardized interface to control the robot through their own defined API. To show the applicability of the generic skill model in combination with industrial robot controllers, I developed C++ OPC UA device adapter applications based on the open-

[1]https://www.universal-robots.com/products/ur5-robot/
[2]https://edo.cloud/
[3]https://www.kuka.com/en-de/products/robot-systems/industrial-robots/lbr-iiwa

source Robotics Library [Rickert and Gaschler, 2017] and implemented the OPC UA Robotics Companion Specification with my own extension of skill-specific Companion Specifications. The Robotics Library provides, among other useful features, a C++ API hardware abstraction and kinematics calculation for different robots.

Based on the already presented generic C++ class model in Section 6.4.2, the implementation for different robots requires implementing the specific lambda callbacks to control the robot based on the skill type, provided parameters, and called state control methods.

The communication with the Universal Robots UR5 robot was already implemented as part of the Robotics Library and the C++ API can be used directly. For the Comau e.DO, which provides a ROS interface, I implemented the corresponding abstraction layer inside the Robotics Library, to be able to use the higher-level functionality provided by the Robotics Library (inverse kinematics, forward kinematics, position control). To control the KUKA iiwa via the Robotics Library I also implemented a custom protocol based on TCP, which opens a dedicated TCP socket to the robot controller to send asynchronous move commands from the client side to a custom developed Java Application on the KUKA controller.

The OPC UA Robotics Companion Specification Part 1 [OPC Foundation, 2019a] defines nodes for various hardware properties of a robot. To initialize the server's address space, a trivial approach would be, to manually write code based on the used OPC UA stack for every robot and all the required nodes. Another approach is to extend the base specification and define more specific Companion Specifications for every robot type, and then automatically generate the initialization code (see Section 6.4.1). This allows to simply reuse the specification as a basis and reduces the implementation effort. Therefore, I created specific ModelDesign files for every robot type and used the pipeline shown in Section 6.4.1 to automatically generate the corresponding initialization code for the open62541 stack. This also results in specific namespace URIs for every robot.

Figure 8.2 shows the dependencies between the Companion Specifications for a specific robot type. The robot-specific information models contain concrete instances of the skill types, and define nodes containing the robot status, e.g., current joint angle values, currents, or temperatures. Note that the address space initialization code only generates the nodes and their references, but does not connect the nodes with their implementation, e.g., providing the current joint value, or connecting a method with its callback (see Section 6.4). My Companion Specifications developed as part of this thesis are available on my GitHub account[4].

[4]https://github.com/pro/opcua-device-skills

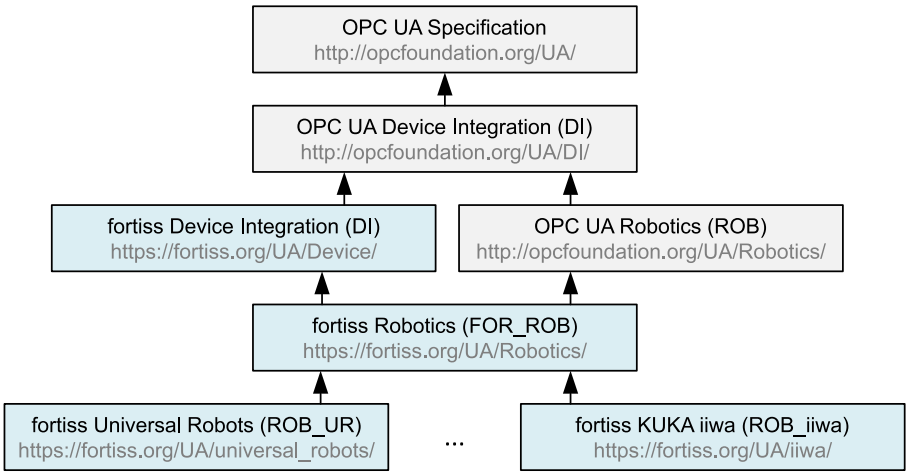


Figure 8.2: Companion specifications for a specific robot model. The arrows indicate a dependency.

Figure 8.3 shows an example of the implemented address space for the Universal Robot Device Adapter. The Robotics Companion Specification [OPC Foundation, 2019a] is extended with additional skill types and axis parameters. In Figure 8.3b the skill state machine properties (transition methods, current state, last transition) and parameters on the example of the *JointPtpMoveSkill* are shown.

Since all my OPC UA servers provide the same move skills independent of the manufacturer, a generic robot skill client based on the concepts of Section 7.1 was developed. This generic client can control any robot implementing the previously described skills without knowing the robot specifications (kinematics, hardware setup). The skill interface can be used for simple robot movements, e.g., to move the robot's tool-center-point from one specific position to another as a linear motion. The movements are only limited by the robot's kinematics and its reachable cartesian space. For more complex trajectories and scenarios, additional skill types can be defined which are used in combination with the kinematics information in the OPC UA address space. In the OPC UA Robotics Companion Specification working group it is currently being defined how the geometry and kinematic information of a robot can be included in its address space.

If an error occurs during the execution of a movement, the state machine of the skill emits a state transition to the *Halted* state which indicates an error state. This error state needs to be confirmed by explicitly calling the reset method on the skill itself. Depending on the underlying implementation and the controller, the state machine may switch to the ready state, or stay in the halted state. It is up to the skill implementation to decide if the current error state allows simple resetting or needs human intervention. A new skill execution can only be started if the state machine is in the ready state.

(a) List of Axes and their parameters

(b) List of Skills and their parameters

Figure 8.3: Implemented OPC UA address space on the example of a Universal Robots UR5 robot. This shows a screenshot viewed by an OPC UA client.

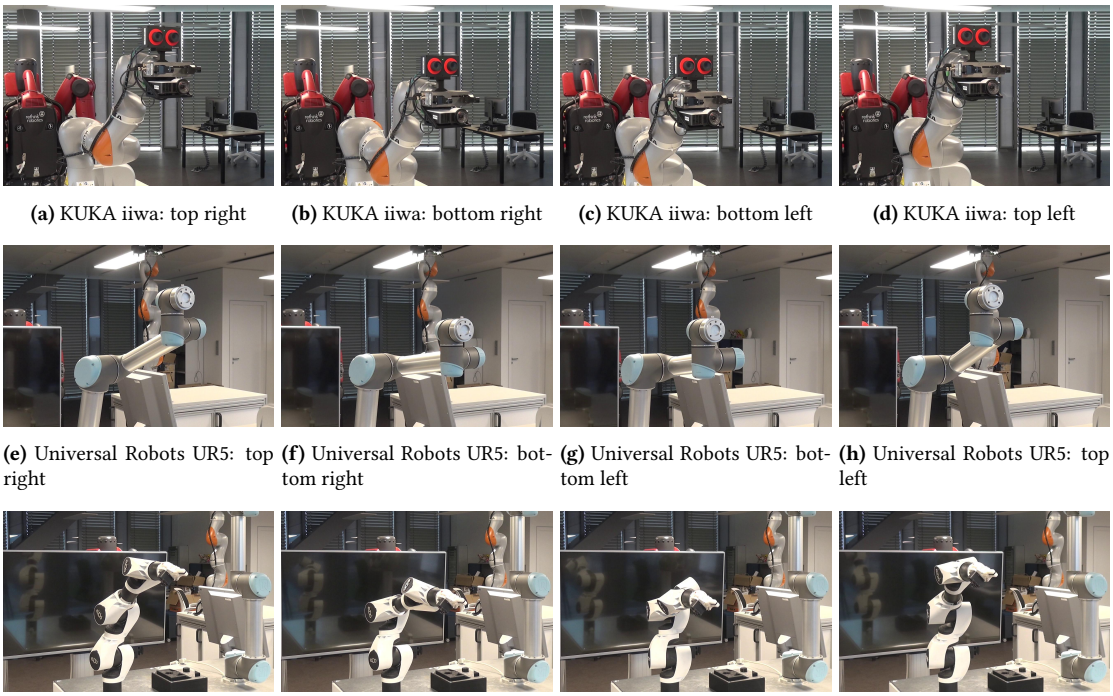
I evaluate the generic robot skill interface by implementing the following move sequence in the robot skill client along a square with the width and height of 10 cm:

1. PTP cartesian move to upper-left corner of square
2. Linear cartesian move to upper-right corner of square
3. Linear cartesian move to lower-right corner of square
4. Linear cartesian move to upper-left corner of square
5. PTP joint move to home position

First, it is important to mention that all robots have a completely different hardware setup and joint lengths. Especially the KUKA iiwa has seven degrees of freedom, whereas the UR5 and the e.DO robot have six. Still, the same *CartesianLinearMoveSkill* can be used for cartesian movements since the robot trajectory is interpolated by the Robotics Library.

The relative cartesian coordinates of the start position vary depending on the robot, since every robot has its own cartesian space it can reach. In my experiment, this start position is configured based on the robot type. In more complex setups, this information could come from a knowledge base or the work cell setup, therefore the client's adaptability is not impacted.

A video of all robots executing these steps is available online[5] and selected screenshots are shown in Figure 8.4.



(a) KUKA iiwa: top right **(b)** KUKA iiwa: bottom right **(c)** KUKA iiwa: bottom left **(d)** KUKA iiwa: top left

(e) Universal Robots UR5: top right **(f)** Universal Robots UR5: bottom right **(g)** Universal Robots UR5: bottom left **(h)** Universal Robots UR5: top left

(i) Comau e.DO: top right **(j)** Comau e.DO: bottom right **(k)** Comau e.DO: bottom left **(l)** Comau e.DO: top left

Figure 8.4: Photo sequences showing the execution of the square movement. Every robot moves along a 10cm square clockwise using the same OPC UA client. The whole execution can be seen in https://youtu.be/w4V9Boh3ZIo.

Using this client my skill model has proven to be very robust and a suitable way for controlling robots using OPC UA. The skill implementation does not depend on the duration of the movement.

[5] https://youtu.be/w4V9Boh3ZIo

The presented *MoveSkillType*s only support sequential robot movements: a new move command can only be sent after the previous execution has finished. If blending robot movements are required, one must use the *PositionStreamSkillType* directly, or create a new software component which takes a list of points and blending configuration to control the robot via its position streaming interface. It is also possible to extend my skill definition with parallel and sequential skill queues where clients can queue up skill execution instances, and the device adapter can automatically execute these skills.

A shortcoming of my skill model is the support of atomic operations. A client first needs to set the skill parameters and then call the start method. In the current implementation, race conditions may occur where another client writes a new set of parameters in-between another client's write and start call. This may lead to dangerous behavior. In my test setup I only allow one client to be connected to the robot. This problem can be circumvented by adding intermediate ready states: If a skill is ready to receive new parameters, it is in the *ReadyNotConfigured* state. As soon as a client parametrizes a skill, the internal state changes to *ReadyConfigured*. While this state is active, clients are not allowed to set new parameters but only to start the skill execution. Another approach is to use the recently updated OPC UA Companion Specification for Devices, which introduces specific parameter locking mechanisms.

One further issue I faced when using the generic client is the different geometries and workspace areas of the robots. Not all robots can reach the same cartesian position. Therefore, the client first checks, to which robot it is connected and adapts the start pose correspondingly. Future planned extensions of the Robotics Companion Specification will include robot kinematic descriptions and geometry information, which allows generic clients to adapt the target position correspondingly or reject the connected robot at all if its reachable cartesian space is too small.

The skill model can be extended for more specific skill types: The KUKA iiwa robot includes force-torque sensors in the joints. Therefore, the device adapter should provide additional force move skills besides the standard move skills. This can be achieved by subtyping the *CartesianLinearMoveSkillType* and adding an optional *MaxForce* parameter. With this hierarchical skill model, the previously presented generic robot client is accessing the base cartesian move skill type, while a more sophisticated robot client can also use the force move skill. If the force exceeds the predefined limit, the state machine has additional states to represent that case, i.e., a *HaltedForce* state. The extension of the skill model with force move skills is a work which was conducted as a Bachelor Thesis of one of my supervised students [Breitkreuz, 2019] and therefore not part of this thesis.

8.2 Skill Type Extension

A specific type definition for a skill instance defines the semantic meaning of the skill and its required and optional parameters. Therefore, the skill type is one of the most essential parts of my presented skill model. All skill types need to be subtypes of the generic *SkillType*, which defines basic properties of a skill, e.g., the various states and state control methods. A subtype defines the parameters for that specific skill type, and by its unique type name (based on the combination of the namespace URI and the node id) the semantic meaning is defined. Clients, like the skill detector in Section 7.2, use the type name to search for a specific skill type. In the previous section, I show the extension of the generic skill model for specific skill types for an industrial robot.

In this section, I show how my generic skill model can be extended with additional types for tool changers, grippers, and software components such as a Pick-and-Place component. These extensions are required for the evaluation in the next sections.

If a skill implements a specific type, all its supertypes are also implemented. For instance, a robotic hand could implement a more specific hand grasp skill type (subtype of GraspSkill), while a parallel gripper could implement a parallel force grasp skill type (subtype of GraspSkill). Both tools still need to support the basic GraspSkill parameters and therefore other components can still use that base skill, even if they only know the GraspSkill type, but not its more specific (manufacturer-specific) types. In this case, the more generic skill execution must intelligently choose default parameters so that the semantic meaning of the base skill is still valid, i.e., the object is grasped. Alternatively, if that is not possible implementation-wise, it must use a different supertype.

Figure 8.5 depicts a simplified overview of the additional skill types that I am describing in this section, grouped by the corresponding Companion Specification. To visualize the model, I am using the official OPC UA modeling notation, as described in Appendix A.

The basis for all specifications is the OPC UA Default Namespace. Based on that, I define my own Companion Specifications: fortiss Devices, fortiss Robotics, fortiss Toolchanger, and fortiss Composite Skills. The two latter ones are new, compared to the types defined in Section 6.2.2.

https://fortiss.org/UA/Device/ (fortiss Devices) extends the OPC UA for Devices Integration (DI) nodeset (OPC UA DI) and contains the definition of the following types: *SkillType*, *GripperSkillType*, *GraspGripperSkillType*, *GraspGripperSkillType*, *ReleaseGripperSkillType*.

https://fortiss.org/UA/Robotics/ (fortiss Robotics) is based on *fortiss DI* and the recently released OPC UA Companion Specification for Robotics Part 1 (OPC UA ROB).

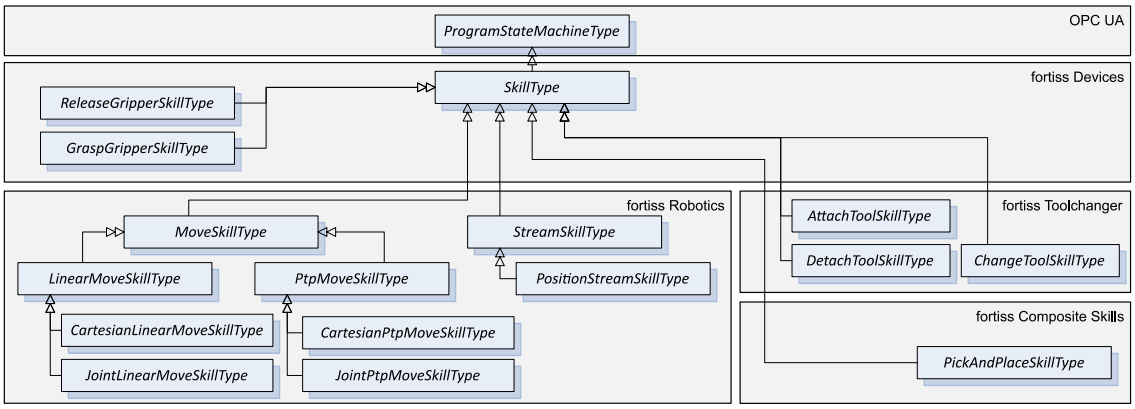


Figure 8.5: Extended OPC UA skill model with robot skill types, gripper skill types, tool changer skill type, and pick-and-place skill type. The OPC UA modeling notation is used (see Appendix A).

fortiss ROB contains the move skill types and parameter definitions as shown in Figure 6.3.

https://fortiss.org/UA/KelvinToolchanger/ (fortiss Toolchanger) is based on *fortiss DI* and contains specific skill types for a tool changer, i.e., *ChangeToolSkillType*, *DropToolSkillType*.

https://fortiss.org/UA/CompositeSkills/ (fortiss Composite Skills) is based on *fortiss DI* and is a collection of various software-based composite skills. As an example, it contains the *PickAndPlaceSkillType*.

Every specific robot and device implementation derives its Companion Specification from these base specifications to add additional device-specific nodes and properties, and especially, to define instances of specific skill types. For example, I created specific Companion Specifications for every robot type in the previous section: **https://fortiss.org/UA/iiwa/** (fortiss IIWA), **https://fortiss.org/UA/edo/** (fortiss EDO), **https://fortiss.org/UA/universal_robots/** (fortiss UR).

Parameters and properties of the *ProgramStateMachineType*, *SkillType*, and skills defined in the fortiss Robotics specification are described in Section 6.2.2 and not discussed in more detail here. To summarize their structure, *SkillType* is the base type for all skill types. The *fortiss Robotics* Companion Specification contains skill types specific to robot hardware, e.g., it defines a *CartesianLinearMoveSkillType* with several required parameters like *TargetPosition* or *MaxVelocity*. The newly introduced skill types and their parameters are described in more detail here.

Both the *GraspGripperSkillType* and *ReleaseGripperSkillType* are generic skills for any kind of gripper. The semantics of a *grasp* is defined as activating the gripper hardware in such a way that a thing or object positioned at the grasp point is getting attached to it. For a simple parallel gripper this could mean closing its fingers, while a vacuum gripper would enable its suction system until an object is attached or a timeout occurs. Depending on the complexity of the skill implementation, skill states can be used to indicate a grasp success or failure. *Release* is defined as the opposite action directly implemented inside the skill, e.g., detaching the object by opening the parallel gripper or disabling the suction system. However, the specific implementation of the skill inside the component is not defined by my model as it differs for specific gripper hardware. If a component implements a specific skill, it must adhere to the defined functionality, to enable other components to rely on it. The grasp skill has a read-only parameter that gives basic information on the grip point of the gripper:

- *grip point offset* (3DFrameType): offset and pose from the tool mounting plate to the grip point
- *grip point type* (Enumeration): gripping type, i.e., parallel, vacuum-based, or multi-finger (hand)

For a robot movement, the grip point offset is required to move the robot with the attached gripper to the correct position. The gripping type is required to adapt the grip point offset based on the object's shape: a vacuum gripper normally picks an object from the top, while a parallel gripper needs to be further down to grasp the object from the side. This definition of a grasp skill is intentionally kept on a basic level to have a generic interface, such that other components do not have to know how an object is manipulated. Specific grippers may implement multiple instances of a grasp skill to represent multiple grasp points or implement more specific subtypes giving the skill callee more parametrization possibilities. In future, it may also be necessary to define a new basic skill type which has a more flexible interface for grip points.

Similar to the gripper skill types, the *DetachToolSkillType*, *AttachToolSkillType*, and *ChangeToolSkillType* define the semantics of a tool change task. The semantics of the detach tool skill is to detach a tool, if there is one currently attached, and to place it at the given location. The attach tool skill attaches a new tool to the tool mounting plate. The change tool skill is a combination of first detaching a tool if present, and then attaching a new tool. In addition, the skills need to include a reference to a movable component, on which they are mounted, e.g., a robot flange. This definition does not restrict how the tool changing steps are performed. For instance, the motion component can be a robot arm, or other moving devices, like linear axes. Therefore, the connected motion component defines the peculiarity of such tool change skills, and they cannot be used as stand-alone skills. In my experiment, which is described in the next section, the tool changer indicates a connection to a robot component and uses its *CartesianLinearMoveSkillType* to reach the tool docking station. As a result, upper-layer components

do not need to directly control the underlying moving device. This is handled transparently by the specific implementation of the tool changer device adapter and its way of parameterizing underlying skills. Detach tool, attach tool, and change tool provide a read-only parameter:

- *move skill controller endpoint* (String): OPC UA endpoint URL of the move component used to reach the docking station.

The move skill controller endpoint can either be configured statically or automatic skill detection can be used to find the correct endpoint.

In addition to it's read-only parameter, the detach tool skill type defines an input parameter *tool position* (3DFrameType) which gives the absolute coordinate frame (orientation and position) where the currently attached tool shall be placed. If the internal detach skill implementation is not able to reach this position, the skill's state machine should change to the halt state to indicate an error.

Attach tool and change tool require two additional writable parameters:

- *tool position* (3DFrameType): absolute coordinate frame indicating the orientation and position of the tool that shall be attached
- *tool app URI* (String): application URI of the tool that will be attached. Required for the skill detector to automatically connect to the tool (via automatic discovery) after successful attachment, and to read the attached tool properties to provide that info to other components

The *PickAndPlaceSkillType* is a composite skill software component, as it reuses other skills that are available in the system, i.e., gripper skills and move skills. A composite skill is a skill component which combines the functionality of other skills in the system to fulfill its own purpose. Pick-and-Place is semantically defined as picking an object (identified by a specific ID) with the given tool, moving the manipulator and placing the object at a given position with a given orientation. I explicitly did not model its parameters to include the object size or grasping parameters. This should be handled by the corresponding Pick-and-Place implementation itself. The caller can rely on the capability (physical effect) that after successful completion the given object is moved to the target position. It does not need to know how this is achieved. To query the object's position, the implementation can, for example, connect to a world model component or use an object detection skill to find the object in the environment. To find suitable grasping parameters, a grasp planning component could be used inside the skill implementation. This generic definition does not limit the trajectory of the robot or other motion components. If more specific options are required (e.g., advanced collision avoidance, move with force feedback), specific subtypes of this generic Pick-and-Place skill can be introduced to represent that functionality for higher-level components.

My definition of a generic Pick-and-Place skill requires four writable parameters:

- *object ID* (String): unique identifier of an object, e.g., used as parameter to query the world model or vision component
- *tool skill controller endpoint* (String): OPC UA endpoint URL of the tool skill controller used to control the tool for grasping and releasing the object
- *move skill controller endpoint* (String): OPC UA endpoint URL of the move component or manipulator used to transport the tool with the attached object from the pick position to the place position
- *place position* (3DFrameType): target position and orientation of the object

Depending on the Pick-and-Place skill implementation, the move component can be a robot with a Cartesian linear move skill or any other component controlling the tool position. There can also be multiple skill implementations of a specific type at the same time with different implementation specifics. It is then up to the higher-level component to select the most suitable one, e.g., based on more specific skill types.

In the next section I present the final evaluation experiment which was conducted to evaluate the additional skill types introduced in this chapter, and more importantly, the whole Plug & Produce architecture presented in my thesis.

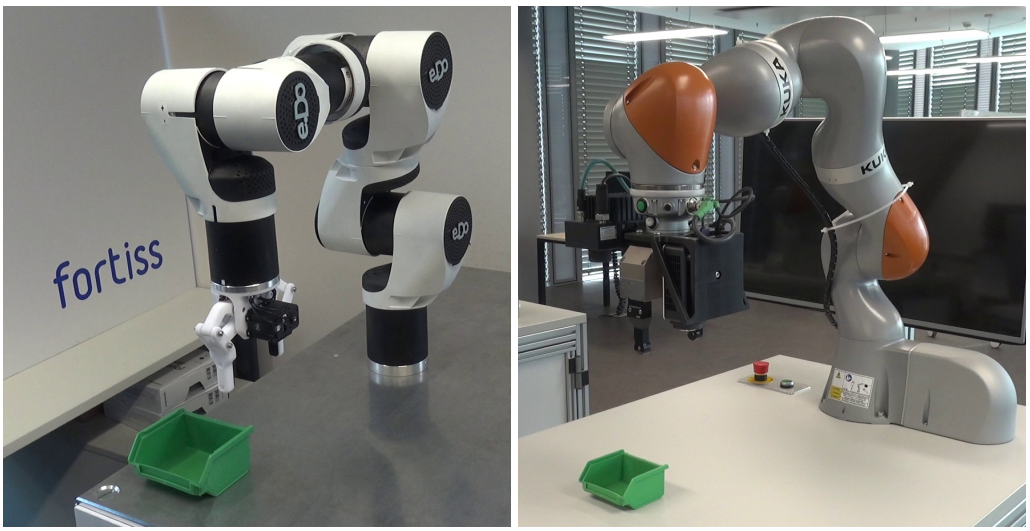
8.3 Skill Composition Evaluation

After evaluating the generic interface for industrial robots and defining an extension of the skill types, I evaluated the applicability of the skill composition concept, including a first version of the skill detector for the composition of a robot and tool skills to provide a Pick-and-Place skill. The theoretical aspect of skill composition is in further detail described in Section 7.3.

A pick and place task can be described by the following sequence of sub-steps:

1. Move to pick approach position
2. Move to pick position
3. Close gripper
4. Move to pick approach position
5. Move to place approach position
6. Move to place position
7. Open gripper
8. Move to place approach position

To compare the skill composition with a directly implemented Pick-and-Place skill, I used two different robot work cells: one work cell consisted of the Comau e.DO robot, which is delivered with a built-in parallel gripper directly attached to the robot (see Figure 8.6a). The second work cell is built up with the KUKA iiwa robot and a Schunk WSG50[6] parallel gripper (see Figure 8.6b).



(a) Comau e.DO robot with built-in parallel gripper

(b) KUKA iiwa robot and a Schunk WSG50 Gripper

Figure 8.6: Pick-and-Place work cells used for evaluating a first version of the Pick-and-Place skill, as shown in the second part of the previously linked video: https://youtu.be/w4V9Boh3ZIo

The major difference between the two work cells is the external and internal gripper. On the Comau e.DO, which has already a proprietary gripper integrated as a seventh joint, the *PickPlaceSkill* is implemented on the robot's OPC UA server. For the external Schunk WSG50 Gripper mounted on the KUKA iiwa, two separate device adapters are necessary, while the Pick-and-Place skill is provided through a Software Component as shown in Figure 8.7. All the device adapters and software components were implemented on a PC, connected to the corresponding hardware component via ethernet.

In contrast to the parameters defined in the previous sections, the implemented *PickPlaceSkill* uses a slightly modified parameter set to explicitly define the position of the object to pick instead of just the object identifier. For this evaluation I did not integrate an object detection as it would overly complicate the demonstrating use case.

On startup, the Pick-and-Place client is using the Skill Detector implementation in combination with the OPC UA discovery services, to find available skills in the network.

[6]https://schunk.com/de_en/gripping-systems/series/wsg/

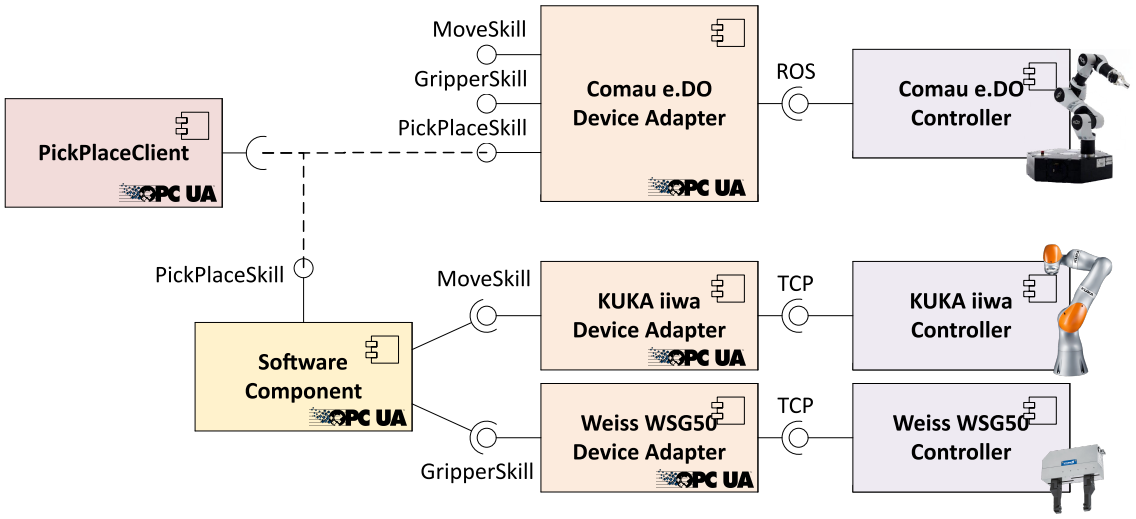


Figure 8.7: Components for the evaluation of skill composition and re-usage by a generic Pick-and-Place client.

Since on both work cells the same Pick-and-Place skill is provided, the client only needs to implement that single interface, independent of the underlying hardware. The object pick position and place position were predefined inside the client implementation. The endpoints for both, the robot and the tool were automatically detected by the Skill Detector.

The execution of the Pick-and-Place skill on the e.DO Robot and the KUKA iiwa is also included in the previously linked video[7].

This evaluation shows that by using my presented generic skill model and skill composition concept, it is possible to abstract low-level skills via software components to reach higher-level functionality. The same skill interface with different parameter sets can be used, independent of the underlying hardware. In this experiment, the additional software component introduced a small delay of approximately 80ms between the end of a robot movement and start of the gripper movement. This delay comes from the communication overhead between the OPC UA client in the software component and the robot and gripper device adapter.

It is important to mention that exchanging the hardware while keeping the same high-level functionality is limited by the robot workspace. If the robot workspace significantly differs, the client cannot use predefined pick and place positions. A solution could be to add more intelligence to the client: using the standardized geometry model structure included in the OPC UA Robotics Companion Specification, a client can infer the robot's workspace and adapt the position correspondingly.

[7] https://youtu.be/w4V9Boh3ZIo

Still, the common Pick-and-Place interface provides a major advantage compared to the direct implementation of proprietary interfaces in the client and reduces the integration effort in more complex setups, especially if the used hardware is constantly changing.

8.4 Plug & Produce System Evaluation

The final evaluation for a real Plug & Produce scenario with a tool changer and multiple tools was performed on the third work cell, composed of a Universal Robots UR5, a Kelvin Tool changer[8], and two tools: the parallel gripper Robotiq 2F-85[9], and the vacuum gripper Schmalz ECBPi[10], as shown in Figure 8.8. A video of the execution can be found online[11] and is described in this section in more detail.

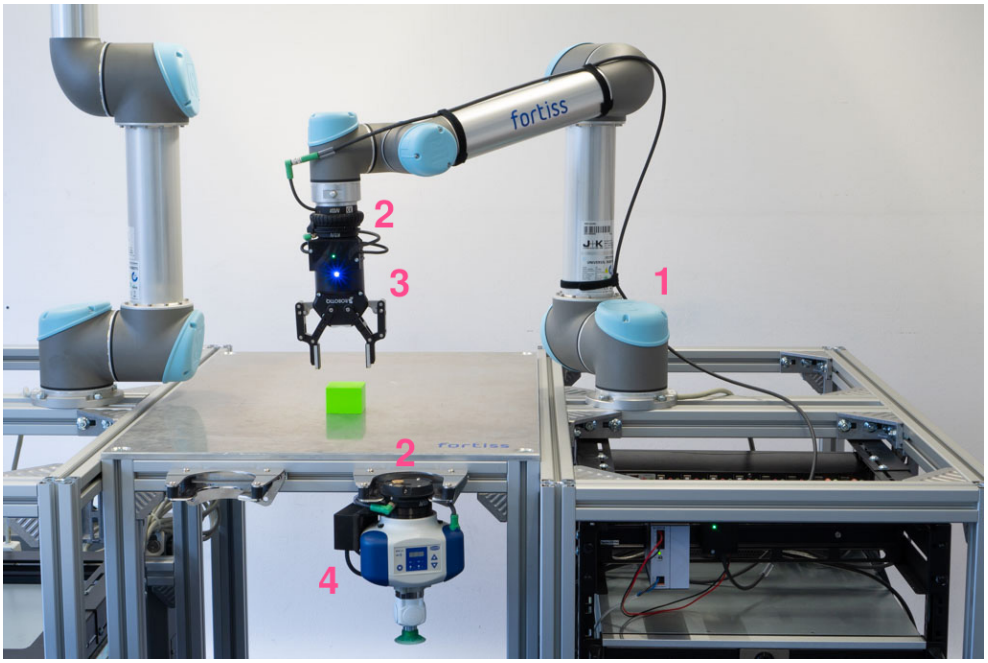


Figure 8.8: Robot work cell used for evaluating the proposed Plug & Play architecture. Composed of the following components: Universal Robots UR5 (1), Kelvin tool changer (2), Robotiq 2-Finger Gripper (3), Schmalz Vacuum Gripper (4), and custom OPC UA adapter controllers for these components

[8] https://www.toolchanger.eu/
[9] https://robotiq.com/products/2f85-140-adaptive-robot-gripper
[10] https://www.schmalz.com/en/vacuum-technology-for-automation/vacuum-components/vacuum-generators/electrical-vacuum-generators/vacuum-generator-ecbpi
[11] https://youtu.be/xiUZnj5qo00

To integrate these components into my Plug & Produce system, I developed various device adapter components responsible for wrapping the proprietary interfaces to my standard OPC UA skill model. The device adapter concept is in more detail described in Chapter 6.

Figure 8.9 shows a simplified overview of the components, which are described bottom-up in the following paragraphs. The source code and OPC UA nodesets that were developed for the whole system and its components are published on GitHub[12]. It is also possible to run the whole system in simulation, as described in the corresponding README file on GitHub. Some excerpts of defined ModelDesign files can be found in the Appendix B.

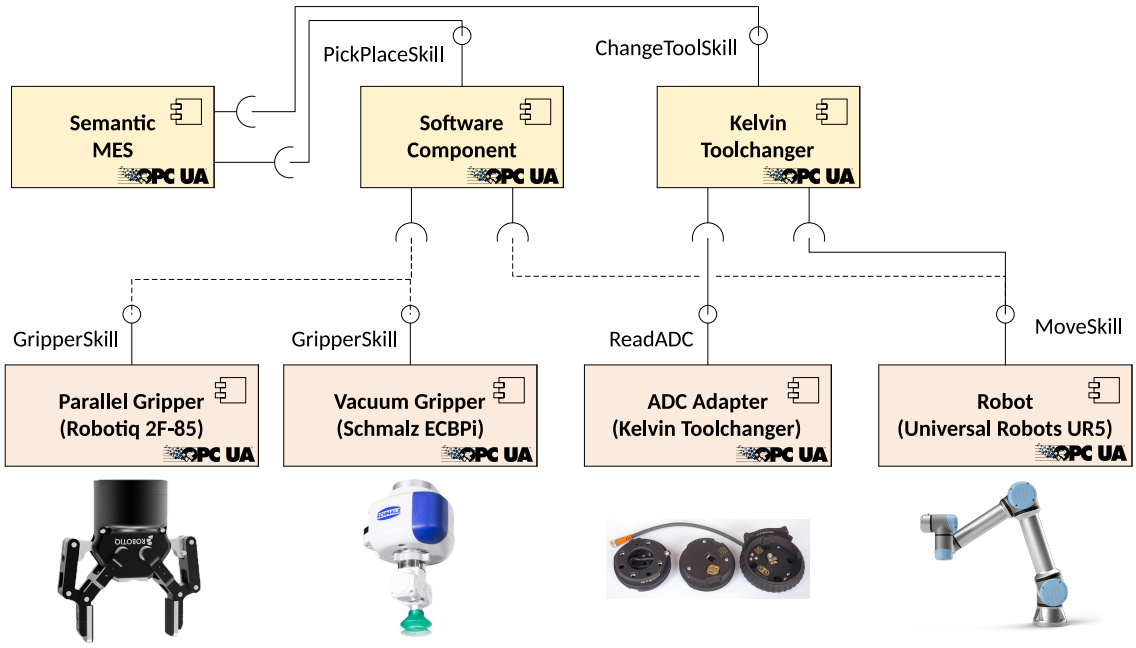


Figure 8.9: Architecture setup for hierarchical skill composition. Hardware components are wrapped with a custom OPC UA server and provide their skills to higher-level components.

All components described in the following paragraphs are implemented in C++ and available in my GitHub repository together with the nodesets for each component. The code uses the open-source open62541 stack[13] for OPC UA, through which I am contributing significant parts to the open-source OPC UA community, and with this easing the integration of my presented concepts into future products. Furthermore, all components implement the LDS-ME feature of OPC UA to be discoverable in the network. The discovery features are in more detail described in Chapter 5.

[12]https://github.com/opcua-skills/plug-and-produce
[13]https://github.com/open62541/open62541

Wi-Fi Microcontroller Boards with OPC UA for close-to-hardware device adapters Since most of the hardware components require specific hardware circuits to adapt the proprietary interfaces, while providing an Ethernet interface, I decided to use a microcontroller with built-in Wi-Fi to deploy the component's device adapter onto it. I first choose the RaspberryPi Zero with Wi-Fi and the Raspbian Operating System. Due to a boot up time of more than 20 seconds, even after various performance tweaks, I abandoned the RaspberryPi and chose to use a more suitable microcontroller with faster startup time for my custom device adapters. A good choice with enough memory (4 MB) to hold the OPC UA information model, and exceedingly small dimensions (18 mm x 32 mm) is the TinyPICO board based on the ESP32 platform by Espressif, using FreeRTOS as its operating system[14]. I developed an OPC UA server, which can be directly flashed onto this microcontroller and provides the basic feature set of OPC UA through the controller's Wi-Fi connection. In the following, I will refer to the TinyPICO board with my customized OPC UA server as TinyUA. The ready-to-use implementation of TinyUA is available on GitHub[15]. It takes around 8 seconds for the microcontroller to power on, join the Wi-Fi network, get the current time via NTP, start the OPC UA application, and announce itself through the OPC UA discovery services, which is significantly faster compared to a RaspberryPi. On the long term all the hardware device adapters described in the following paragraphs should be directly integrated into the tool itself without the need of additional adapter plates and interfaces. This evaluation is a proof-of-concept to show that the proposed system is suitable for Plug & Produce applications.

Kelvin Tool Changer & ADC Adapter The Kelvin Tool Changer[16] is a non-active tool changer: It does not use electricity or pneumatic control to attach or detach tools. The manual locking mechanism can either be operated by a human or autonomously by specific robot movements and the corresponding docking station. Through an analog voltage divider pin, it provides the current state of the tool changer (open, intermediate, locked) and the attached tool ID. Since I also need this data inside the tool changer software component for the skill implementation, I decided to split the implementation into two device adapters. The basic component is a TinyUA server, which uses the analog-digital converter of the TinyPICO microcontroller to provide the current analog voltage encoding the tool changer's state. This voltage value can be read through an OPC UA variable. The second component is a software component which runs on a PC and provides the *AttachToolSkill*, *DetachToolSkill*, and *ChangeToolSkill*. It is connected via the Wi-Fi network to the ADC Adapter to fetch tool changer states. Its startup configuration needs to specify, on which robot the tool changer is mounted. This is achieved by simply passing the robot's OPC UA server endpoint to the software component. The software

[14]https://www.tinypico.com/
[15]https://github.com/Pro/open62541-esp32
[16]https://www.toolchanger.eu/

component indicates its ready state after it receives the corresponding announce message of the robot. This pre-configuration is necessary to indicate which move controller needs to be used to move the mounted tool changer to its docking station. Especially for the case where there are two or more robots in the same work cell, automatic detection cannot be achieved easily.

Vacuum Gripper: Schmalz ECBPi The Schmalz ECBPi Vacuum Gripper uses the IO-Link Protocol as proprietary control interface. To connect the TinyUA board with that interface, I integrated the IO-Link Master Board from TeConcept with the corresponding software stack. For the hardware setup I developed the electrical circuit as shown in Figure 8.10. With this setup, the TinyUA board can implement the *GraspSkill* and *ReleaseSkill* directly on the microcontroller and map control commands to the IO-Link protocol. I constructed and 3D-printed the casing for the microcontroller to mount it onto the gripper, which is shown in Figure 8.12. This setup only requires an external power supply of 24 V and can also be used to adapt any other IO-Link hardware to OPC UA directly on the tool side.

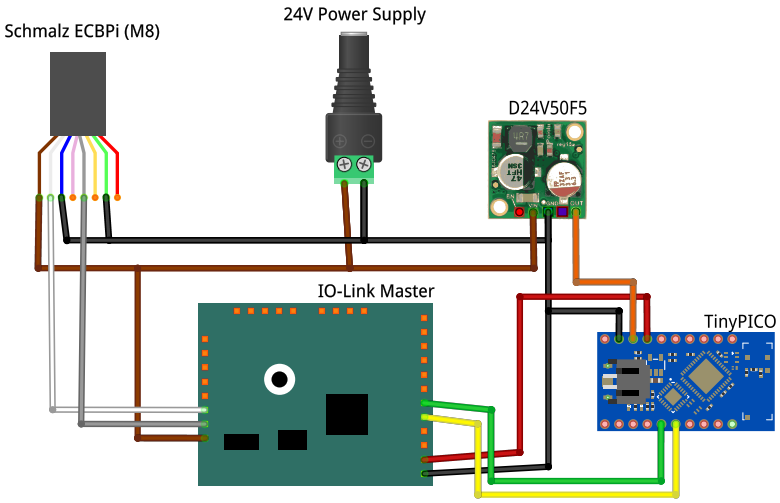


Figure 8.10: Electrical circuit for connecting the TinyPICO microcontroller to the Schmalz ECBPi gripper via its IO-Link protocol.

Parallel Gripper: Robotiq 2F-85 The Robotiq Gripper is shipped with a Modbus RS-485 interface. The serial interface of the TinyUA board can be used together with a MAX485 level shifter chip to connect to the RS-485 interface. For the hardware setup I developed the electrical circuit as shown in Figure 8.11. I am using parts of the Robotics Library [Rickert and Gaschler, 2017] for implementing the Modbus protocol directly on the ESP32 microcontroller. All put together, the TinyUA board provides the *GraspSkill* and *ReleaseSkill* to other components. Same as for the vacuum gripper, I constructed

and 3D-printed the casing for the microcontroller to mount it directly onto the Robotiq gripper, as shown in Figure 8.12.

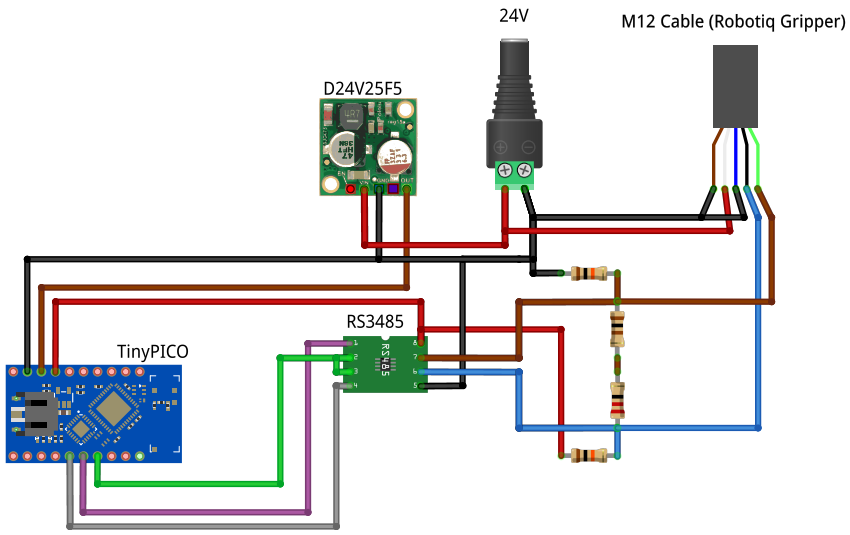


Figure 8.11: Electrical circuit for connecting the TinyPICO microcontroller to Robotiq 2-Finger gripper via its Modbus RS485 protocol.

As already mentioned, these custom developed device adapters for the grippers and the tool changer should in the long run be directly integrated in the tool itself. Since my source code is available on GitHub, manufacturers can use it as a template to implement their own OPC UA servers based on my skill model.

Universal Robots UR5 To implement my OPC UA skill model for the real-time interface of the Universal Robots UR5, I developed a separate C++ application, which combines path planning and robot control abstraction of the Robotics Library [Rickert and Gaschler, 2017] with the open62541 OPC UA stack. This application provides all robot movement skills as depicted in Figure 8.5 through the OPC UA interface to other components. The application is running on a real-time PC to ensure the 8 ms control frequency required by the controller. This PC is directly connected via its own ethernet connection to the robot controller. This is an improved version of the Universal Robots Device Adapter presented in Section 8.1. By contributing to the OPC UA Robotics Companion Specification, I am aiming at directly integrating the proposed skill model into robot controllers without the need of additional external OPC UA servers.

Pick-and-Place Software Component This component is a software component which is not directly connected to any hardware but provides its functionality by composing and orchestrating lower-level skills as described in Section 8.3. Every skill can

Figure 8.12: Custom Wi-Fi OPC UA tool device adapter based on the TinyPICO Microcontroller board with FreeRTOS. The only external connection required is a 24 V power supply.

either be implemented within its own server instance or one server instance can implement multiple skills. The latter case should be preferred where it makes sense, mainly to reduce the load on the system. Every server instance introduces additional overhead. In my experiment I developed one software component implementing the *PickPlaceSkill*. The internal skill-specific implementation receives the ID of the object that should be picked and queries the world model for the object's properties (geometry, current pose). The implementation then searches for a grasp skill on the given tool skill controller. As explained before, the grasp skill contains a description of the tool pose. Combining this information with the object's properties, a grasp planner can determine the optimal grasp pose for the object. My specific implementation of the Pick-and-Place skill just adds a 5 cm offset in Z direction for the approach positions to pick and place the object. A more advanced implementation could include more complex path planning while keeping the same interface. Additional semantic information from lower-level gripper device adapters, such as the maximum gripper span, can be aggregated by this software component and, e.g., used in grasp planners. When all required information is available,

the skill implementation searches in its skill type map for a *CartesianLinearMoveSkill* of the given move controller and triggers the correct lower-level skill sequence.

Since the move controller endpoint is passed as a parameter to the skill, it is trivial for the implementation to choose the right robot. It could be the case that there is more than one robot in the system, with the effect of multiple skills of the same type being available. In this case, a higher-level component would need to intelligently select the correct endpoint based on additional information from the task or world model, e.g., robot position and cartesian workspace. In another experiment together with my colleagues I showed a solution based on SDN to create different network segments and thereby shift the intelligence from the lower level to a central instance [Madiwalar et al., 2019].

The execution and automatic adaptation of the Pick-and-Place skill on two different tools is shown in the previously linked video. Measuring the timing of the system, I calculated the following averaged values for 5 test runs. These values indicate the time between starting the component or connecting the power supply respectively, until the skills of the component are successfully detected by other components.

- Universal Robots UR5: 313 ms
- Pick-and-Place component: 211 ms
- Kelvin Tool changer component: 253 ms
- Robotiq 2F: 8422 ms (8.4 s)
- Schmalz ECBPi: 9635 ms (9.6 s)

As can be seen, automatic component discovery and skill detection takes in general less than 300 milliseconds. The Robotiq 2F and Schmalz ECBPi adapters based on TinyUA take around 9 seconds to boot up and be ready. This higher value stems from summing up various necessary steps: Bootloader (1.8 s), connect to Wi-Fi (2.9 s), initialize time with NTP (1.9 s), start the OPC UA application and announce itself (1.8 s). The initialization of the IO-Link board takes an additional 1.2 seconds on the Schmalz ECBPi TinyUA. These longer setup times can be reduced by improving the prototypical implementation of my TinyUA controller. With corresponding effort, I am estimating that this startup time could be reduced to be below 5 seconds.

Overall, this evaluation proofs that the initial hypothesis of a Plug & Produce system can be achieved, given that more effort is necessary for the initial implementation of component adapters which can be re-used. The following section evaluates the concept in more depth, and the next chapter discusses strengths and weaknesses of my proposed system.

8.5 Summary & Discussion

While a smaller part of this evaluation chapter is looking at the performance of the concept it is not the focus of my thesis. The focus lies on the applicability and extensibility of the presented generic skill concept and the presented Plug & Produce architecture.

The evaluation in previous sections on three different robot work cells with various hardware components shows that a very well-performing generic Plug & Produce system can be achieved using OPC UA in combination with my generic skill model. The evaluation is separated into multiple parts: First, I am evaluating the previously presented skill types on three robots from different manufacturers. This is followed by evaluating the possibility to extend the presented generic skill model with additional types, especially for more complex scenarios where software components are necessary. Since one of the strengths of the skill model is the hierarchical composition of skills, I also evaluate the automatic skill detection and control of lower-level skills through a software component for Pick-and-Place. The evaluation is concluded by a complete system evaluation for a robot work cell with multiple tools and a tool changer. In this last part I also show the applicability of my presented approaches for a real Plug & Produce system with custom (Wi-Fi) device adapters and automatic detection of skills in the system.

Even though all parts of the presented architecture are released as open-source on GitHub and should therefore ease the adaption of the proposed system, integration of custom devices still implies significant effort. It is necessary to adapt proprietary protocols and to agree on a common standardized skill model for wide-range adoption of hardware components. As an active member of various joint working groups of OPC UA and VDMA, I am supporting the standardization process of a generic skill model in OPC UA. The work leading to this thesis intends to contribute to this process and bring it closer to its goal. The effort of implementing device adapters for specific hardware quickly pays off, as they significantly reduce the required reconfiguration time especially for small lot production.

My experiments show that component integration based on a Wi-Fi connection does not necessarily slow down the system. Even robot tools can be efficiently controlled through stable Wi-Fi connections. The impact of unstable network connection still needs to be investigated, though.

In general, it can be seen that for more flexible or generic systems, the flexibility leads to a higher performance impact. Generic robot movement skills move the robot's tool center point from any valid pose to another, but a specifically programmed trajectory still can be more efficient. Therefore, an important factor to decide on how flexible a manufacturing system should be is the required level of performance and optimization.

9 Conclusion

In this last chapter, I am answering the initial research question and give a summary on the used key methodologies and techniques which I used to provide an answer to the question. I also take a closer look into the strengths and weaknesses of my presented approach. This is followed by showing the relevance of my work not only for the industrial automation domain, but all domains where different hardware components need to be controlled through a generic interface. This thesis concludes with a take home message and possible future work.

9.1 Thesis Research Question Revisited

The main research question of this thesis is, as presented in Section 1.3, if it is possible to build up a Plug & Produce industrial robot cell, where system components and robot tools can be easily exchanged without the need of reprogramming or adapting higher-level control components.

In current state-of-the-art robot work cells, components, such as robots or grippers, typically come with different proprietary interfaces, and therefore these components cannot easily be exchanged with other components providing the same functionality.

I subdivided the main question into multiple subsequent questions, i.e., which communication protocol is suitable for a Plug & Produce setup, how components and their functionality can be automatically discovered, how the provided component services need to be described, and what performance such a generic Plug & Produce system can achieve.

9.2 Key Methodologies and Techniques

An answer to the first question, to which communication protocol is suitable for a Plug & Produce setup, I developed a testing framework which can be used to evaluate the

performance of a variety of communication protocols. Here, I evaluate the round-trip time, throughput, CPU load and other key performance indices on the implementation of following commonly used protocols: ROS, DDS, MQTT, and OPC UA. The evaluation shows that OPC UA is very suitable and delivers an exceptionally good performance. Its main strength is the semantically described information model and standardized protocol.

To achieve automatic component discovery, I extend the open-source OPC UA implementation open62541 (C) and Eclipse Milo (Java) to support the discovery features of OPC UA and evaluate both implementations against the official reference implementation. Based on these discovery services, I develop a skill detector class in C++, which can be easily integrated into the developed components and is used to find all available skills in the network.

A skill is a specific functionality which is offered by a hardware or software component. One of the main contributions of this thesis is the development of a generic skill model which can be used to access skills of components through a well-defined interface. This skill model is directly modeled in OPC UA as set of Companion Specifications and uses the concept of a finite state machine to represent the current skill state. The concept of the presented model is not specifically bound to OPC UA, as it can also be implemented by other middlewares which provide a semantic information model. Parametrization of a skill execution is achieved through the creation of specific skill subtypes for specific functionalities. For example, I create specific skill types for an industrial robot interface, which allow controlling any 6-DOF robot thorough the same interface. Even robots with more than 6-DOF can be controlled through this interfaces as long as the skill implementation handles the additional null-space configurations correspondingly. Using skill composition, new more complex skills with higher-level functionality can be formed by re-using existing skills.

My other significant contribution in this thesis is the generic Plug & Produce architecture based on the generic skill model. The combination of all previously mentioned concepts results in an architecture which supports on-the-fly exchange of system components, and automatic detection of new skills. A generic skill client is used to connect to, and control such modeled skills. To be able to use proprietary devices in combination with the presented skill model, the concept of device adapters is introduced.

These concepts are evaluated on different robot work cells, first by evaluating the generic skill concept to control different industrial robots (Universal Robots UR5, Comau e.DO, KUKA iiwa). This evaluation is followed by composing a Pick & Place skill based on generic robot and gripper skills and evaluating it on two different robot/tool combinations without the need of changing the client implementation. Finally, the overall Plug & Produce architecture is evaluated on a robot work cell consisting of a robot,

a tool changer and two different tools, all using custom developed device adapters to implement the generic skill interface for the proprietary interfaces.

9.3 Strengths and Weaknesses of the Presented Approach

The results of the experiments on different robot work cells show that my presented generic skill model and Plug & Produce architecture provides a particularly good performance for easy component exchange in the context of Industry 4.0. The detection of components and their skills takes less than half a second and even performs well when using Wi-Fi connections. Since OPC UA provides a standardized protocol, feature-rich semantic annotation of information, and additional services, such as the discovery service set, my presented architecture only depends on OPC UA as its single base technology, but can also be implemented on other middlewares providing the same set of features. With the defined skill model, skills of any Industry 4.0 component can be abstracted and controlled through the generic interface. This allows to replace used hardware without any reprogramming of the control applications. The list of components is not limited to hardware abstraction, as the generic skill interface can also be used for the abstraction of algorithms in software components. Using skill composition, lower-level skills can further be abstracted and new functionality can be offered. Since skill composition is based on lower-level standardized skill interfaces, it is even possible to introduce the concept of skill stores where manufacturers and system integrators can offer or sell additional functionality, which can simply be downloaded and used to add an additional range of features to a work cell.

A major hurdle, which currently prevents the direct application of my model is the fact, that nearly every device comes with its own protocol specification. Therefore, custom device adapters must be developed (as shown in Chapter 8) which implement my generic skill model and adapt these proprietary protocols. The effort of implementing work-around adapters for specific hardware quickly pays off, as they significantly reduce the required reconfiguration time especially for small lot production. The current momentum for Industry 4.0 is remarkably high, and there are many activities around standardized models and Companion Specifications for different types of hardware. By actively working in those standardization groups I am contributing to the success and fast adoption of my presented model. However, it will take even more time until a majority of components support a common Plug & Produce standard.

The requirement to base all skill implementations on a specific skill type assumes that different manufacturers agree on these skill types, and especially the specific set of pa-

rameters associated to a skill type. This might be a difficult task, as discussions in current OPC UA joint working groups show. Nonetheless, it is necessary to find a common ground on which future improvements can be built. My skill model is very flexible and allows step-by-step extension with specific skill types.

9.4 Relevance to the Industrial Automation Domain

The RAMI 4.0, VDMA Guideline for the Introduction of OPC UA (see Section 2.2), and the Multi-Annual-Roadmap (MAR) [SPARC, 2020] are highly relevant publications in the industrial automation domain, and clearly underline the importance of efficient integration of system components as one of the basic technologies in the current Industry 4.0 movement. This can only be achieved through a standard interface, and a commonly agreed communication protocol.

Both, RAMI 4.0 and VDMA strongly recommend using the OPC UA framework for this component integration. Since my thesis and my presented solutions are based on OPC UA and directly solve the raised question by defining a generic skill interface to access the component services, the results of my thesis are highly relevant not only for the industrial automation domain, but for any other domain dealing with different hardware components connected through a network. For example, in the Internet of Things domain, my skill model can also be applied to adapt proprietary interfaces.

The difficult task to convince device manufacturers to agree to a standard for specific skill types can be simplified by reusing already existing standards: Using the OPC UA dictionary reference, it is possible to link external entities, e.g., the corresponding eCl@ss or VDI 2860 definition, to a specific skill type and therefore assign a semantic meaning to it which more clients may understand. Still, the specific parameters need to be agreed upon, or automatically inferred by interpreting its semantic annotations.

In the future, skill implementations that comply with standardized skill types can be distributed through open software libraries or commercial app stores for manufacturing skills (skill stores). The German Industry 4.0 Index 2019 [Staufen AG, 2019] shows that especially small- and medium-sized enterprises will benefit from such a generic Plug & Play system.

9.5 Take-home Message & Future work

As shown in this thesis, it is possible to achieve a particularly good performing Plug & Produce system based on well-defined skill interfaces and using generic software components which control lower-level skills. Skill composition allows forming new functionalities based on re-using offered skills in the system.

One should keep in mind, that the cost of flexibility and configurability is performance. In general, for more flexible or generic systems, a higher performance impact can be expected. Executing a sequence of generic skills introduces communication and synchronization overhead, while a dedicated low-level implementation achieving the same task can achieve higher performance. Therefore, an important factor for deciding on how flexible a manufacturing system should be is the required level of performance and optimization. Yet, more flexible systems, as the one presented in this thesis may result in a small performance impact, but significantly reduce the setup and configuration time.

I will continue developing additional device adapters to support more hardware, e.g., for automatic robot screwdrivers and force-enabled assembly skills such as peg-in-hole. The skill model can be extended with additional queuing mechanisms, e.g., to automatically execute skills sequentially or in parallel. Instances of a parametrized skill can be added to such a queue and automatically started by the queue controller. This will then also support, for example, blending movements of robot arms, or more efficient execution of multiple skills. The presented semantic skill model also needs to be extended to include a semantic description of device properties in a generic way, e.g., available gripper span, and possible robot payload, which allows more complex composite skills based on device capabilities. Since all my developed source code is available on GitHub, new research can be based on my results.

It is my personal goal to continue collaborating with various standardization groups to integrate the results of this thesis into newly released OPC UA Companion Specifications, and to collaborate with manufacturers to support their development of new Plug & Produce ready devices.

A OPC UA Address Space, Information Model, & Modelling Notation

One of the strengths of OPC UA is its semantic information model. The information model is the structural definition of the nodes and references, while the address space is the concrete implementation of an information model and its node instances inside an OPC UA server.

The information model can be graphically visualized using the official OPC UA modelling notation.

This Appendix gives an overview of the information model concepts and the official OPC UA modelling notation.

A.1 OPC UA Address Space & Information Model

An information model defines the nodes and their structure provided in the server's address space. Similar to object-oriented programming, an information model defines types which can be extended and instantiated. In addition, these types can be semantically enriched by using specific reference types to other nodes. Since the address space is basically a combination of triples (source node, reference, target node) forming a directed graph, it is easily possible to transfer the knowledge to a typical graph-based triple store as shown in [Perzylo, Profanter, et al., 2019].

The OPC Foundation defines a base information model in the official OPC UA specification Part 5[1]. It defines necessary base object types, reference types, variable nodes and more, as defined in the OPC UA address space and further explained in this sec-

[1] https://reference.opcfoundation.org/v104/Core/docs/Part5/

tion. Based on this section I also wrote an online documentation which is publicly available[2].

A.1.1 The Basics

An OPC UA information model is a collection of nodes and their references. Additionally, the information model itself defines the node id within that information model. The OPC UA address space defines the following base node types:

- **VariableType**: Type definition for a variable
- **Variable**: Instance of a variable type
- **ObjectType**: Type definition for an object node
- **Object**: Instance of an object type
- **Method**: Specific node type indicating a callable method
- **ReferenceType**: Type definition for a reference
- **Reference**: Instance of a reference type. A reference connects two nodes in a directed graph.
- **DataType**: Definition of data types used for the value of a variable.
- **View**: Node to collect a subset of nodes in the address space viewable by a client.

A more detailed description on these node types is given in the OPC UA Specification Part 3[3].

Every node is identified by a unique node id inside its specific nodeset or namespace. The nodeset is identified by a unique Namespace URI. A node id is only complete if the id itself, and the namespace URI are indicated.

A.1.2 Namespace URI and Namespace Index

Every information model must have its own unique identification URI. For the base OPC UA specification, this URI is `http://opcfoundation.org/UA/`.

Additional Companion Specifications or custom specifications define their own namespace URI, e.g., `http://opcfoundation.org/UA/DI/` for the DI specification. These namespace URIs do not necessarily have to be accesible URLs, as they only identify the namespace. Still, it is recommended to provide at least the corresponding NodeSet2 XML definitions under the given URL.

[2] https://opcua.rocks/address-space/
[3] https://reference.opcfoundation.org/v104/Core/docs/Part3/

Inside an OPC UA server, the namespace is identified by a namespace index. Index 0 is always the base nodeset, and index 1 is for any instances of nodes which do not belong to a specific nodeset. Depending on the use-case, a server can load additional nodesets into its address space. These nodesets typically start at index 2.

After connecting, a client should always read the namespace array of the server to determine which namespaces are currently loaded in the server, and to be able to match the namespace URI to a namespace index. It is not guaranteed that a server keeps the same namespace index after restarting it. Therefore, it is crucial to query the namespace array after every new connection by a client.

A.1.3 Node ID

A node id uniquely identifies a node inside its node set. This node id and namespace index combination is only valid for this specific server instance. As explained in previous section, the namespace index may change during different server instances.

Therefore, the node id always exists of two parts, the namespace index, and the id part. The id can be one of the following types:

- Numeric
- String
- GUID (Global Unique Identifier)
- Byte String

Most used are the numeric node id, e.g., for automatic generation of ids, and the string node id for human-readable node ids. Every information model defines the node ids for all the nodes which are included in the model definition.

A.1.4 Variables: Properties and Data Variables

A variable is used to represent a value in the OPC UA address space model. OPC UA Part 3 defines two types of Variables: Properties and Data Variables.

Properties are characteristics of Objects, DataVariables and other nodes defined by the server.
One major difference to DataVariables is, that Properties do not allow to have child-properties. In general, a Property should not have any child nodes and should be a leaf node.

Also, a Property BrowseName shall be unique in the context of a Node in order to uniquely identify it.

Data Variables represent specific value collections inside an Object. For a file this could be the binary string, while a property is used to provide modification times.
Compared to Properties, DataVariables may also have child DataVariables to form a more complex structure.

A.1.5 Companion Specifications

The OPC UA base information model can be extended through Companion Specifications (see Section 4.2.1). A Companion Specification basically extends the core types with custom object types or defines new variable types.

Some example Companion Specifications are: OPC UA for Devices (DI), OPC UA for Robotics (ROB), or OPC UA for PLCopen.

A full list of officially released specifications can be obtained on the OPC Foundation webpage[4].

An OPC UA server can implement any combination of given Companion Specifications to provide the corresponding hardware data. The loaded Companion Specifications are listed inside the server's namespace array variable.

In addition to the officially released Companion Specifications, anyone can create his own Companion Specification which extends the basic OPC UA node set with custom types and definitions.

A.2 Graphical Modeling Notation

To visualize the information model with all its specifics, typical UML models do not provide enough features. Therefore, the OPC Foundation developed a specific OPC UA Modeling Notation. It is similar to the UML notation with some more extensions shown in Figure A.1.

This graphical notation can be used to fully visualize the information represented in an information model and is easier to understand compared to XML files.

[4]https://opcfoundation.org/developer-tools/specifications-opc-ua-information-models

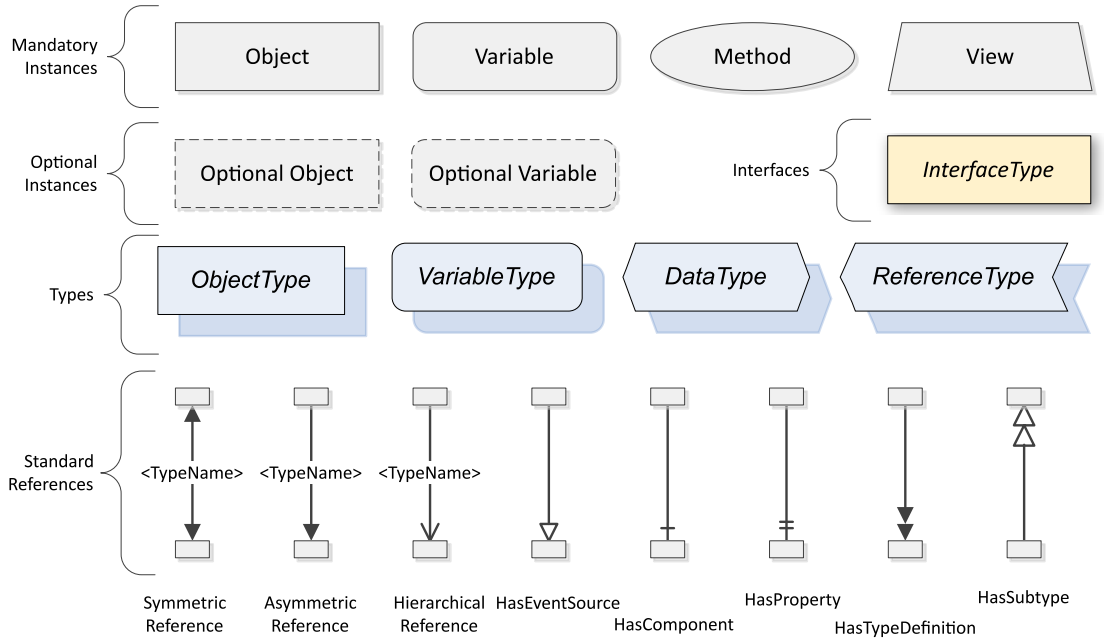


Figure A.1: Extended OPC UA modelling notation used for visualizing the information model throughout this thesis.

There are three main concepts: Node Types, Node Instances, and Node References.

Node types are depicted as blue boxes with a hard shadow. The four different types (ObjectType, VariableType, DataType, ReferenceType) all have their own separate shape. A specific kind of node type is the *InterfaceType*. It is shown as a yellow box with faded shadow to indicate its special meaning.

Node instances use the same shape as their corresponding type node, but are depicted without a shadow and with gray or green background. An object type node may have mandatory and optional child nodes. Mandatory child nodes must be instantiated by the object instance and are marked with a solid border line. Optional child nodes can be instantiated by an object instance, depending on the use-case, and have a dashed border line.

A reference between two nodes is generally shown with a directed arrow connection. Since the OPC UA specification defines a set of commonly used standard references, they have separate arrow types. Custom reference types are typically subtyping one of these reference types. These connecting lines can have optional labels to indicate the reference subtype.

In Figure 6.2 on page 77 and in Figure 6.3 on page 80 I show some examples how this graphical notation looks on a specific example.

B OPC UA ModelDesign Excerpts

This chapter contains some excerpts of my custom OPC UA Companion Specifications as ModelDesign files. Section 6.4.1 describes how these ModelDesign files are used to initialize the address space of an OPC UA server. The complete set of ModelDesign files is available in my GitHub repository[1].

B.1 ModelDesign Excerpt: Skill Definition

Excerpt of the *fortiss DI* ModelDesign XML file[2] defining the SkillType, which is a subtype of the ProgramStateMachineType, and its parameters, as well as transition methods. This excerpt also includes the definition of the ISkillControllerType interface, and specific skill types such as the GraspGripperSkillType and ReleaseGripperSkillType.

```
<?xml version="1.0" encoding="utf-8"?>
<ModelDesign xmlns="http://opcfoundation.org/UA/ModelDesign.xsd">
  <!-- Additional metadata. Full file see https://github.com/opcua-skills/skills-common/blob/master/deps/robotics_cs/deps/
       device/fortissDeviceModel.xml -->
  <ObjectType SymbolicName="DEVICE:SkillType" BaseType="OpcUa:ProgramStateMachineType" IsAbstract="true">
    <Description>A skill type</Description>
    <Children>
      <Property SymbolicName="DEVICE:Name" DataType="OpcUa:String" ValueRank="Scalar" ModellingRule="None">
        <Description>Name of the skill</Description>
      </Property>
      <Property SymbolicName="OpcUa:MaxInstanceCount" DataType="OpcUa:UInt32" ValueRank="Scalar"
                ModellingRule="Mandatory"/>
      <Object SymbolicName="OpcUa:FinalResultData" TypeDefinition="OpcUa:BaseObjectType"
              ModellingRule="Optional"/>
      <Method SymbolicName="OpcUa:Halt" ModellingRule="Mandatory"/>
      <Method SymbolicName="OpcUa:Reset" ModellingRule="Mandatory"/>
      <Method SymbolicName="OpcUa:Resume" ModellingRule="Mandatory"/>
      <Method SymbolicName="OpcUa:Suspend" ModellingRule="Mandatory"/>
      <Method SymbolicName="OpcUa:Start" ModellingRule="Mandatory"/>
    </Children>
  </ObjectType>

  <ObjectType SymbolicName="DEVICE:GripperSkillType" BaseType="DEVICE:SkillType" IsAbstract="true">
    <Description>A gripper skill type</Description>
  </ObjectType>
```

[1] https://github.com/opcua-skills/plug-and-produce

[2] https://github.com/opcua-skills/skills-common/blob/master/deps/robotics_cs/deps/device/fortissDeviceModel.xml

```
25
26  <ObjectType SymbolicName="DEVICE:GraspGripperSkillType" BaseType="DEVICE:GripperSkillType" IsAbstract="false">
27    <BrowseName>GraspGripperSkill</BrowseName>
28    <Description>Close the gripper to its minimum width</Description>
29    <Children>
30      <Object SymbolicName="DI:ParameterSet" ModellingRule="Optional">
31        <Children>
32          <Variable SymbolicName="DEVICE:Force" TypeDefinition="OpcUa:AnalogItemType" DataType="OpcUa:Double"
33                    ModellingRule="Optional" ValueRank="Scalar" AccessLevel="ReadWrite">
34            <Description>The gripper force</Description>
35          </Variable>
36        </Children>
37      </Object>
38    </Children>
39  </ObjectType>
40
41  <ObjectType SymbolicName="DEVICE:ReleaseGripperSkillType" BaseType="DEVICE:GripperSkillType" IsAbstract="false">
42    <BrowseName>ReleaseGripperSkill</BrowseName>
43    <Description>Open the gripper to its maximum width</Description>
44    <Children>
45      <Object SymbolicName="DI:ParameterSet" ModellingRule="Optional">
46        <Children>
47          <Variable SymbolicName="DEVICE:Force" TypeDefinition="OpcUa:AnalogItemType" DataType="OpcUa:Double"
48                    ModellingRule="Optional" ValueRank="Scalar" AccessLevel="ReadWrite">
49            <Description>The gripper force</Description>
50          </Variable>
51        </Children>
52      </Object>
53    </Children>
54  </ObjectType>
55
56  <ObjectType SymbolicName="DEVICE:ISkillControllerType" BaseType="OpcUa:BaseInterfaceType" IsAbstract="true"
57              SupportsEvents="true">
58    <Description>The interface definition for a skill controller type. Represents an object which contains skill
59      instances.
60    </Description>
61    <Children>
62      <Property SymbolicName="DEVICE:Name" DataType="OpcUa:LocalizedText" ValueRank="Scalar"
63                ModellingRule="Optional"/>
64      <Object SymbolicName="DEVICE:Skills" TypeDefinition="OpcUa:BaseObjectType" ModellingRule="Mandatory">
65        <Description>Contains the skills of the Component</Description>
66        <Children>
67          <Object SymbolicName="DEVICE:Skill__No_" TypeDefinition="DEVICE:SkillType"
68                  ModellingRule="OptionalPlaceholder">
69            <BrowseName>Skill_&lt;No&gt;</BrowseName>
70          </Object>
71        </Children>
72      </Object>
73    </Children>
74  </ObjectType>
75 </ModelDesign>
```

Listing B.1: Excerpt of the *fortiss DI* ModelDesign XML file defining the SkillType and its parameters, and specific gripper skill types.

B.2 ModelDesign Excerpt: Robot Skill Types

Excerpt of the *fortiss Robotics* ModelDesign XML file[3] defining the hierarchical structure and parameters of the LinearMoveSkillType, which is based on the MoveSkillType, and the interface definition of the ICartesianMoveSkillParameterType.

```
<?xml version="1.0" encoding="utf-8"?>
<ModelDesign xmlns="http://opcfoundation.org/UA/ModelDesign.xsd">
  <!-- Additional metadata. Full file see https://github.com/opcua-skills/skills-common/blob/master/deps/robotics_cs/
       fortissRoboticsModel.xml -->

  <ObjectType SymbolicName="FOR_ROB:MoveSkillType" BaseType="DEVICE:SkillType" IsAbstract="true">
    <Description>Move a robot using a specific tool frame</Description>
    <Children>
      <Object SymbolicName="DI:ParameterSet" ModellingRule="Mandatory">
        <Children>
          <Variable SymbolicName="FOR_ROB:ToolFrame" TypeDefinition="OpcUa:BaseDataVariableType" DataType="OpcUa:
              String"
                    ModellingRule="Mandatory" ValueRank="Scalar" AccessLevel="ReadWrite">
            <Description>The name of the tool frame to be used for the motion</Description>
          </Variable>
        </Children>
      </Object>
    </Children>
  </ObjectType>


  <ObjectType SymbolicName="FOR_ROB:LinearMoveSkillType" BaseType="FOR_ROB:MoveSkillType" IsAbstract="true">
    <Description>Move a robot using a specific tool in a linear motion</Description>
    <Children>
      <Object SymbolicName="DI:ParameterSet" ModellingRule="Mandatory">
        <Children>
          <Variable SymbolicName="FOR_ROB:MaxAcceleration" TypeDefinition="OpcUa:AnalogUnitType"
                    DataType="OpcUa:Double" ModellingRule="Mandatory" ValueRank="Array"
                    ArrayDimensions="6" AccessLevel="ReadWrite">
            <Description>Maximum acceleration of the robot should move. First three parameters are for x,y,z in m/s^2.
              The next three for orientation in rad/s^2
            </Description>
          </Variable>
          <Variable SymbolicName="FOR_ROB:MaxVelocity" TypeDefinition="OpcUa:AnalogUnitType"
                    DataType="OpcUa:Double" ModellingRule="Mandatory" ValueRank="Array"
                    ArrayDimensions="6" AccessLevel="ReadWrite">
            <Description>Maximum velocity of the robot should move. First three parameters are for x,y,z in m/s. The
              next three for orientation in rad/s
            </Description>
          </Variable>
        </Children>
      </Object>
    </Children>
  </ObjectType>

  <!-- Additional ObjectType definitions, e.g., PtpMoveSkillType, IJointMoveSkillParameterType, JointLinearMoveSkillType, ...
       -->

  <ObjectType SymbolicName="FOR_ROB:ICartesianMoveSkillParameterType" BaseType="OpcUa:BaseInterfaceType"
              IsAbstract="true"
              SupportsEvents="true">
    <Description>The interface definition of a cartesian move skill.
```

[3] https://github.com/opcua-skills/skills-common/blob/master/deps/robotics_cs/fortissRoboticsModel.xml

```
    </Description>
    <Children>
      <Object SymbolicName="OpcUa:FinalResultData" ModellingRule="Mandatory">
        <Children>
          <Variable SymbolicName="FOR_ROB:ForcesExceeded" TypeDefinition="OpcUa:ThreeDVectorType"
                    ModellingRule="Mandatory" ValueRank="Scalar" AccessLevel="Read">
            <Description>The amount by which the force limits were exceeded</Description>
          </Variable>
          <Variable SymbolicName="FOR_ROB:ForcesMax" TypeDefinition="OpcUa:ThreeDVectorType"
                    ModellingRule="Mandatory" ValueRank="Scalar" AccessLevel="Read">
            <Description>Maximum force measured during execution of the skill</Description>
          </Variable>
        </Children>
      </Object>
      <Object SymbolicName="DI:ParameterSet" ModellingRule="Mandatory">
        <Children>
          <Variable SymbolicName="FOR_ROB:TargetPosition" TypeDefinition="OpcUa:ThreeDFrameType"
                    DataType="OpcUa:ThreeDFrame" ModellingRule="Mandatory" ValueRank="Scalar"
                    AccessLevel="ReadWrite">
            <Description>Absolute goal position in cartesian space</Description>
          </Variable>
          <Variable SymbolicName="FOR_ROB:AxisBounds" TypeDefinition="OpcUa:BaseDataVariableType"
                    DataType="OpcUa:Range" ModellingRule="Mandatory" ValueRank="Array"
                    AccessLevel="ReadWrite">
            <Description>Define a range within which the joints should end up in. Used to limit the solutions for the
              inverse kinematics calculation.
            </Description>
          </Variable>
        </Children>
      </Object>
    </Children>
  </ObjectType>

  <ObjectType SymbolicName="FOR_ROB:FortissMotionDeviceType" BaseType="ROB:MotionDeviceType" IsAbstract="false">
    <!-- Additional parameters for MotionDevice, e.g., flange load. -->
  </ObjectType>

</ModelDesign>
```

Listing B.2: Excerpt of the *fortiss Robotics* ModelDesign XML file defining the basic robot move skills and their parameters.

B.3 ModelDesign Excerpt: Skill Instantiation Example for a Universal Robot

Excerpt of the *fortiss UR* ModelDesign XML file[4] which is creating a specific instance of a MotionDeviceSystemType, implementing the ISkillControllerType interface, and instantiating different robot move skills inside the Skills node on the example of a Universal Robots UR5.

[4] https://github.com/opcua-skills/plug-and-produce/blob/master/robot/universal-robots/opcua/universalRobotsModel.xml

```xml
<?xml version="1.0" encoding="utf-8"?>
<ModelDesign xmlns="http://opcfoundation.org/UA/ModelDesign.xsd">
  <!-- Additional metadata. Full file see https://github.com/opcua-skills/plug-and-produce/blob/master/robot/universal-robots/opcua/universalRobotsModel.xml -->

  <Object SymbolicName="ROB_UR:UrMotionSystem" TypeDefinition="ROB:MotionDeviceSystemType">
    <Description>The UR Robot</Description>
    <References>
      <Reference IsInverse="true">
        <ReferenceType>ua:Organizes</ReferenceType>
        <TargetId>DI:DeviceSet</TargetId>
      </Reference>
    </References>
    <Children>
      <Object SymbolicName="ROB:MotionDevices" TypeDefinition="ua:BaseObjectType" ModellingRule="Mandatory">
        <Children>
          <Object SymbolicName="ROB_UR:UrRobot" TypeDefinition="FOR_ROB:FortissMotionDeviceType"
                  ModellingRule="Mandatory">
            <BrowseName>UR Robot</BrowseName>
            <Children>
              <Object SymbolicName="ROB:Axes" TypeDefinition="ua:BaseObjectType" ModellingRule="Mandatory">
                <Description>Contains the axis set of the motion device.</Description>
                <Children>
                  <Object SymbolicName="ROB:Axis_1" TypeDefinition="ROB:AxisType" ModellingRule="Mandatory">
                    <Description>The bottom-most axis</Description>
                    <Children>
                      <Property SymbolicName="ROB:Name" DataType="ua:String" ValueRank="Scalar"
                                ModellingRule="Optional">
                        <Description>Joint0</Description>
                      </Property>
                      <Property SymbolicName="ROB:MotionProfile" DataType="ROB:AxisMotionProfileEnumeration"
                                ValueRank="Scalar" ModellingRule="Mandatory">
                        <Description>The kind of axis motion as defined with the AxisMotionProfileEnumeration.
                        </Description>
                        <DefaultValue>
                          <!-- 1 = ROTARY -->
                          <uax:UInt16>1</uax:UInt16>
                        </DefaultValue>
                      </Property>
                      <Object SymbolicName="DI:ParameterSet" ModellingRule="Mandatory">
                        <Children>
                          <Variable SymbolicName="ROB:ActualPosition" TypeDefinition="ua:AnalogUnitType"
                                    DataType="ua:Double" ModellingRule="Mandatory">
                            <Description>The axis position inclusive Unit and RangeOfMotion.</Description>
                          </Variable>
                          <Variable SymbolicName="ROB:ActualSpeed" TypeDefinition="ua:AnalogItemType"
                                    DataType="ua:Double" ModellingRule="Mandatory">
                            <Description>The axis speed on load side (after gear/spindle) inclusive Unit.</Description>
                          </Variable>
                          <Variable SymbolicName="ROB_UR:ActualCurrent" TypeDefinition="ua:BaseDataVariableType"
                                    DataType="ua:Double" ModellingRule="Mandatory">
                            <Description>Actual joint current</Description>
                          </Variable>
                          <Variable SymbolicName="ROB_UR:Temperature" TypeDefinition="ua:BaseDataVariableType"
                                    DataType="ua:Double" ModellingRule="Mandatory">
                            <Description>Joint Temperature</Description>
                          </Variable>
                        </Children>
                      </Object>
                    </Children>
                  </Object>
                  <!-- Additional children defining remaining axes -->
                  <Object SymbolicName="ROB:Axis_2" TypeDefinition="ROB:AxisType" ModellingRule="Mandatory"/>
                  <Object SymbolicName="ROB:Axis_3" TypeDefinition="ROB:AxisType" ModellingRule="Mandatory"/>
                  <Object SymbolicName="ROB:Axis_4" TypeDefinition="ROB:AxisType" ModellingRule="Mandatory"/>
```

```
                    <Object SymbolicName="ROB:Axis_5" TypeDefinition="ROB:AxisType" ModellingRule="Mandatory"/>
                    <Object SymbolicName="ROB:Axis_6" TypeDefinition="ROB:AxisType" ModellingRule="Mandatory"/>
                  </Children>
                </Object>
              </Children>
            </Object>
          </Children>
        </Object>
        <Object SymbolicName="ROB:Controllers" TypeDefinition="ua:BaseObjectType" ModellingRule="Mandatory">
          <Children>
            <Object SymbolicName="ROB_UR:UrController" TypeDefinition="ROB:ControllerType" ModellingRule="Mandatory">
              <BrowseName>UR Controller</BrowseName>
              <Children>
                <Object SymbolicName="DEVICE:Skills" TypeDefinition="ua:BaseObjectType" ModellingRule="Mandatory">
                  <Children>
                    <Object SymbolicName="ROB_UR:JointLinearMoveSkill" TypeDefinition="FOR_ROB:
                        JointLinearMoveSkillType"
                          ModellingRule="Mandatory">
                    </Object>
                    <Object SymbolicName="ROB_UR:JointPtpMoveSkill" TypeDefinition="FOR_ROB:JointPtpMoveSkillType"
                          ModellingRule="Mandatory">
                    </Object>
                    <Object SymbolicName="ROB_UR:CartesianLinearMoveSkill"
                          TypeDefinition="FOR_ROB:CartesianLinearMoveSkillType" ModellingRule="Mandatory">
                    </Object>
                    <Object SymbolicName="ROB_UR:CartesianPtpMoveSkill" TypeDefinition="FOR_ROB:
                        CartesianPtpMoveSkillType"
                          ModellingRule="Mandatory">
                    </Object>
                  </Children>
                </Object>
              </Children>
              <References>
                <Reference IsInverse="false">
                  <ReferenceType>ua:HasInterface</ReferenceType>
                  <TargetId>DEVICE:ISkillControllerType</TargetId>
                </Reference>
              </References>
            </Object>
          </Children>
        </Object>
      </Children>
    </Object>
</ModelDesign>
```

Listing B.3: Excerpt of the *fortiss UR* ModelDesign XML file defining a specific robot instance and its skills on the example of a Universal Robots UR5.

Bibliography

Akin Başer, Ç. (2018). Master Thesis: Extension of an OPC UA Test Suite for Performance Evaluation of Embedded Devices using open62541.

Arai, T., Aiyama, Y., Maeda, Y., Sugi, M., & Ota, J. (2000). Agile Assembly System by "Plug and Produce". *CIRP Annals*, *49*(1). https://doi.org/10.1016/S0007-8506(07)62883-2

Ayres, R. U. (1989). *Technological transformations and long waves* (tech. rep.). International Institute for Applied Systems Analysis. https://linkinghub.elsevier.com/retrieve/pii/0040162590900573

Azaiez, S., Boc, M., Cudennec, L., Da Silva Simoes, M., Haupert, J., Kchir, S., Klinge, X., Labidi, W., Nahhal, K., Pfrommer, J., Schleipen, M., Schulz, C., & Tortech, T. (2016). Towards Flexibility in Future Industrial Manufacturing: A Global Framework for Self-organization of Production Cells. *Procedia Computer Science*, *83*(September). https://doi.org/10.1016/j.procs.2016.04.264

Balador, A., Ericsson, N., & Bakhshi, Z. (2017, September). Communication middleware technologies for industrial distributed control systems: A literature review, In *IEEE International Conference on Emerging Technologies and Factory Automation (ETFA)*, IEEE. https://doi.org/10.1109/ETFA.2017.8247730

Bedenbender, H., Bock, J., Boss, B., Diedrich, C., Garrels, K., Graf Gatterburg, A., Heidrich, K., Hillermeier, O., Sauer. Manuel, Schmidt, J., Werner, T., & Zimmermann, P. (2019). *Verwaltungsschalen in der Praxis* (tech. rep.). Plattform Industrie 4.0. www.bmwi.de

Bernstein, P. A. (1993). *Middleware: An Architecture for Distributed System Services* (tech. rep.). Digital Equipment Corporation, Cambridge Research Lab. https://www.hpl.hp.com/techreports/Compaq-DEC/CRL-93-6.pdf

Brandenbourger, B., & Durand, F. (2018, September). Design Pattern for Decomposition or Aggregation of Automation Systems into Hierarchy Levels, In *IEEE International Conference on Emerging Technologies and Factory Automation (ETFA)*, IEEE. https://doi.org/10.1109/ETFA.2018.8502627

Breitkreuz, A. (2019). Bachelor Thesis: Implementation of Modular Robot Force Skills in OPC-UA for Industrial Robots.

Cândido, G., Jammesy, F., De Oliveira, J. B., & Colomboz, A. W. (2010). SOA at device level in the industrial domain: Assessment of OPC UA and DPWS specifications. *IEEE International Conference on Industrial Informatics (INDIN)*. https://doi.org/10.1109/INDIN.2010.5549676

Cavalieri, S., & Cutuli, G. (2010). Performance evaluation of OPC UA. *2010 IEEE 15th Conference on Emerging Technologies & Factory Automation (ETFA 2010)*. https://doi.org/10.1109/ETFA.2010.5641184

Cenedese, A., Frodella, M., Tramarin, F., & Vitturi, S. (2019). Comparative assessment of different OPC UA open-source stacks for embedded systems. *IEEE International Conference on Emerging Technologies and Factory Automation, ETFA*. https://doi.org/10.1109/ETFA.2019.8869187

Chang, T.-C., & Wysk, R. A. (1997). *Computer-Aided Manufacturing* (2nd). Prentice Hall PTR, Upper Saddle River, NJ, United States.

Chen, Y., & Kunz, T. (2016, April). Performance evaluation of IoT protocols under a constrained wireless access network, In *International Conference on Selected Topics in Mobile and Wireless Networking, MoWNeT*, IEEE. https://doi.org/10.1109/MoWNet.2016.7496622

Cheng, C. H., Geisinger, M., Ruess, H., Buckl, C., & Knoll, A. (2012). MGSyn: Automatic synthesis for industrial automation. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*. https://doi.org/10.1007/978-3-642-31424-7_46

Constable, G., & Somerville, B. (2004). *A century of innovation: twenty engineering achievements that transformed our lives* (Vol. 42). https://doi.org/10.5860/choice.42-0268

Dai, W., Wang, P., Sun, W., Wu, X., Zhang, H., Vyatkin, V., & Yang, G. (2019). Semantic Integration of Plug-and-Play Software Components for Industrial Edges Based on Microservices. *IEEE Access*, *7*. https://doi.org/10.1109/ACCESS.2019.2938565

Decotignie, J.-D. (2005). Ethernet-Based Real-Time and Industrial Communications. *Proceedings of the IEEE*, *93*(6). https://doi.org/10.1109/JPROC.2005.849721

de Melo, P. F. S., & Godoy, E. P. (2019, June). Controller Interface for Industry 4.0 based on RAMI 4.0 and OPC UA, In *Workshop on Metrology for Industry 4.0 and IoT (MetroInd4.0&IoT)*, IEEE. https://doi.org/10.1109/METROI4.2019.8792837

Directorate General for Internal Policies. (2016). European Study 2016: Industry 4.0.

Dorofeev, K., Cheng, C.-H., Guedes, M., Ferreira, P., Profanter, S., Zoitl, A., Guedes, M., Profanter, S., & Zoitl, A. (2017, September). Device Adapter Concept towards Enabling Plug&Produce Production Environments, In *IEEE International Conference on Emerging Technologies and Factory Automation (ETFA)*. https://doi.org/10.1109/ETFA.2017.8247570

Dorofeev, K., Profanter, S., Cabral, J., Ferreira, P., & Zoitl, A. (2019). Agile Operational Behavior for the Control-Level Devices in Plug&Produce Production Environments. *IEEE International Conference on Emerging Technologies and Factory Automation, ETFA*, (July). https://doi.org/10.1109/ETFA.2019.8869208

Dorofeev, K., & Wenger, M. (2019, September). Evaluating Skill-Based Control Architecture for Flexible Automation Systems, In *IEEE International Conference on Emerging Technologies and Factory Automation (ETFA)*, IEEE. https://doi.org/10.1109/ETFA.2019.8869050

Dorofeev, K., & Zoitl, A. (2018). Skill-based Engineering Approach using OPC UA Programs, In *Proceedings of the IEEE International Conference on Industrial Informatics (INDIN)*. https://doi.org/10.1109/INDIN.2018.8471978

Drahos, P., Kucera, E., Haffner, O., & Klimo, I. (2018). Trends in industrial communication and OPC UA, In *Proceedings of the 29th International Conference on Cybernetics and Informatics, K and I*. https://doi.org/10.1109/CYBERI.2018.8337560

Drath, R., Lüder, A., Peschke, J., & Hundt, L. (2008). AutomationML - The glue for seamless Automation engineering. *IEEE International Conference on Emerging Technologies and Factory Automation, ETFA*. https://doi.org/10.1109/ETFA.2008.4638461

Dürkop, L., Imtiaz, J., & Trsek, H. (2012). Service-Oriented Architecture for the Autoconfiguration of Real-Time Ethernet Systems. *Iot-At-Work.Eu*.

Dürkop, L., Imtiaz, J., Trsek, H., Wisniewski, L., & Jasperneite, J. (2013). Using OPC-UA for the autoconfiguration of real-time Ethernet systems. *IEEE International Conference on Industrial Informatics (INDIN)*. https://doi.org/10.1109/INDIN.2013.6622890

Dürkop, L., Trsek, H., Otto, J., & Jasperneite, J. (2014). A field level architecture for reconfigurable real-time automation systems. *IEEE International Workshop on Factory Communication Systems - Proceedings, WFCS*. https://doi.org/10.1109/WFCS.2014.6837601

Eichhorn, M., Pfannenstein, M., Muhra, D., & Steinbach, E. (2010). A SOA-based middleware concept for in-vehicle service discovery and device integration. *IEEE Intelligent Vehicles Symposium, Proceedings*. https://doi.org/10.1109/IVS.2010.5547977

Ferreira, P., & Lohse, N. (2012). Configuration model for evolvable assembly systems. *CIRP Conference on Assembly Technologies and Systems (CATS) 2012*, (May 2012).

Freeman, C., & Soete, L. (1997). *The Economics of Industrial Innovation* (Third Edit, Vol. 16). MIT Press.

Gašpar, T., Deniša, M., Radanovič, P., Ridge, B., Savarimuthu, T. R., Kramberger, A., Priggemeyer, M., Roßmann, J., Wörgötter, F., Ivanovska, T., Parizi, S., Gosar, Ž., Kovač, I., & Ude, A. (2020). Smart hardware integration with advanced robot programming technologies for efficient reconfiguration of robot workcells. *Robotics and Computer-Integrated Manufacturing*, *66*(March). https://doi.org/10.1016/j.rcim.2020.101979

Girbea, A., Demeter, R., & Sisak, F. (2011). Automatic address space generation for an OPC UA server of a flexible manufacturing system. *SACI 2011 - 6th IEEE International Symposium on Applied Computational Intelligence and Informatics, Proceedings*. https://doi.org/10.1109/SACI.2011.5873052

Girbea, A., Suciu, C., Nechifor, S., & Sisak, F. (2014). Design and implementation of a service-oriented architecture for the optimization of industrial applications. *IEEE Transactions on Industrial Informatics*, *10*(1). https://doi.org/10.1109/TII.2013.2253112

Guttman, E. (1999). Service location protocol: automatic discovery of IP network services. *IEEE Internet Computing*, *3*(4). https://doi.org/10.1109/4236.780963

Haage, M., Profanter, S., Kessler, I., Perzylo, A., Somani, N., Sörnmo, O., Karlsson, M., Robertz, S., Nilsson, K., Resch, L., & Marti, M. (2016). On cognitive robot woodworking in SMErobotics, In *International Symposium on Robotics (ISR)*.

Hammerstingl, V., & Reinhart, G. (2015). Unified Plug&Produce architecture for automatic integration of field devices in industrial environments, In *Proceedings of the IEEE International Conference on Industrial Technology*. https://doi.org/10.1109/ICIT.2015.7125383

Haskamp, H., Meyer, M., Mollmann, R., Orth, F., & Colombo, A. W. (2017, July). Benchmarking of existing OPC UA implementations for Industrie 4.0-compliant digitalization solutions, In *IEEE International Conference on Industrial Informatics (INDIN)*, IEEE. https://doi.org/10.1109/INDIN.2017.8104838

Hodek, S., & Schlick, J. (2012). Ad hoc field device integration using device profiles, concepts for automated configuration and web service technologies. *International Multi-Conference on Systems, Signals and Devices, SSD 2012 - Summary Proceedings*. https://doi.org/10.1109/SSD.2012.6197997

Hohpe, G., & Woolf, B. (2003). *Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions*. Addison Wesley.

Huang, Z., Chen, L., Zhang, L., Fan, S., & Fan, D. (2019). Research on software synchronization method of real-time ethernet distributed motion control system. *Assembly Automation*, *39*(5). https://doi.org/10.1108/AA-12-2018-0265

Imtiaz, J., & Jasperneite, J. (2013). Scalability of OPC-UA down to the chip level enables Internet of Things. *IEEE International Conference on Industrial Informatics (INDIN)*. https://doi.org/10.1109/INDIN.2013.6622935

Jirkovsky, V., Obitko, M., Kadera, P., & Marik, V. (2018). Toward Plug&Play Cyber-Physical System Components. *IEEE Transactions on Industrial Informatics*, *14*(6). https://doi.org/10.1109/TII.2018.2794982

Kainz, G., Keddis, N., Pensky, D., Buckl, C., Zoitl, A., Pittschellis, R., & Kärcher, B. (2013). AutoPnP – Plug-and-produce in der Automation. *atp edition*, *55*(4).

Kambhampati, C., Patton, R., & Uppal, F. (2006). Reconfiguration in Networked Control Systems: Fault Tolerant Control and Plug-and-Play, In *IFAC Proceedings Volumes*, IFAC. https://doi.org/10.3182/20060829-4-CN-2909.00020

Kaspar, M., Kogan, Y., Venet, P., Weser, M., & Zimmermann, U. E. (2018). Tool and technology independent function interfaces by using a generic OPC UA representation, In *Proceedings of the IEEE International Conference on Emerging Technologies and Factory Automation (ETFA)*. https://doi.org/10.1109/ETFA.2018.8502647

Kaur, J., & Kaur, K. (2017). Internet of Things: A Review on Technologies, Architecture, Challenges, Applications, Future Trends. *International Journal of Computer Network and Information Security*, *9*(4). https://doi.org/10.5815/ijcnis.2017.04.07

Keddis, N., Kainz, G., & Zoitl, A. (2014). Capability-based planning and scheduling for adaptable manufacturing systems, In *IEEE International Conference on Emerging Technologies and Factory Automation (ETFA)*. https://doi.org/10.1109/ETFA.2014.7005213

Keddis, N., Kainz, G., & Zoitl, A. (2015). Product-Driven Generation of Action Sequences for Adaptable Manufacturing Systems, In *International Federation of Automatic Control*, Elsevier Ltd. https://doi.org/10.1016/j.ifacol.2015.06.299

Kinzel, H. (2017). Industry 4.0 – Where Does This Leave the Human Factor? In *Journal of Urban Culture Research*. https://doi.org/10.14456/JUCR.2017.14

Knoll, A. C. (2001). Distributed contract networks of sensor agents with adaptive reconfiguration: Modelling, simulation, implementation and experiments. *Journal of the Franklin Institute*, *338*(6). https://doi.org/10.1016/S0016-0032(01)00019-9

Koch, V., Geissbauer, R., Kuge, S., & Schrauf, S. (2014). Chancen und Herausforderungen der vierten industriellen Revolution. *Pwc*.

Koziolek, H., Burger, A., & Doppelhamer, J. (2018). Self-Commissioning Industrial IoT-Systems in Process Automation: A Reference Architecture. *IEEE International Conference on Software Architecture (ICSA)*. https://doi.org/10.1109/ICSA.2018.00029

Koziolek, H., Burger, A., Platenius-Mohr, M., Ruckert, J., & Stomberg, G. (2019). OpenPnP: A Plug-And-Produce Architecture for the Industrial Internet of Things, In *Proceedings of the IEEE/ACM International Conference on Software Engineering: Software Engineering in Practice, ICSE-SEIP*. https://doi.org/10.1109/ICSE-SEIP.2019.00022

Koziolek, H., Burger, A., Platenius-Mohr, M., Rückert, J., Mendoza, F., & Braun, R. (2020). Automated industrial IoT-device integration using the OpenPnP reference architecture. *Software - Practice and Experience*, *50*(3). https://doi.org/10.1002/spe.2765

Kraft, M. (2016). Bachelor Thesis: Design and Evaluation of an Intuitive Graphical User Interface for Industrial Robotic System Programming.

Lepuschitz, W., Zoitl, A., Vallée, M., & Merdan, M. (2011). Toward self-reconfiguration of manufacturing systems using automation agents. *IEEE Transactions on Systems, Man and Cybernetics Part C: Applications and Reviews*, *41*(1). https://doi.org/10.1109/TSMCC.2010.2059012

Madiwalar, B. (2019). Master Thesis: Plug & Produce for Industry 4.0 using Software-defined Networking and OPC UA.

Madiwalar, B., Schneider, B., & Profanter, S. (2019). Plug and Produce for Industry 4.0 using Software-defined Networking and OPC UA, In *Proceedings of the IEEE International Conference on Emerging Technologies And Factory Automation (ETFA)*.

Maeda, Y., Kikuchi, H., Izawa, H., Ogawa, H., Sugi, M., & Arai, T. (2007). "Plug & Produce" functions for an easily reconfigurable robotic assembly cell. *Assembly Automation*, *27*. https://doi.org/10.1108/01445150710763286

Maruyama, Y., Kato, S., & Azumi, T. (2016). Exploring the performance of ROS2. *Emsoft '16*, 15334406. https://doi.org/10.1145/2968478.2968502

Meeussen, W., Fernandez Perdomo, E., Bohren, J., Coleman, D., Magyar, B., Pradeep, V., Marder-Eppstein, E., Lüdtke, M., Raiola, G., Chitta, S., & Rodríguez Tsouroukdissian, A. (2017). ros_control: A generic and simple control framework for ROS. *The Journal of Open Source Software*, *2*(20). https://doi.org/10.21105/joss.00456

Mehrabi, M. G., Ulsoy, A. G., & Koren, Y. (2000). Reconfigurable manufacturing systems: key to future manufacturing. *Journal of Intelligent Manufacturing*, *11*(4). https://doi.org/10.1023/A:1008930403506

Mun, D.-H., Dinh, M. L., & Kwon, Y.-W. (2016, June). An Assessment of Internet of Things Protocols for Resource-Constrained Applications, In *2016 IEEE 40th Annual Computer Software and Applications Conference (COMPSAC)*, IEEE. https://doi.org/10.1109/COMPSAC.2016.51

National Research Council. (1998, November). *Visionary Manufacturing Challenges for 2020* [The study identifed reconfigurable manufacturing as the highest priority for future research in manufacturing, and one of the six key manufacturing challenges for the year 2020.]. National Academies Press. https://doi.org/10.17226/6314

Neumann, P. (2007). Communication in industrial automation-What is going on? *Control Engineering Practice*, *15*(11). https://doi.org/10.1016/j.conengprac.2006.10.004

Nof, S. Y. (1985). *Handbook of Industrial Robotics*.

Nsaibi, S., & Leurs, L. (2016). Time Sensitive Networking. *atp edition*, *58*. https://doi.org/https://doi.org/10.17560/atp.v58i10.583

OASIS. (2009). *Devices Profile for Web Services Version 1.1* (tech. rep.). OASIS.

OPC Foundation. (2015). *OPC UA Specification Part 12: Discovery* (tech. rep.). OPC Foundation. https://opcfoundation.org/developer-tools/specifications-unified-architecture/part-12-discovery/

OPC Foundation. (2019a). *OPC UA for Robotics Companion Specification – Part 1: Vertical integration* (tech. rep.). OPC Foundation.

OPC Foundation. (2019b). *OPC UA Specification Part 10 - Programs. Release 1.04* (tech. rep.). OPC Foundation.

Palm, F., Grüner, S., Pfrommer, J., Graube, M., & Urbas, L. (2014). open62541 – der offene OPC UA Stack. *5. Jahreskolloquium "Kommunikation in der Automation" (KommA 2014)*.

Panda, S. K., Schroder, T., Wisniewski, L., & Diedrich, C. (2018, September). Plug & Produce Integration of Components into OPC UA based data-space, In *IEEE International Conference on Emerging Technologies and Factory Automation (ETFA)*, IEEE. https://doi.org/10.1109/ETFA.2018.8502663

Panda, S. K., Wisniewski, L., Ehrlich, M., Majumder, M., & Jasperneite, J. (2020, April). Plug & Play Retrofitting Approach for Data Integration to the Cloud, In *2020 16th IEEE International Conference on Factory Communication Systems (WFCS)*, IEEE. https://doi.org/10.1109/wfcs47810.2020.9114523

Pardo-Castellote, G. (2003). OMG data distribution service: architectural overvie, In *IEEE Military Communications Conference (MILCOM)*, IEEE. https://doi.org/10.1109/MILCOM.2003.1290110

Pedersen, M. R., Bøgh, S., Madsen, O., Krüger, V., Nalpantidis, L., Schou, C., & Andersen, R. S. (2015). Robot skills for manufacturing: From concept to industrial deployment. *Robotics and Computer-Integrated Manufacturing*, *37*. https://doi.org/10.1016/j.rcim.2015.04.002

Perzylo, A., Grothoff, J., Lucio, L., Weser, M., Malakuti, S., Venet, P., Aravantinos, V., & Deppe, T. (2019). Capability-based semantic interoperability of manufacturing resources: A BaSys 4.0 perspective, In *IFAC Conference on Manufacturing Modeling, Management, and Control (MIM)*. https://doi.org/10.1016/j.ifacol.2019.11.427

Perzylo, A., Kessler, I., Profanter, S., & Rickert, M. (2020). Toward a Knowledge-Based Data Backbone for Seamless Digital Engineering in Smart Factories, In *IEEE International Conference on Emerging Technologies and Factory Automation (ETFA)*.

Perzylo, A., Profanter, S., Rickert, M., & Knoll, A. (2019, September). OPC UA NodeSet Ontologies as a Pillar of Representing Semantic Digital Twins of Manufacturing Resources, In *2019 24th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA)*, IEEE. https://doi.org/10.1109/ETFA.2019.8868954

Perzylo, A., Rickert, M., Kahl, B., Somani, N., Lehmann, C., Kuss, A., Profanter, S., Beck, A. B., Haage, M., Hansen, M. R., Nibe, M. T., Roa, M. A., Sornmo, O., Robertz, S. G., Thomas, U., Veiga, G., Topp, E. A., Kessler, I., & Danzer, M. (2019). SMErobotics: Smart robots for Flexible Manufacturing. *IEEE Robotics and Automation Magazine*, *26*(1). https://doi.org/10.1109/MRA.2018.2879747

Perzylo, A., Somani, N., Profanter, S., Gaschler, A., Cai, C., Griffiths, S., Rickert, M., Lafrenz, R., & Knoll, A. (2015). Ubiquitous Semantics: Representing and Exploiting Knowledge, Geometry, and Language for Cognitive Robot Systems. *IEEE/RAS International Conference on Humanoid Robots (HUMANOIDS), Workshop Towards Intelligent Social Robots - Current Advances in Cognitive Robotics*, (November).

Perzylo, A., Somani, N., Profanter, S., Kessler, I., Rickert, M., & Knoll, A. (2016). Intuitive Instruction of Industrial Robots: Semantic Process Descriptions for Small Lot Production, In *Proceedings of the IEEE International Conference on Intelligent Robots and Systems (IROS)*. https://doi.org/10.1109/IROS.2016.7759358

Perzylo, A., Somani, N., Profanter, S., Rickert, M., & Knoll, A. (2015a). Multimodal Binding of Parameters for Task-based Robot Programming Based on Semantic Descriptions of Modalities and Parameter types, In *CEUR Workshop Proceedings*.

Perzylo, A., Somani, N., Profanter, S., Rickert, M., & Knoll, A. (2015b). Toward Efficient Robot Teach-In and Semantic Process Descriptions for Small Lot Sizes, In *Robotics: Science and Systems (RSS), Workshop on Combining AI Reasoning and Cognitive Science with Robotics*.

Pfrommer, J., Ebner, A., Ravikumar, S., & Karunakaran, B. (2018). Open Source OPC UA PubSub over TSN for Realtime Industrial Communication, In *International Conference on Emerging Technologies and Factory Automation*.

Pfrommer, J., Gruner, S., & Palm, F. (2016, June). Hybrid OPC UA and DDS: Combining architectural styles for the industrial internet, In *IEEE International Workshop on Factory Communication Systems (WFCS)*, Institute of Electrical; Electronics Engineers Inc. https://doi.org/10.1109/WFCS.2016.7496515

Pfrommer, J., Stogl, D., Aleksandrov, K., Escaida Navarro, S., Hein, B., & Beyerer, J. (2015). Plug & produce by modelling skills and service-oriented orchestration of recon-

figurable manufacturing systems. *Automatisierungstechnik*, *63*(10). https://doi.org/10.1515/auto-2014-1157

Pfrommer, J., Stogl, D., Aleksandrov, K., Schubert, V., & Hein, B. (2014, September). Modelling and orchestration of service-based manufacturing systems via skills, In *Proceedings of the 2014 IEEE Emerging Technology and Factory Automation (ETFA)*, IEEE. https://doi.org/10.1109/ETFA.2014.7005285

Profanter, S., Breitkreuz, A., Rickert, M., & Knoll, A. (2019, September). A Hardware-Agnostic OPC UA Skill Model for Robot Manipulators and Tools, In *Proceedings of the IEEE International Conference on Emerging Technologies and Factory Automation (ETFA)*, IEEE. https://doi.org/10.1109/ETFA.2019.8869205

Profanter, S., Dorofeev, K., Zoitl, A., & Knoll, A. (2018). OPC UA for Plug & Produce: Automatic Device Discovery using LDS-ME, In *IEEE International Conference on Emerging Technologies and Factory Automation (ETFA)*. https://doi.org/10.1109/ETFA.2017.8247569

Profanter, S., Perzylo, A., Rickert, M., & Knoll, A. (2020). A Generic Plug & Produce System composed of Semantic OPC UA Skills. *Submitted for review in IEEE Open Journal of the Industrial Electronics Society*.

Profanter, S., Perzylo, A., Somani, N., Rickert, M., & Knoll, A. (2015). Analysis and Semantic Modeling of Modality Preferences in Industrial Human-Robot Interaction, In *IEEE International Conference on Intelligent Robots and Systems*. https://doi.org/10.1109/IROS.2015.7353613

Profanter, S., Tekat, A., Dorofeev, K., Rickert, M., & Knoll, A. (2019). OPC UA versus ROS, DDS, and MQTT: Performance Evaluation of Industry 4.0 Protocols, In *Proceedings of the IEEE International Conference on Industrial Technology (ICIT)*. https://doi.org/10.1109/ICIT.2019.8755050

Quigley, M., Conley, K., Gerkey, B. P., Faust, J., Foote, T., Leibs, J., Wheeler, R. C., & Ng, A. Y. (2009). ROS: an open-source Robot Operating System, In *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA)*.

Rickert, M., & Gaschler, A. (2017, September). Robotics library: An object-oriented approach to robot applications, In *IEEE International Conference on Intelligent Robots and Systems*, IEEE. https://doi.org/10.1109/IROS.2017.8202232

Rocha, M. S., Sestito, G. S., Dias, A. L., Turcato, A. C., & Brandao, D. (2018, April). Performance Comparison Between OPC UA and MQTT for Data Exchange, In *Workshop on Metrology for Industry 4.0 and IoT*, IEEE. https://doi.org/10.1109/METROI4.2018.8428342

Runde, S., Wolf, G., Braun, M., & Siemens, A. G. (2013, September). EDDL and semantic web - From field device integration (FDI) to Future Device Management (FDM), In *IEEE Conference on Emerging Technologies & Factory Automation (ETFA)*, IEEE. https://doi.org/10.1109/ETFA.2013.6647962

Rüth, J., Schmidt, F., Serror, M., Wehrle, K., & Zimmermann, T. (2017). Communication and Networking for the Industrial Internet of Things. https://doi.org/10.1007/978-3-319-42559-7_12

Sauter, T., Soucek, S., Kastner, W., & Dietrich, D. (2011). The Evolution of Factory and Building Automation. *IEEE Industrial Electronics Magazine*, *5*(3). https://doi.org/10.1109/MIE.2011.942175

Schebek, L., Kannengießer, J., Campitelli, A., Fischer, J., Abele, E., Bauerdick, C., Anderl, R., Haag, S., Sauer, A., Mandel, J., Lucke, D., Bogdanov, I., Nuffer, A.-K., Steinhilper, R., Böhner, J., Lothes, G., Schock, C., Zühlke, D., Plociennik, C., & Bergweiler, S. (2017). Ressourceneffizienz durch Industrie 4.0.

Scheifele, S., Friedrich, J., Lechler, A., & Verl, A. (2014). Flexible, Self-configuring Control System for a Modular Production System. *Procedia Technology*, *15*. https://doi.org/10.1016/j.protcy.2014.09.094

Schleipen, M., Lüder, A., Sauer, O., Flatt, H., & Jasperneite, J. (2015). Requirements and concept for Plug-and-Work: Adaptivity in the context of Industry 4.0. *At-Automatisierungstechnik*, *63*(10). https://doi.org/10.1515/auto-2015-0015

Schleipen, M., Pfrommer, J., Aleksandrov, K., Stogl, D., Navarro, S. E., Engler-Bunte-Ring, K., & Beyerer, J. (2014). AutomationML to describe skills of production plants based on the PPR concept, In *Proceedings of the AutomationML User Conference*.

Schmid, L. (2019). Bachelor Thesis: Modular Robot Tools With a generic Software Interface for Flexible Manufacturing using Robots.

Singh, B., & Sellappan, N. (2013). Evolution of Industrial Robots and their Applications. *International Journal of Emerging Technology and Advanced Engineering*, *9001*(5).

SPARC. (2020). *Multi-Annual Roadmap (MAR), Rev B, Section 4.1* (tech. rep.). euRobotics.

Srivastava, S., Anmulwar, S., Sapkal, A. M., Batra, T., Gupta, A. K., & Kumar, V. (2014). Comparative study of various traffic generator tools. *2014 Recent Advances in Engineering and Computational Sciences, RAECS 2014*. https://doi.org/10.1109/RAECS.2014.6799557

Stanford-Clark, A., & Truong, H. L. (2013). *MQTT For Sensor Networks (MQTT-SN) - Protocol Specification* (tech. rep.). International Business Machines Corporation (IBM).

Staufen AG. (2019). *German Industry 4.0 Index 2019* (tech. rep.).

Tekat, A. (2018). Master Thesis: Implementation of a Test Suite to Compare the Performance of Different Industry 4.0 Communication Protocols.

Trunzer, E., Calà, A., Leitão, P., Gepp, M., Kinghorst, J., Lüder, A., Schauerte, H., Reifferscheid, M., & Vogel-Heuser, B. (2019). System architectures for Industrie 4.0 applications. *Production Engineering*, *13*(3-4). https://doi.org/10.1007/s11740-019-00902-6

Tsardoulias, E., & Mitkas, P. (2017, November). *Robotic frameworks, architectures and middleware comparison* (tech. rep.). http://arxiv.org/abs/1711.06842

Turiy, T. (2020). Bachelor Thesis: Control Constructs, Error Handling and Parameter Representation for a Knowledge-based Robot Skill Editor.

VDMA, & Fraunhofer IOSB-INA. (2017). *Industrie 4.0 Kommunikation mit OPC UA - Leitfaden zur Einführung in den Mittelstand* (tech. rep.).

Veichtlbauer, A., Ortmayer, M., Heistracher, T., & Armin Veichtlbauer, Martin Ortmayer, T. H. (2017). OPC UA Integration for Field Devices. *Proceedings - 2017 IEEE 15th*

International Conference on Industrial Informatics, INDIN 2017. https://doi.org/10.1109/INDIN.2017.8104808

Wielander, F. (2020). Master Thesis: Knowledge-based Skill Editor for the Hierarchical Composition of Device Capabilities.

Wollschlaeger, M., Sauter, T., & Jasperneite, J. (2017). The Future of Industrial Communication: Automation Networks in the Era of the Internet of Things and Industry 4.0. *IEEE Industrial Electronics Magazine*, *11*(1). https://doi.org/10.1109/MIE.2017.2649104

Yan, J., Meng, Y., Lu, L., & Li, L. (2017). Industrial Big Data in an Industry 4.0 Environment: Challenges, Schemes, and Applications for Predictive Maintenance. *IEEE Access*, *5*. https://doi.org/10.1109/ACCESS.2017.2765544

Yusuf, Y. Y., Sarhadi, M., & Gunasekaran, A. (1999). Agile manufacturing: the drivers, concepts and attributes. *International Journal of Production Economics*, *62*(1). https://doi.org/10.1016/S0925-5273(98)00219-9

Zamalloa, I., Muguruza, I., Hernández, A., Kojcev, R., & Mayoral, V. (2018). An information model for modular robots: the Hardware Robot Information Model (HRIM) [https://hackernoon.com/introducing-the-hardware-robot-information-model-hrim-8f0da3f22f67], 1802.01459.

Zimmermann, P., Axmann, E., Brandenbourger, B., Dorofeev, K., Mankowski, A., & Zanini, P. (2019). Skill-based Engineering and Control on Field-Device-Level with OPC UA. *IEEE International Conference on Emerging Technologies and Factory Automation, ETFA*, *2019-Septe*(July). https://doi.org/10.1109/ETFA.2019.8869473

ZVEI. (2015a). *The Industrie 4.0 Component* (tech. rep.). Zentralverband Elektrotechnik- und Elektronikindustrie e.V. (ZVEI).

ZVEI. (2015b). *The Reference Architectural Model Industrie 4.0 (RAMI 4.0)* (tech. rep. July). Zentralverband Elektrotechnik- und Elektronikindustrie e.V. (ZVEI).