universität wien

MASTERARBEIT / MASTER'S THESIS

Titel der Masterarbeit / Title of the Master's Thesis

„Collaboration patterns for collaborative robots“

verfasst von / submitted by

Stefan Samhaber BSc

angestrebter akademischer Grad / in partial fulfilment of the requirements for the degree of

Master of Science (MSc)

Wien, 2022 / Vienna, 2022

Studienkennzahl lt. Studienblatt / degree programme code as it appears on the student record sheet:	UA 066 926
Studienrichtung lt. Studienblatt / degree programme as it appears on the student record sheet:	Masterstudium Wirtschaftsinformatik
Betreut von / Supervisor:	Univ.-Prof. Dipl.-Ing. Mag. Dr. techn. Maria Leitner, Bakk.

Acknowledgements

I want to express my sincere appreciation to my supervisor, Univ.-Prof. Dipl.-Ing. Mag. Dr. techn. Maria Leitner, Bakk., who guided me all the way through this research project and provided me with very valuable input through hours of interesting discussions.

Furthermore, I would like to express my gratitude to my friends that I met during my time at the University for the countless hours we spent together.

Finally, I want to thank my family and partner for their emotional support throughout my academic journey.

Author

Stefan Samhaber

Abstract

The manufacturing industry is in constant change, driven by a new era of industrial revolution, Industry 4.0. Along those changes, new demands on the production environment arise, such as highly customized product variants and mass customization. To deal with these new requirements, there is a shift from conventional industrial robots to collaborative robots ("cobots"). This new generation of robots is able to work collaboratively with humans and aims to combine the strengths of humans and robots. Smart factories aim at adaptable, flexible and cost-efficient production. Behind these goals is a production environment where all sorts of machines and information systems are able to exchange data through a common network. However, most of the highly specialized systems on the shop floor are not designed to interact with arbitrary systems.

This research investigates the Human-Robot Collaboration domain, more specifically collaboration patterns between humans and cobots, analyzes various use cases from industry and research, and explores how the collaboration patterns can be specified in business processes. It also identifies the lack of attention to multiple human and robotic actors in the majority of current literature. A formal definition is proposed to describe these collaborative patterns in business processes on the level of tasks. The identified collaboration patterns are called Synchronized, Cooperation, Collaboration and Coexistence. For each pattern, formal requirements are specified and appropriate use cases are instantiated.

A test setup is developed for the implementation of collaborative use cases. This setup combines a Workflow Management System and cobot simulations by using the Open Platform Communications Unified Architecture (OPC UA) data exchange standard. Business Process Model and Notation (BPMN) is used for the design of the process models. The ability to control the implemented use cases with business processes reflects the flexibility requirements of modern production. Using this setup, all four collaborative patterns are evaluated and successfully demonstrated with seven use cases.

Kurzfassung

Die Fertigungsbranche befindet sich in stetigem Wandel, angetrieben durch eine neue Ära der industriellen Revolution, genannt Industrie 4.0. Im Zuge dieser Veränderungen ergeben sich neue Anforderungen an die Produktionslandschaft, wie z. B. stark individualisierte Produktvarianten und kundenspezifische Massenfertigung. Um diesen neuen Anforderungen gerecht zu werden, findet ein Wechsel von herkömmlichen Industrierobotern hin zu kollaborativen Robotern ("Cobots") statt. Diese neue Generation von Robotern ist in der Lage, mit Menschen kollaborativ zu arbeiten und zielt darauf ab, die Stärken von Menschen und Robotern zu kombinieren. Smart Factories haben eine anpassungsfähige, flexible und kosteneffiziente Produktion zum Ziel. Dahinter steht eine Produktionsumgebung, in der alle Arten von Maschinen und IT-Systemen über ein gemeinsames Netzwerk Daten austauschen können. Die meisten dieser hoch spezialisierten Systeme in der Produktion sind jedoch nicht für die Interaktion mit beliebigen Systemen ausgelegt.

Diese Forschungsarbeit untersucht den Bereich der Mensch-Roboter-Kollaboration, insbesondere die Kollaborationsmuster zwischen Menschen und Robotern, analysiert verschiedene Anwendungsfälle aus Industrie und Forschung und untersucht, wie diese Kollaborationsmuster in Geschäftsprozessen spezifiziert werden können. Darüber hinaus wird festgestellt, dass der Großteil der aktuellen Literatur kaum auf die Zusammenarbeit zwischen mehreren Menschen und Cobots eingeht. Es wird eine formale Definition vorgeschlagen, um diese Kollaborationsmuster in Geschäftsprozessen auf der Ebene von Aufgaben zu beschreiben. Die identifizierten Kollaborationsmuster werden als Synchronisation, Kooperation, Kollaboration und Koexistenz bezeichnet. Für jedes Muster werden formale Anforderungen spezifiziert und entsprechende Anwendungsfälle instanziiert.

Für die Implementierung der kollaborativen Anwendungsfälle wird ein Testaufbau entwickelt. Dieser Aufbau kombiniert ein Workflow Management System mit Cobot-Simulationen unter Verwendung des Open Platform Communications Unified Architecture (OPC UA) Datenaustauschstandards. Für das Design der Prozessmodelle wird BPMN (Business Process Model and Notation) verwendet. Die Fähigkeit, die implementierten Anwendungsfälle mit Geschäftsprozessen zu steuern, spiegelt die Flexibilitätsanforderungen der modernen Produktion wider. Unter Verwendung des Testsystems werden alle vier Kollaborationsmuster evaluiert und mit sieben Anwendungsfällen erfolgreich demonstriert.

Contents

List of Tables

List of Figures

Listings

1. Introduction

Over the last few years, the manufacturing industry has and will continue to change significantly, as consumer markets tend from mass-produced goods towards increasingly high customized products and mass customization [54]. A growing amount of product variants combined with shorter product lifecycles and a demand for shorter lead times make for an increasingly complex manufacturing landscape [57]. In order to keep up with the ever-changing consumers' demands, there is a need for an adaptive, flexible and cost-efficient production environment. Industrial automation using traditional industrial robots can very well fulfill the requirements of mass production, such as high efficiency and repeatability [41]. However, industrial robots require safety fences and peripheral safety equipment, thus increasing the needed floor space and decreasing flexibility [43]. Furthermore, they are not suited for frequently changing product variants, as they are specifically programmed for certain tasks and adapting them to new situations requires specialized personnel and usually a significant amount of time. Hence, a lot of tasks are performed by manual labor, as humans are excellent in dealing with uncertainties and variability coming from mass customization [41]. Humans, however, are limited by their physical abilities and can't compete with robots in various aspects like speed, repeatability, strength, etc., which results in reduced efficiency and quality [41]. The combination of the complementary strengths of humans and robots [30], [50] would provide all means to deal with the aforementioned challenges of the modern manufacturing in the Industry 4.0 era. One approach to achieve this is to use of a new generation of robots, the so-called collaborative robots or cobots [57]. The most recent research discipline on this topic is called Human-Robot Collaboration (HRC).

Human-Robot Collaboration

Human-Robot Collaboration is a promising interdisciplinary research field. It comprises several areas like classical robotics, cognitive sciences and psychology [7]. HRC focuses on enabling robots and humans to complete collaborative tasks together [41]. It will potentially have a great economic impact as it offers a wide range of possible applications and future scenarios [7]. A survey of Chandrasekaran and Conrad [20] showed a multitude of different application scenarios. For example, robots assisting elderly and aged people, or interacting with young children for learning purposes (e.g., playfully improving geometrical thinking). Other possible fields of application are medicine, entertainment, industry, military, and also space exploration [20]. No matter the area, effective human-robot collaboration requires the robot to be able to understand and interpret various communication mechanisms similar to those used in human to human interaction [20].

The focus in this work will for the most part be set on the industry field. Current HRC in industry may increase production output, quality and reduce production cost, yet close collaboration between humans and robots has for the most part been prevented due to safety concerns [21]. Over the years, though, safety mechanisms have vastly advanced and there are various different strategies to deal with safety issues. Of course, collisions should generally be prevented, but with the randomness introduced by human behavior, this can often not be guaranteed. Thus, there have to be mechanisms in place to react to collisions [40]. Usually, special sensors (e.g., skin contact, joint torque) are used to detect and react to collisions with humans, but as that limits application of other manipulators that do not have such sensors equipped, Lee, Kim and Song investigated a method to detect these kinds of collisions without the use of such sensors, by identifying the external torques applied to the robot [25]. Robla-Gómez, Becerra, Llata *et al.* discussed multidisciplinary approaches to quantify and estimate the level of injury by collisions [32]. Furthermore, they discussed mechanical (e.g., viscoelastic covering, light-weight structured robots) and software based (collision/contact detection) designs to minimize the consequences of collisions, and also strategies to avoid them (e.g., motion capture systems & simulated environments, vision & range systems) [32].

Collaborative robots (cobots)

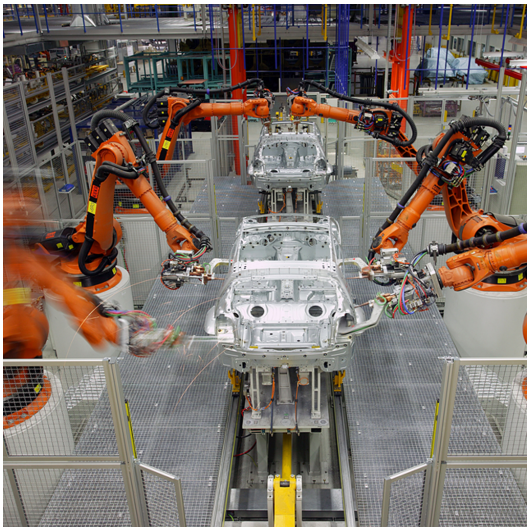
A typical HRC scenario in industry consists of a collaborative task which is performed by a human and a collaborative robot through interactions in a shared workspace [41]. Cobots are a special type of conventional industrial robots. They are usually smaller and much lighter than their counterparts. They are specifically designed to directly interact with humans, which builds a core part of their functionality. This is well reflected in their overall appearance. An example of a typical cobot can be seen in Fig. 1.1b. To prevent injuries like cuts, cobots usually do not have open sharp edges. Additionally, their cabling is integrated, so that nothing nearby can get caught by them. [50]

Due to their light weight, small space requirements and easy programming, cobots

Trait	Conventional industrial robots	Collaborative robots
Flexibility	Low	High
Space requirements	High	Small
Speed & accuracy	High	Lower
Payload	High	Small
Size	From small to big	Small
Weight	Heavy	Lightweight
Working environment	Separated areas	Alongside humans
Initial costs	High	Low
Setup & programming	Complex	Easy

Table 1.1.: Comparison of conventional industrial robots and collaborative robots.

are relatively easy to deploy. As they can take over repetitive tasks from humans, they can boost productivity and quality [75]. A lot of the potentially partially automatable tasks can only be considered for task automation due to the cobots' ability to work alongside humans. Cobots are also rather flexible. They can be easily re-deployed from one task to another [75]. This is especially helpful in the fast-paced Industry 4.0 era, where products can change rather quickly. It also vastly reduces the investment risk [75], which is particularly important for small and medium enterprises (SME).



(a) Industrial robots in an automotive factory.
Source: Photo by KUKA Systems GmbH [16]

(b) Cobot playing chess.
Source: Photo by Pavel Danilyuk [56]

Figure 1.1.: Traditional robot (left) and cobot (right).

Cobots can help to increase job attractiveness [50] and reduce workplace injuries [37], by relieving human operators from physically demanding and tedious, repetitive tasks. Furthermore, working together with cobots can be seen as an innovative task and increase the job's reputation, especially for younger, technology-oriented people [50]. Finally, from a financial perspective, they are usually less expensive compared to traditional industrial robots [50]. This is due to the fact that conventional robots require additional investments, such as safety fences and other equipment such as safety Programmable Logic Controllers (PLC). They also usually require longer integration times and specialized personnel like a robot programmer to set them up.

Research in HRC and cobots has a great focus on technologies that enable human-awareness, like visual perception, action recognition, intention prediction and safe on-line motion planning, resulting in more flexible behavior than conventional action-sequence programs [41], [54]. Another core research topic deals with intuitive cobot programming, which aims to allow non-experts to create and modify cobot programs quick and easy [41], thus providing a huge boost in adaptability and flexibility.

The flexible environments of modern smart production systems require adaptable processes. Workflow Management Systems WfMS provide the functionality to support flexible processes. However, the majority of practical implementations of cobots in industry are usually specialized solutions, designed to interact with specific other systems on the shop floor, but not with other layers in the business, which, in turn, counteracts flexibility. Closing the gap between business information systems and the systems running on the shop floor is a hot topic in research and industry itself (keywords Cyber-Physical System (CPS) and Cyber-Physical Production System (CPPS)).

The overall goal of this thesis is to investigate different patterns in the domain of Human-Robot Collaboration and to explore how such patterns can be used in applications of suitable use cases in combination with a WfMS. The goal is approached by answering the following research questions.

Q1. Which patterns and use cases for the collaboration between humans and collaborative robots can be identified?

Q2. How can such patterns be defined in business processes?

Q3. How to realize those patterns in the context of human-robot collaboration interacting with workflow management systems?

The first research question (Q1) investigates patterns in the collaboration between humans and cobots. This is addressed by reviewing current literature on HRC and cobots, as well as an analysis of use cases found in industry and research. The second question (Q2) is how those patterns can be specified in the business process domain. For this, a formal definition is proposed, which is then used to describe general characteristics of the patterns, along with the specification of some examples from the use case analysis before. An additional focus is set on configurations with multiple cobots and operators, as the majority of research in HRC focuses on scenarios with a single cobot and operator. Yet, various use cases in practice show the necessity to further investigate setups with multiple robotic and human actors. The third question (Q3) deals with the implementation of the identified patterns in the context of Human-Robot Collaboration interacting with Workflow Management Systems. For that purpose, a test setup is developed, which combines a WfMS and a cobot simulation. Subsequently, this setup is utilized to demonstrate the implementation of the patterns.

The outline of this research project is as follows. An introduction to the topic and the basics of Human-Robot Collaboration and collaborative robots was presented in the introduction chapter. Further insights into the background of collaborative robots is given in Chapter 2. An investigation of different use cases, mainly in industry, but also research, follows in Chapter 3. In Chapter 4, a formalization approach of collaborative patterns is proposed. Chapter 5 analyzes the technical aspects of a potential test setup to combine Workflow Management Systems and collaborative use cases. The chapters 6 and 7 deal with the implementation of the setup and the evaluation of the implemented patterns respectively. Finally, Chapter 8 concludes the main findings of this work and gives an outlook on potential future work.

2. Background: Collaborative Robots

2.1. Collaboration scenarios

As there are different ways of collaboration, it is important to distinguish them. This can be done by defining collaboration patterns with specific characteristics. At the outset, it should be mentioned that neither the following classification nor the terminology are unique. In fact, several more or less different definitions can be found in literature [8], [26], [31], [35], [41], [47], [50].

Figures 2.1 and 2.2 show two different yet similar approaches to define such collaboration categories. Both attempts specify four degrees of collaboration, although this number may vary in other literature. In the approach of Müller, Vette and Mailahn [26] (cf. Fig. 2.1), the variants are distinguished by temporal and spatial separation and are called Coexistence, Synchronized, Cooperation and Collaboration.

El Zaatari, Marei, Li *et al.* classified the tasks according to the relation between the cobot, worker, workpieces, and processes being performed. Their naming scheme, as depicted in Fig. 2.2, is Independent, Simultaneous, Sequential and Supportive [41].

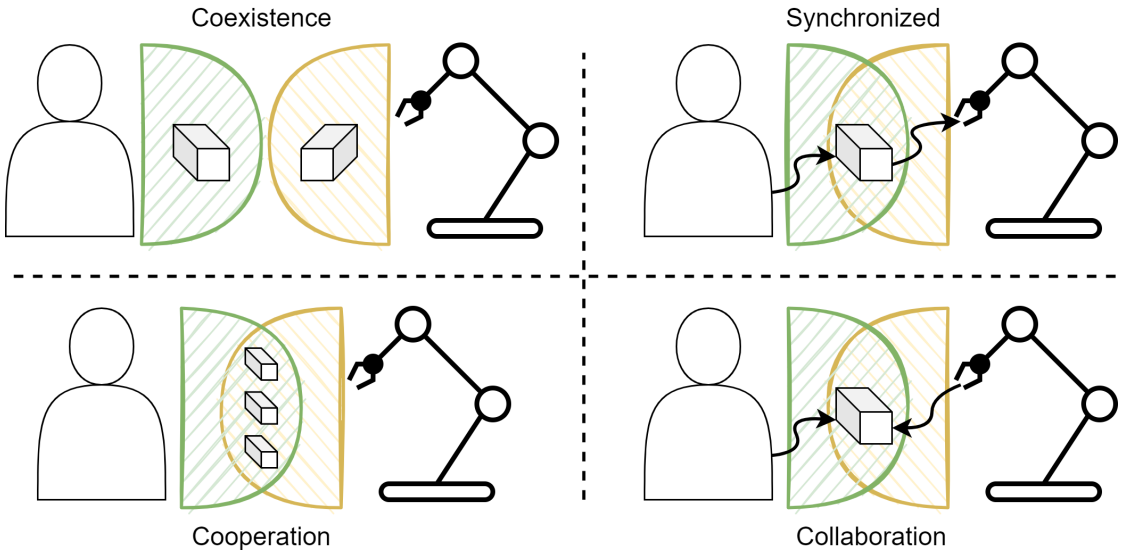


Figure 2.1.: Types of collaboration between humans and cobots based on [26], [43] (own representation).

For simplicity reasons, the approaches of Müller, Vette and Mailahn and El Zaatari, Marei, Li *et al.* are matched and only the former naming (Coexistence, Synchronized,

Cooperation, Collaboration) will be used in the remainder of this paper. As they fit well together the matching used is Coexistence and Independent, Synchronized and Sequential, Cooperation and Simultaneous and finally Collaboration and Supportive.

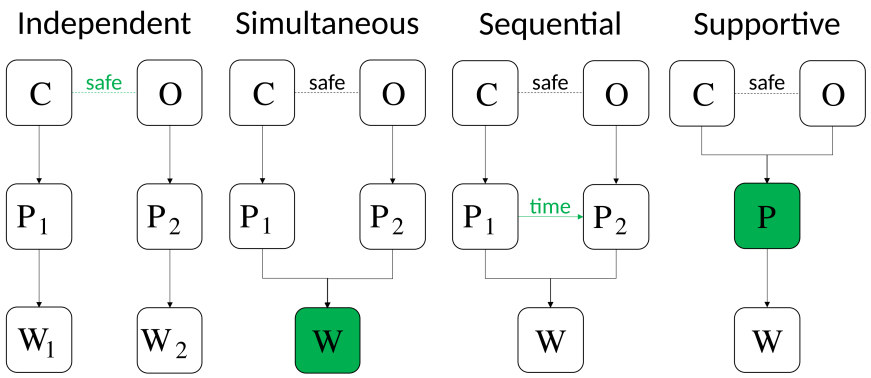


Figure 2.2.: Degrees of collaboration in industrial scenarios based on [41] (redrawn).

2.1.1. Coexistence

The collaborative element in this category is the shared working environment between operator and cobot, without any fences or guards. Other than that, both parties work independently on separate workpieces and individual manufacturing processes [41] and generally do not interact with each other [43]. To achieve the required safety levels, the cobot's intrinsic safety elements are utilized. Additionally, to further enhance safety, external hardware and software safety elements can be added. With the help of these features, the cobot is aware of the workers and acts accordingly to ensure their safety [31], [41].

2.1.2. Synchronized

In synchronized cases, operator and cobot work in the same workspace, but at different times [43]. Both perform their manufacturing tasks on the same workpiece. As these processes are time-dependent from each other, the output of one working step acts as an input for another work step. Cobot and operator each have different tasks to accomplish, but most of the time, the cobot is set up to take over tedious tasks to improve the working conditions for the worker [41].

2.1.3. Cooperation

This category describes an environment where an operator and a cobot perform separate processes on the same workpiece concurrently [31], [41]. Although they are independent of each other in terms of time and tasks, the cobot still needs to respect the worker's space. To ensure this, the cobot has to be spatially aware of the worker and their task requirements [41]. The ability to act concurrently on one workpiece improves productivity and space utilization [31], [41].

2.1.4. Collaboration

Collaboration builds the highest level of collaborative behavior between a worker and a cobot [41]. Both work interactively towards the same process on the same workpiece [41]. Their actions depend on each other. That is, one cannot perform its task without the help of the other [41]. To successfully provide appropriate assistance, the cobot needs to understand the worker's intentions and the task requirements [41]. A simple example for this category is a worker fastening some screws on a workpiece while the cobot holds it in place [41].

2.2. Cobot activities

Cobot activities are considered fundamental functionalities and abilities of cobots. By using and chaining them together, tasks resulting from different use cases can be fulfilled. A few of these activities are described below, although this list certainly is not complete and there are many more, especially in regard to security related functions (e.g., safety-rated monitored stop (SRMS), speed and separation monitoring (SSM), power and force limiting (PFL)) [53].

2.2.1. Positioning

Positioning is a basic functionality of robots in general. Their ability to perform this with extremely high precision, speed and repeatability [33], [49] makes them especially suitable for many tasks in the manufacturing domain, where accuracy and repeatability are key factors for many tasks.

Another variant of movement that can be considered positioning, is to completely move between different locations. This ability enables mobile robots to perform tasks in the intralogistics domain (e.g., automated warehouses, transport of workpieces between different work spaces). Different approaches can be found in literature, like ultrasonic sensors [51] or RFID signal strength sensing [52]. Existing mobile robots are equipped with different tools, like sensors and cameras, in order to avoid collisions [67].

2.2.2. Gripping

Gripping is an essential mechanism, as it enables the cobot to manipulate things in its surrounding in a controlled way. Robotic manipulators can be equipped with end-effectors,

including various types of grippers [45]. Since there are multiple classification types and overlapping terminology, the following four main types are based on the methods to control the gripper itself. Each of them have their advantages and disadvantages, like speed or strength, which influences the applicability in different kinds of use cases.

Vacuum grippers

Vacuum grippers use rubber or foam suction cups to pick up items [45]. The vacuum flow is generated by small pumps and must be uninterrupted to ensure a safe grip of an item. They offer a high level of flexibility, as they only need a plane surface to provide a good grip of an object.

Pneumatic grippers

Pneumatic grippers have the advantage of a compact size and a low weight [45], which makes them suitable for operation in tight spaces. They have two stances, opened and closed, and require compressed air to function. Due to the fact that the dimensions of the object to pick are much more important than in the case of a vacuum gripper, pneumatic grippers are more suitable for handling single part-types.

Hydraulic grippers

Hydraulic grippers are able to produce high amount of forces and hold heavy objects. They also don't need energy supply when standing still, even while holding an object. On the other side, hydraulic gripper systems need constant maintenance [45], due to the complexity of the hydraulic pump system and high forces on the gripper itself.

Servo-electric grippers

Servo-electric grippers offer a high level of flexibility. They can be moved to any position in its motion range, hence they are not limited to one single item type [45]. The movement of the fingers or jaws is controlled by an electric motor. They are easy to control and have low maintenance costs [45]. However, they typically provide less gripping force than the pneumatic variant and are more expensive.

2.2.3. Motion synchronization

The ability to synchronize to other motions is useful in different kinds of tasks. A requirement could be for example to move in tandem with another cobot or follow the motions of a human [11]. Another simple task would be to follow the speed of a conveyor, to be able to handle moving parts [1]. Regardless of the actual case, the cobot needs accurate data of the motion it should synchronize onto. This information can be provided by different means (e.g., encoders, camera systems).

2.2.4. Object recognition

Whenever the task requires dealing with objects in non-standard positions and orientation, some sort of object recognition is required. This is usually performed by additional camera systems that can be stationary but also attached onto the cobot's arm [75]. The latter is particularly suitable for inspection tasks, since this way one camera can be used where otherwise multiple would have been required.

2.2.5. Object re-orientation

In case that objects needs to be placed somewhere in a certain way, but can only be picked up in random orientations, it is necessary to re-orientate the workpiece. In order to manage this task, the cobot needs detailed information about the part it is operating on (e.g., CAD file) [55], [73]. Furthermore, the gripper needs to be flexible as the workpiece has to be gripped in several different ways. As such, vacuum grippers seem to be the best choice for this kind of application.

2.2.6. Constant force application

The ability to apply a constant force onto a workpiece while moving along predefined paths enables use cases such as polishing and sanding. As this is not a trivial mechanism, research shows several approaches on this topic [12], [27], [36].

2.2.7. Hand guiding

In comparison to conventional robots, cobots are easily adaptable to new circumstances. In hand guiding mode, a cobot can be moved manually by a human. This makes easy path teaching and also complex and interactive collaboration possible [41]. A simple example could be that the cobot holds a workpiece while a human inspects it. To view the piece from different angles, the human just moves the cobot into a different position by hand. Such an example is shown in this video[1].

2.2.8. Collision detection/avoidance

Avoiding and detecting collisions[2] is especially important in collaborative scenarios. Different methods exist to deal with the uncertainty a human introduces to the workspace, such as SRMS, SSM and PFL [41]. Additionally, research is done in arm movement intention recognition based on EEG (electroencephalogram) data [54].

Besides that, there are already existing systems that make use of so-called digital twins to avoid collisions and replan paths [57]. A digital twin, as the name suggests, is a copy of the workspace in a virtual environment, including the components and the dynamics of the physical system [57].

[1] `https://youtu.be/dVLIphklXSQ?t=106` (visited on 10th Oct. 2021)
[2] `https://youtu.be/AWbJ9Pbm97s` (visited on 10th Oct. 2021)

2.2.9. Path planning/replanning

As mentioned prior, digital twins in conjunction with camera systems can predict possible collisions[3]. There are successful approaches, to anticipate and avoid collisions in scenarios with varying contents, that use live footage of the scene and produce real-time results [23]. In such cases, the system can plan new paths for the cobot and simulate them using the digital twin before actually performing them [22]. An example can be seen in this video[4].

2.3. Communication types

Classical industrial robots mainly offer communication via buttons or graphical user interfaces. Usually, there is a corded control panel attached to the robot, that offers this functionality. Interactions with cobots, however, aim to be more intuitive by using communication means similar to humans [30]. They can be of direct, but also indirect nature. The direction can be both ways, from human to robot and vice versa. This chapter briefly presents and describes some types of communication.

2.3.1. Speech

Voice communication is a very practical way to interact, as it allows the operator to communicate with the cobot, while performing other tasks where hands and visual contact are needed. In order to provide an easily understandable communication base, a domain specific language has to be developed for each use case [41]. This also facilitates instances where different languages are spoken in a single factory. However, especially in industrial settings, communicating by voice is often not possible due to the noisy environment [30], [34].

2.3.2. Gestures

In contrast to speech communication, gesture recognition is not impaired by noisy environments. Although it might not always be as intuitive as voice communication, a cleverly designed domain specific language can build a very solid gestural communication ground. The used gestures should be kept simple, like thumps up/down, showing different amount of fingers or pointing into a direction (left/right) [34]. El Makrini, Elprama, Van den Bergh *et al.* deployed a cobot named "Walt" in a car manufacturing plant [37]. In this example, the communication was done via gestures and consisted of pointing left/right to indicate the assembly line, showing two or four fingers to set the part-type and thumbs up/down to confirm or deny actions.

[3] `https://youtu.be/tpUX-8if1pY` (visited on 10th Oct. 2021)
[4] `https://youtu.be/dVLIphklXSQ?t=186` (visited on 10th Oct. 2021)

2.3.3. Face recognition

Face recognition is often used to identify the operator, as most likely not every worker is authorized to work with the cobot [29], [34], [37]. As the trend continues towards more human-like robots, actuated robotic heads can be used to communicate with the operator and express emotions, as it is the case with Walt [37]. Hence, face recognition can also be used to locate the operator's head and enable the robot to look into this exact direction for communication purposes.

2.3.4. Fingerprint scanning

In addition to the user identification via face recognition, a fingerprint scanner can be included to further secure the identification process [37]. It may also replace the face recognition completely if none of the additional features of face recognition are required. Fingerprint scanning is also a more reliable system in an industrial setting, as it faces much less variation. The scanned fingers of different workers stay the same, as opposed to faces, which can change depending on the person. For example, growing or shaving beards, adding or removing glasses, or a prominent example during the recent pandemic, wearing masks that cover decent parts of the face. In a private setting, where each device can be readjusted by the users themselves, this might not be a problem at all. But in a setting where many people use the same identification system, which has to be readjusted by different authorized people, this can be a severe problem.

2.3.5. Eye gaze

Humans often use eye gazing to indicate different kinds of intentions. For example, looking in a certain direction might be a signal for another person to look into that direction as well, or pick up an object that lies in this direction. The same can be achieved with a display attached to the cobot. It allows the cobot to communicate intentions (e.g., looking to the left or right) to the user, which leads to increased safety and acceptance [29].

2.3.6. Head motion

A very natural, if not the most natural way of simple communication for showing positive or negative reactions is head movement. The meaning of a head nod and head shake is equivalent across many cultures. So, similar to eye gazing, head movement is ideal to send feedback to the user [29]. A head nod could indicate acceptance, while a head shake indicates refusal [29]. In a collaborative scenario, this can be used to convey reactive feedback to the operator [29].

2.3.7. Haptics

Cobots are equipped with a so-called "compliant" mode, where they move according to forces exerted on them [41]. This enables humans to control the cobot directly with their hands [41]. Furthermore, small taps or pushes into certain directions can also be used as

signals to initiate movements between different predefined positions. This can be utilized, for instance, when the cobot holds a piece to be inspected by a human operator, and the operator wants to switch between different predefined viewing angles, so they can have a direct look at certain parts of it.

2.3.8. Learning

It is possible for a cobot system to learn skills similar to how a human would [41]. Examples for that are learning through demonstrations, trial and error, asking questions and receiving feedback [41]. An operator might also be able to directly influence the cobot during runtime by providing additional data (e.g., answers, feedback, and demonstrations) [41].

2.4. Summary

The bottom line of the research of literature on collaborative robots is that four different patterns of collaboration can be identified in the domain of Human-Robot Collaboration. These patterns are mainly described on the abstraction level of processes and consider the collaboration between a single operator and cobot only. Most of the reviewed literature on HRC focuses on one-to-one scenarios as well. Hence, this research will also consider the possibility of multiple human and robotic actors in the following chapters. Furthermore, this work discusses patterns of collaboration at the abstraction level of tasks within processes rather than processes themselves. This chapter also shows the wide variety of cobot activities and possible ways of communication between cobots and humans, which will also be an aspect in the next chapter.

3. Use cases

Cobots are suited for many different use cases. The fact that they don't need to be separated by safety fences makes them a lot more flexible and affordable in comparison to classical industrial robots [41], as they can be easily inserted into existing production lines without wasting excessive amounts of shop floor space. This chapter provides an overview of possible use cases and identifies which cobot activities and communication types may be used in each case (cf. Table 3.1). Additionally, each use case is classified into possible collaboration scenarios (cf. Table 3.1).
In total, 14 use cases were analyzed:

- Pick-and-place
- Collaborative assembly
- Machine tending
- Packaging & palletizing
- Process tasks (gluing, dispensing, welding)
- Finishing tasks (polishing, grinding, deburring)
- Quality inspection
- Materials handling
- Cobot co-pilot
- Co-manipulation
- Fixture
- Handover
- Illumination
- Robotic surgery

3.1. Pick-and-place

In the most simple version of a pick-and-place task, the cobot picks up workpieces from a specified position and places them into another clearly specified location, which might also involve changing the orientation of the piece [75]. The core action is the handling of the workpiece, rather than any other process being performed on it [75]. In any case, the cobot needs some kind of end-effector, usually a vacuum cup or gripper, to pick up the workpieces [75]. The difficulty of the task can vary from case to case. Vision systems may be needed if the parts are in random positions or orientation [75]. Additionally, the parts could be located on a moving conveyor. In this case, the cobot has to synchronize onto the movement of the conveyor while picking the parts.

Another challenging variation is a bin-picking task, where the workpieces must be picked from a pile inside a container [55]. Universal Robots' ActiNav system [73] uses CAD files of the parts along with a vision system to identify them and be able to change their orientation accordingly [55]. This system can also be used in machine tending scenarios (cf Sec. 3.3) [55].

Pick-and-place tasks are very common in the industry. Very often, they represent a highly repetitive and boring task for human workers. This can lead to reduced efficiency and increased error potential, in case workers are not rotated frequently enough [75]. Depending on the weight of the piece that has to be handled, the task also imposes the risk of injuries through repetitive strain or other accidents [75]. For these reasons, and the repetitive and relatively simple nature of the task, it makes for an excellent choice for a cobot automated task [75].

3.2. Collaborative assembly

In a collaborative assembly scenario, operator and cobot complement each other while working together on an assembly process, hence bringing advantages of humans and robots together. El Makrini, Merckaert, Lefeber *et al.* developed a collaborative architecture, consisting of four modules (face recognition, gesture recognition, human-like robot behavior, and visual inspection) [29]. Their application use case consists of a box that has to be assembled under the guidance of the cobot. The cobot assists by holding the box and handing the correct tools and parts to the human [29]. It also inspects the quality of the assembly (e.g., missing screws) using the visual inspection module [29]. The human task comprises the correct assembly of a plate on the box. For communication purposes, the modules face and gesture recognition as well as human-like robot behavior are used [29]. The latter includes an eye gaze and head motion system, together with a text display, to communicate the cobot's intentions to the operator [29]. A video can be found here[1]. Such advanced cases of collaborative behavior are mostly found in research, and prototypical industrial settings.

[1] `https://youtu.be/0azX0cjgwRc` (visited on 13th Mar. 2022)

3.3. Machine tending

Machine tending is a more specialized variant of a pick-and-place task [75]. Additional coordination with the machines, which are fed with parts, is required (e.g., signals for cycle start and end) [75]. The cobot's task is to pick a part from some sort of feeder configuration and place it into a fixed location in the machine [75]. Depending on the setup, the cobot takes out the workpiece after the machine finished processing the part [75]. Subsequently, the process can be repeated. Without automation, one worker would always have to stand in front of the machine and supply it with individual pieces. Opposed to that, one worker is now able to operate on larger batches of pieces and handle several machines alone [75]. This leads to several improvements like increased productivity, and a more interesting workplace for a human worker [75].

3.4. Packaging & palletizing

Packaging goods is almost always a necessity in the production process and usually a highly repetitive task involving small payloads [75]. Hence, it is ideally suited for cobots. The activities performed by the cobot can vary from case to case. Products could be placed into a shrink-wrapping machine for packaging [75]. They can also be picked from a conveyor and placed inside boxes in a specific way, after which the goods are placed onto pallets for shipping [75]. The use of cobots for such scenarios is also beneficial for productions running high mix low volume, where the rapid change of products is a key requirement [75]. The easy adaptability of cobots, allows for a quick reconfiguration of the application [75].

3.5. Process tasks (gluing, dispensing, welding)

Process tasks typically share the same key characteristics [75]. A tool interacts with the workpiece, while it is moved along a fixed path [75]. Training new employees on these kinds of tasks often takes a considerable amount of time, since there are multiple variables influencing the quality of the outcome [75]. Also, it is difficult for a human to repeat a process that requires minimal variation over longer periods of time, whereas cobots are specially designed for such tasks [75]. In case that new stations are needed, for example to increase the production volume, an additional cobot can be set up relatively quickly, as the programs can just be copied from one cobot to another without the need for major adjustments [75].

3.6. Finishing tasks (polishing, grinding, deburring)

Similar to process tasks, finishing tasks also share some key details. In this case, a specific force has to be applied across the surface of a workpiece in order to remove a certain amount of material [75]. Although the details differ between polishing, grinding, and

deburring, the requirements for the cobot are essentially the same. Performing this kind of tasks entirely manual often requires the worker to apply a large amount of force over longer periods of time [75]. This can generate vibrations which, over time, can lead to injuries [75]. Hence, using cobots for the major workload of finishing tasks not only can improve efficiency, but also reduce injuries [75]. Depending on the use case, it can be beneficial to have the human worker check the quality of the cobot's work (e.g., polishing), and if needed, fix the imperfect spots.

3.7. Quality inspection

Ensuring various quality aspects during and/or after assembly steps or production processes is vital for many products. Looking at the manufacturing of cars, for example, a missing bolt in a security-related part or position could have severe consequences. As humans are prone to errors, automatic quality inspection usually delivers a faster and more reliable result. Depending on the type of process to be inspected, the required camera system, capturing the images, can be quite expensive [75]. A cobot is able to move one camera to several positions, therefore reducing the number of cameras needed [75]. Thus, the costs of such systems can be significantly reduced [75].

3.8. Materials handling

Intralogistics tasks often require a serious amount of working hours. Usually, workers use forklifts and similar vehicles to move material from one point to another or even walk the whole distance. Thereby, the majority of work time is consumed by the transportation itself. An Autonomous mobile robot (AMR) can increase efficiency, by performing the task of transportation autonomously, and therefore, free up time of the personnel. Instead of moving material around, they can focus on planning the resources. In a factory in Spain, two AMRs travel an approximate distance of 100 km each month, which enabled two workers to focus on higher-value tasks [68]. Additional software systems can be deployed to manage the AMRs based on current production needs. This saves even more work time, as required material can be provided automatically. The AMRs themselves are equipped with systems like sensors, laser-scanning technology, and 3D cameras to avoid any kind of collisions [67].

3.9. Cobot co-pilot

The research project with the name ALIAS (Aircrew Labor In-Cockpit Automation System) [28] by Aurora Flight Sciences demonstrates a system that aims to extend the human capabilities and act as a second pilot. ALIAS has demonstrated its abilities by utilizing the auto-landing system of a Boeing 737-800NG simulator to safely land the aircraft in the event of pilot incapacitation [28]. ALIAS has also performed in-flight cockpit procedures numerous times, most recently in a DA42 aircraft [28]. In these demonstrations,

where an onboard safety pilot supervised the system, also a fully automated landing was performed at a simulated site at 3000 feet altitude [28]. Although this project focuses solely on airplanes, it shows the potential of cobots to control systems similarly to humans.

3.10. Co-manipulation

In some domains, a common task is to move loads between two points [10]. Very often, the weight that has to be transported, is too high for a human to lift. Robotic assistance is welcomed in such situations. Teleoperated crane-like robots can be used here, but this approach has disadvantages in terms of costs and space requirements [10]. Also, the human intelligence is a key factor in the positioning task, along with the direct interaction with the environment [10]. Therefore, an approach like robot-assisted load carrying by human operators is a desired solution [10]. A cobot and an operator perform a joint task, that is, transporting and positioning of an object. The human controls the positioning, while the cobot handles the weight of the object and follows the directed movements [41].

3.11. Fixture

The cobot's task is to hold a workpiece in position, while the operator performs a task on it [41]. Depending on the implementation, the preferred position may be influenced by the operator on the fly. There are different approaches possible where this kind of use case applies. One is mounting a piece onto a larger part, where either the piece to mount is too heavy to be held in place and mounted by one operator, or it is just physically impossible for one worker to hold and mount the part simultaneously. For example, the piece needs to be held on one side in a certain position, and has to be mounted from the other side of the part where it is attached to. Another variation is that some work steps have to be performed on the workpiece itself. In this situation, it can be very helpful when the orientation, in which the cobot holds the piece, can be adjusted arbitrarily by pushing the cobot in the desired direction.

Either way, it eases the work of the human actor, and also provides the possibility for efficiency gains by making specific tasks concurrent. An example for that could be the preparation of the next part for mounting, done by the cobot, while the operator grabs the stuff needed for mounting the part.

3.12. Handover

Handing over objects is a common interaction in many scenarios. Humans naturally adapt their behavior according to the receiver, by either verbal communication, or non-verbal communication, (e.g., behavior of arms and hands). Huang, Cakmak and Mutlu [24] identified different coordination strategies. These strategies were used in an experiment with varying task demands of the receiver. The results indicated, that depending on the receiver's task demands, the favorable coordination strategy changes [24]. Furthermore,

the results of the experiment showed differences in subjective and objective measurements, i.e., objectively better strategies were, under certain circumstances (e.g., level of task demand on the human), perceived as less fluent interactions by the human receiver [24].

3.13. Illumination

Many scenarios require some sort of lighting. Most of the time, static lighting from an angle above is sufficient. But there are also situations, in which dynamic lighting from different angles can be a huge advantage to a worker performing his actions. In case of a collaborative HRC scenario, the cobot is equipped with a light source as tool [9]. Its task is to illuminate a part a human operates on, and adjust the angle of the light source to not get blocked by the operator, while avoiding collisions with the operator [9].

3.14. Robotic surgery

Surgery often requires utmost precision. Such is the case in spinal surgery. At the Nottingham Trent University, a system, consisting of two cobots, is being developed, that aims to perform spinal surgery with greater accuracy than current methods [46]. The planned procedure includes a scan of the patient's spine to determine the exact issue [46]. The data is then fed to the system and the surgeon can guide the cobot system into position, where it performs the drilling procedure [46].

3.15. Analysis

For illustration purposes of the use case analysis, a table (cf. Table 3.1) is used. The symbol "✓" denotes a fairly clear assignment or required functionality, while a "+" indicates a conceivable but not essential possibility. It should be noted that this classification is based on a limited number of examples and may not fit every case, as use cases can vary depending on the actual implementation.

As for the collaboration scenarios, a rather clear assignment could be made. Most of the analyzed use cases qualify best for one of them. Of course, there are also examples, where, depending on the concrete implementation, multiple categorizations are possible.

It comes of no surprise that the basic capabilities of positioning and gripping are used in the majority of use cases in Table 3.1. Activities like motion synchronization, object recognition and re-orientation are most suitable in industrial scenarios. For example, working with parts that come in random orientations and locations on a moving conveyor belt, and need to be picked up without stopping the conveyor belt. Applying specific amounts of forces constantly are very specific, but essential for tasks like grinding and polishing. Collision detection and avoidance is vital in all of the examples, as each of them enable humans to operate within the work space of the cobots.

Across the board, it can be observed that the communication types are, for the most part, optional. This makes sense, as some of them (e.g. Face recognition, Fingerprint

scanning) are often used for security aspects, and this might not be necessary in many cases. In the typical industrial use cases that take place in manufacturing environments, speech communication is often not applicable, due to the high ambient noise level. Instead, gestures are mainly used for the communication. Haptics could be used in the majority of examples. It is extremely useful for different tasks (e.g. teaching positions and paths, pushing/tapping the cobot as a signal to move to a different position, getting them out of locations for better access, etc.). The same applies to the hand guiding activity, where the cobot moves according to the direction given by hand.

Table 3.1.: Overview of use cases and their associations with cobot activities, communication types and collaboration scenarios.

		Use cases													
		Pick & place	Collaborative assembly	Machine tending	Packaging & palletizing	Process tasks	Finishing tasks	Quality inspection	Materials handling	Cobot co-pilot	Co-manipulation	Fixture	Handover	Illumination	Robotic surgery
Collaboration scenarios	Coexistence	✓		✓	✓	✓	✓	✓	✓						
	Synchronized				✓	✓	✓	✓	✓						
	Cooperation						✓	✓							
	Collaboration		✓			✓	✓	✓		✓	✓	✓	✓	✓	✓
Cobot activities	Positioning	✓	✓	✓	✓	✓	✓	✓	✓	✓	+	✓	+	✓	✓
	Gripping	✓	✓	✓	✓					✓	✓	✓	✓		
	Motion synchronization	+	+	+	+						✓				+
	Object recognition	+	✓	+	+			✓					+		
	Object re-orientation	+	✓	+	+										
	Constant force application					+	✓								
	Hand guiding	+	+	+	+	+	+	+			✓	✓	✓	✓	✓
	Collision detection/avoidance	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
	Path planning/replanning		✓			✓	✓	✓	✓					✓	
Communication types	Speech		+					+		✓	✓	+	+	+	+
	Gestures	+	✓	+	+	+	+	+			✓	+		+	+
	Face recognition	+	✓	+	+	+	+	+	+		+	+		+	+
	Fingerprint scanning	+	+	+	+	+	+	+	+		+	+		+	+
	Eye gaze		✓					+							
	Head motion		✓					+							
	Haptics	+	✓	+	+	+	+	+	✓	✓	✓	✓	✓	✓	✓
	Learning		+		+	+	+	+			+	+	✓	+	

4. Collaboration patterns

This chapter discusses different patterns of collaboration between human and robotic process actors. Building upon the previous insights, the main elements of collaborative patterns are described. Following that, a formal definition for collaboration patterns is proposed. Subsequently, using the proposed definition, formal requirements for each of the patterns *Synchronized, Cooperation, Collaboration and Coexistence* are defined. Finally, the formal definition is used for the instantiation of exemplary cases for the patterns with varying amounts of process actors.

4.1. Background

A collaborative pattern describes a collaborative working environment where humans and cobots work together in an environment, with the possibility to interact with each other. The main elements of a collaborative pattern are:

- a process
- actors
- a workpiece type
- constraints

This approach of a formal definition focuses on the general architecture of such environments and on the different kinds of constraints between actors and their tasks within a process. The goal of this notation is not to model use cases in complete detail but to provide a clear insight into how and which actors' actions are interdependent.

The following describes the main elements mentioned above, alongside some definitions.

Process

The workflow that is happening in a collaborative working environment is summarized in a process, which is carried out by one or multiple actors and is executed on a specific workpiece type. A process comprises multiple tasks, where each of them is executed by one process actor and has a distinct beginning and end. Moreover, the possible flow of the process is clearly defined. Therefore, this approach uses Petri nets, as defined in [13], for this part of the notation. The transition elements of the Petri net are used to denote the tasks of the process. Tasks describe a simple action of an actor (e.g., pick an object, push a button, move to a position). Each task is performed by exactly one process actor.

Actor

An actor is an executing role in a collaborative pattern. That is, actors are an active part in the execution of processes. An actor can either be a human, called operator, or a cobot. In the HRC domain, at least one human actor, as well as one collaborative robot, are part of a collaborative environment. As presented in Chapter 3, some use cases already indicated that there may also be multiple actors of both types present in a working environment. Either way, the definition needs to account for multiple actors. Hence, a set is used to specify the actors.

Workpiece type

The next part is the workpiece type. Workpieces are the input and/or output of the process. During the process execution, multiple operations may be performed on the workpiece. In this formal definition, the workpiece type is defined as a single element, which is additionally assigned to the process.

Constraints

In a collaborative pattern, tasks of different process actors are principally independent. Hence, their execution order is undefined, and they might also be executed in parallel. This may change as soon as additional constraints are added. Each task can be constrained by other actors' tasks. These constraints can be of different types, leading to different behavior in the task flow.

A constraint is specified by two tasks of different actors, and the type of the constraint. They are used to model different interdependencies between the tasks within a process. In a collaborative working environment, three general types have been identified [41].

The first one is a time-based constraint, where the output of one task is needed as the input of another task. The resulting effect on two tasks of different actors is that it constrains and clearly defines the order in which they are executed.

The second type is driven by spatial requirements, which arise when the order of multiple actors' tasks is of no importance, hence not time-dependent, but the actors need to access the same working area to perform these tasks. Such scenarios aim to speed up cycle times by concurrent task execution, which can be achieved by clever design of the working environment and workflow. Due to the limiting factor of space, tasks may occasionally be interrupted, i.e., the cobot must slow down or stop if a human actor gets too close, or an operator interrupts its task to make room for the cobot.

The third type describes the relation between interactive tasks across actors that rely on each other in order to be executed successfully. This type, called functional constraint, is usually present in highly collaborative environments, where different actors work interactively together to achieve a common goal. Tasks that are connected by this type of constraint have to be executed together. In this sense, they are performed in parallel, with the difference that one cannot be completed without the other. To give a

simple example: a gesture task like thumbs up, performed by an operator, needs to have a matching counterpart, for example, a gesture recognition task on the side of a cobot.

4.2. Definition: Collaborative Pattern

Hence, let the definition of a collaborative pattern *CP be a 7-tuple (N, A, w, actassign, wpassign, tperformer, constr) where*

- *N is a Petri net [13] (P, T, F), depicting a business process, where*
  - *P is a finite set of places*
  - *T is a finite set of transitions, that denotes tasks of a process, such that* $P \cap T = \emptyset$
  - $F \subseteq (P \times T) \cup (T \times P)$ *is a set of directed arcs*
- $A = C \cup O$ *is a finite set of actors, such that* $|A| \geq 1$*, where*
  - C *is a finite set of cobots*
  - O *is a finite set of operators*
  - *such that* $C \cap O = \emptyset$
- w *is a workpiece type*
- $actassign \in A \rightarrow \{N\}$ *is a total surjective function, assigning all actors to the process, with*
  - *domain:* $dom(actassign) = A$
  - *range:* $rng(actassign) = \{N\}$
- $wpassign \in \{w\} \rightarrow \{N\}$ *is a total surjective function, assigning the workpiece type to the process*
- $tperformer \in T \nrightarrow A$ *is a partial surjective function, assigning transitions to actors, with*
  - $dom(tperformer) \subseteq T$
  - $rng(tperformer) = A$
- $constr \in T \times T \nrightarrow \{time, space, func\} : (t_a, t_b) \mapsto constr(t_a, t_b)$ *is a partial function denoting the type of constraint between two transitions (tasks), such that*
  - $dom(constr) \subseteq T \times T$
  - $rng(constr) \subseteq \{time, space, func\}$
  - $(t_a, t_b) \in dom(constr) : t_a \neq t_b$ *the constrained tasks are different*
  - $(t_a, t_b) \in dom(constr) : tperformer(t_a) \neq tperformer(t_b)$ *the constrained tasks have different task performers*

- $\{time, space, func\}$ *denote the type of constraint (e.g., timely (time), spatially (space), and functionally (func) constrained).*
  - *timely constrained:* $constr(t_a, t_b) = time \implies t_b$ *is timely constrained by* t_a*, i.e.,* t_b *must be executed after* t_a
  - *spatially constrained:* $constr(t_a, t_b) = space \implies t_a$ *and* t_b *are spatially constrained, i.e., the order of execution is undefined and potentially concurrent, and task interruptions due to space limitations are possible*
  - *functionally constrained:* $constr(t_a, t_b) = func \implies t_a$ *and* t_b *are functionally constrained, i.e.,* t_a *and* t_b *must be executed together to complete the operation*

In the following, the collaborative patterns *Synchronized, Cooperative, Collaboration* and *Coexistence* are described using this definition. Each of them starts with a brief description, followed by the formal requirements of the pattern.

4.2.1. Collaborative Pattern: Synchronized

Synchronized use cases typically share a few key features. Actors perform their tasks in a shared workspace on the same workpiece type. An important characteristic is that these tasks are time-constrained, which implies two things. First, the constrained tasks are executed at different times, and second, one of them requires the output of the corresponding task in order to be executed. Therefore, time constraints are a characterization of the synchronized pattern.
A synchronized collaborative pattern must meet certain requirements:
$CP_{synch} = (N_i, A_i, w_i, actassign_i, wpassign_i, tperformer_i, constr_i)$

- $|C_i| \geq 1 \ \wedge \ |O_i| \geq 1$ there is at least one cobot and at least one operator
- $\forall constr_{i_m} \in constr_i : constr_{i_m}(t_{i_j}, t_{i_k}) = time$ all constraints between transitions are time constraints

4.2.2. Collaborative Pattern: Cooperation

Cooperative scenarios are, in a certain way, similar to synchronized ones, as the actors perform their work on the same type of workpiece. The key difference is, that these steps are not timely but spatially constrained. That is, the order in which they are performed does not matter. These use cases aim to execute tasks concurrently, thus, increasing efficiency and productivity. The fact that actors may need to operate in each other's workspace simultaneously demands for additional requirements, especially regarding safety aspects. As a result, the cobots need to have mechanisms to be aware of other actors and protect them. This is especially important when human actors are involved. Furthermore, there is also usually less interaction between actors than in synchronized cases, if any at all.
A cooperative collaborative pattern must meet certain requirements:
$CP_{coop} = (N_i, A_i, w_i, actassign_i, wpassign_i, tperformer_i, constr_i)$

- $|C_i| \geq 1 \ \wedge \ |O_i| \geq 1$ there is at least one cobot and at least one operator
- $\forall constr_{i_m} \in constr_i : constr_{i_m}(t_{i_j}, t_{i_k}) = space$ all constraints between transitions are space constraints

4.2.3. Collaborative Pattern: Collaboration

Collaborative cases are designed with interactive behavior in mind. The actors are working together to achieve a common goal. Parts of their actions are functionally dependent on some tasks of other actors. That is, those tasks rely on each other to be executed successfully. This type of scenario represents the most comprehensive type of collaboration. In theory, the cobot should act instinctively, based on the behavior of human actors, e.g., repositioning of the axes to make it easier for the operator to access certain locations when approaching them, or monitoring the environment to anticipate and avoid collisions, and re-plan trajectories [22], [23].
A collaborative pattern of type collaboration must meet certain requirements:
$CP_{coll} = (N_i, A_i, w_i, actassign_i, wpassign_i, tperformer_i, constr_i)$

- $|C_i| \geq 1 \ \wedge \ |O_i| \geq 1$ there is at least one cobot and at least one operator
- $\forall constr_{i_m} \in constr_i : constr_{i_m}(t_{i_j}, t_{i_k}) = func$ all constraints between transitions are functional constraints

4.2.4. Collaborative Pattern: Coexistence

Working environments that fit in the coexistence pattern have specific characteristics as well. They comprise different types of workpieces and separate processes that are performed on those workpiece types. Different actors execute these processes. Hence, they are operating independently of each other. The difference to classic industrial settings is that there is the possibility for actors to access the workspace of other actors. That is, safety fences and the like are not mandatory, which leads to improved space utilization of the shop floor.
A coexistent collaborative pattern must meet certain requirements:
$CP_{coex} = \{CP_1, \ldots, CP_n\} = \{(N_1, A_1, w_1, actassign_1, wpassign_1, tperformer_1, constr_1), \ldots, (N_n, A_n, w_n, actassign_n, wpassign_n, tperformer_n, constr_n)\}$

- $\forall CP_j, CP_k \in CP_{coex} : CP_j \neq CP_k \implies CP_j \cap CP_k = \emptyset$ all CP_j, CP_k in CP_{coex} are pairwise disjoint
- $\forall CP_i \in CP_{coex} : |A_i| = 1$ there is exactly one actor in every CP_i in CP_{coex}
- $\left| \bigcup_{i=1}^{n} C_i \right| \geq 1$ there is at least one cobot in all C_i in CP_{coex}
- $\left| \bigcup_{i=1}^{n} O_i \right| \geq 1$ there is at least one operator in all O_i in CP_{coex}
- $\forall CP_i \in CP_{coex} : constr_i = \emptyset$ there are no constraints in every CP_i in CP_{coex}

4.3. Instantiation of patterns

In this section, exemplary instantiations of the previously discussed patterns are shown. These instantiations are loosely based on the use cases presented in Chapter 3. The formalizations of these examples use the definition from Section 4.2. Each of them starts with a brief description, followed by a formal representation. It should be noted that these cases are purely exemplary, as their whole purpose is to showcase the approach. Hence, they are very simplified versions of real-world examples. One further note on the naming (e.g., $CP_{synch_{1O1C}}$). The index "$xOyC$" indicates the number of operators (O) and cobots (C) present in the example.

4.3.1. Synchronized

Assembly: one operator and one cobot

The formalization below describes a configuration with two actors, the operator and the cobot, performing a process on a type of workpiece. That is, the assembly of an insert into the workpiece. Both actors start with an independent task. The cobot picks the insert (t_{c_1}), and meanwhile, the operator prepares the workpiece (t_{o_1}). The cobot's second task (placement of the insert t_{c_2}), though, is constrained by the first task of the operator. Similarly, the second operator task (checking of the assembled insert t_{o_2}) needs the output of the third and final cobot task (fixing the insert t_{c_3}). After that, the process finishes.

Formalization Let the synchronized collaborative pattern be defined by $CP_{synch_{1O1C}} = (N, A, w, actassign, wpassign, tperformer, constr)$ with

- $P = \{p_1, p_2, p_3, p_4, p_5, p_6, p_7, p_8, p_9\}$
- $T = \{t_1, t_2, t_{c_1}, t_{c_2}, t_{c_3}, t_{o_1}, t_{o_2}\}$
- $F = \{(p_1, t_1), (t_1, p_2), (t_1, p_6), (p_5, t_2), (p_8, t_2), (t_2, p_9),$
  $(p_2, t_{c_1}), (t_{c_1}, p_3), (p_3, t_{c_2}), (t_{c_2}, p_4), (p_4, t_{c_3}), (t_{c_3}, p_5),$
  $(p_6, t_{o_1}), (t_{o_1}, p_7), (p_7, t_{o_2}), (t_{o_2}, p_8)\}$
- $C = \{c\}, \quad O = \{o\}$
- $actassign = \{(c, N), (o, N)\}, \quad wpassign = \{(w, N)\}$
- $tperformer = \{(t_{c_1}, c), (t_{c_2}, c), (t_{c_3}, c), (t_{o_1}, o), (t_{o_2}, o)\}$
- $constr = \{((t_{o_1}, t_{c_2}), time), ((t_{c_3}, t_{o_2}), time)\}$

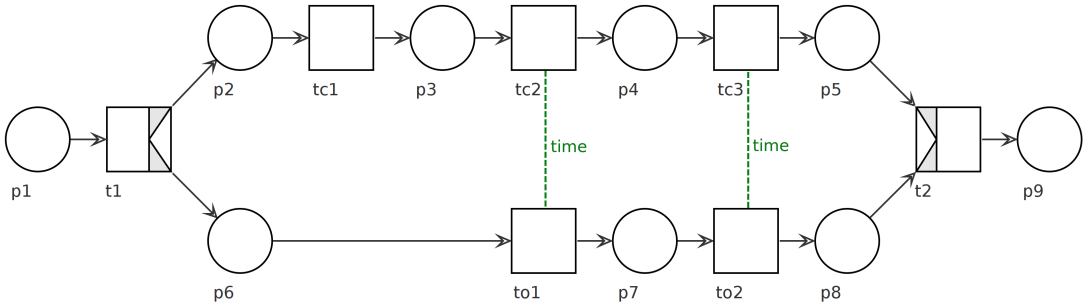


Figure 4.1.: Petri net N of the Assembly use case. The time constraints of the pattern are highlighted with green dashed lines.

Spot taping: one operator and two cobots

The following formalization describes a case where a process needs various steps to finish a workpiece. Furthermore, the process is carried out in tandem by the operator (o) and two cobots (c_1, c_2). This allows the operator two work on two samples of the workpiece type (w) at once, almost doubling the productivity. This Spot taping use case will be discussed in more detail in Section 7.1.1.

In summary, two work steps have to be performed on each workpiece. These steps ($t_{c_{1_1}}, t_{c_{1_2}}, t_{c_{2_1}}, t_{c_{2_2}}$) are done by the two cobots. Prior to performing those work steps, the workpieces have to be prepared. This is done manually by the operator ($t_{o_1}, t_{o_2}, t_{o_3}, t_{o_4}$). Once the parts are finished, the operator removes them (t_{o_5}, t_{o_6}).

The tasks have to be performed in a specific order, i.e., they are time-constrained. Operator o starts with t_{o_1}. After that, c_1 can perform $t_{c_{1_1}}$. This relation is modeled with a time constraint $((t_{o_1}, t_{c_{1_1}}), time)$. In the meantime, t_{o_2} can be performed, and after that $t_{c_{2_1}}$. Once $t_{c_{1_1}}$ is finished, the operator can start with t_{o_3}. This is systematically repeated towards the end of the process, and is reflected in the function *constr* of the formalization below.

Formalization Let the synchronized collaborative pattern be defined by $CP_{synch_{1O2C}} = (N, A, w, actassign, wpassign, tperformer, constr)$ with

- $P = \{p_1, p_2, p_3, p_4, p_5, p_6, p_7, p_8, p_9, p_{10}, p_{11}, p_{12}, p_{13}, p_{14}, p_{15}\}$
- $T = \{t_1, t_2, t_{c_{1_1}}, t_{c_{1_2}}, t_{o_1}, t_{o_2}, t_{o_3}, t_{o_4}, t_{o_5}, t_{o_6}, t_{c_{2_1}}, t_{c_{2_2}}\}$
- $F = \{(p_1, t_1), (t_1, p_2), (t_1, p_5), (t_1, p_{12}),$
  $(p_2, t_{c_{1_1}}), (t_{c_{1_1}}, p_3), (p_3, t_{c_{1_2}}), (t_{c_{1_2}}, p_4),$
  $(p_5, t_{o_1}), (t_{o_1}, p_6), (p_6, t_{o_2}), (t_{o_2}, p_7), (p_7, t_{o_3}), (t_{o_3}, p_8),$
  $(p_8, t_{o_4}), (t_{o_4}, p_9), (p_9, t_{o_5}), (t_{o_5}, p_{10}), (p_{10}, t_{o_6}), (t_{o_6}, p_{11}),$
  $(p_{12}, t_{c_{2_1}}), (t_{c_{2_1}}, p_{13}), (p_{13}, t_{c_{2_2}}), (t_{c_{2_2}}, p_{14}),$
  $(p_4, t_2), (p_{11}, t_2), (p_{14}, t_2), (t_2, p_{15})\}$

- $C = \{c_1, c_2\}, \quad O = \{o\}$
- $actassign = \{(c_1, N), (o, N), (c_2, N)\}, \quad wpassign = \{(w, N)\}$
- $tperformer = \{(t_{c_{1_1}}, c_1), (t_{c_{1_2}}, c_1), (t_{o_1}, o), (t_{o_2}, o), (t_{o_3}, o), (t_{o_4}, o), (t_{o_5}, o), (t_{o_6}, o), (t_{c_{2_1}}, c_2), (t_{c_{2_2}}, c_2)\}$
- $constr = \{((t_{o_1}, t_{c_{1_1}}), time), ((t_{c_{1_1}}, t_{o_3}), time), ((t_{o_3}, t_{c_{1_2}}), time), ((t_{c_{1_2}}, t_{o_5}), time), ((t_{o_2}, t_{c_{2_1}}), time), ((t_{c_{2_1}}, t_{o_4}), time), ((t_{o_4}, t_{c_{2_2}}), time), ((t_{c_{2_2}}, t_{o_6}), time)\}$

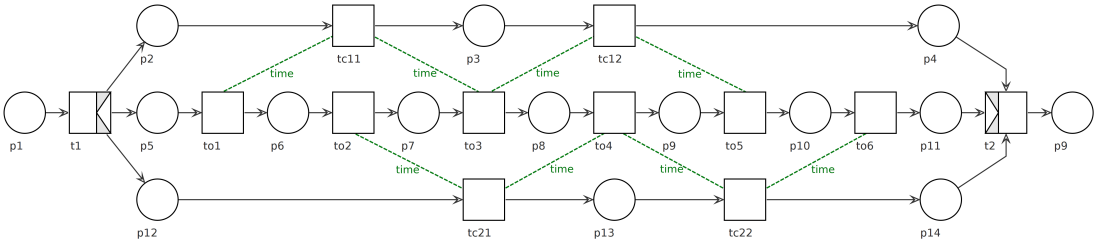


Figure 4.2.: Petri net N of the Spot taping use case. The time constraints of the pattern are highlighted with green dashed lines.

4.3.2. Cooperation

Polishing: one operator and one cobot

A fitting example for the cooperation pattern with one operator and one cobot is a polishing process of a decently sized workpiece type. The cobot's task here is the main polishing work. Meanwhile, the operator is looking for areas with greater defects that may need some prework. Additionally, the operator performs fine-polishing where needed. In more detail, the cobot starts by moving into the starting position (t_{c_1}), followed by the main polishing (t_{c_2}), and finally moving back into its home position (t_{c_3}). In parallel, the operator performs the inspection and preparatory work (t_{o_1}) and then starts the fine-polishing (t_{o_2}).

Formalization Let the cooperative collaborative pattern be defined by $CP_{coop_{1O1C}} = (N, A, w, actassign, wpassign, tperformer, constr)$ with

- $P = \{p_1, p_2, p_3, p_4, p_5, p_6, p_7, p_8, p_9\}$
- $T = \{t_1, t_2, t_{c_1}, t_{c_2}, t_{c_3}, t_{o_1}, t_{o_2}\}$
- $F = \{(p_1, t_1), (t_1, p_2), (t_1, p_6), (p_5, t_2), (p_8, t_2), (t_2, p_9), (p_2, t_{c_1}), (t_{c_1}, p_3), (p_3, t_{c_2}), (t_{c_2}, p_4), (p_4, t_{c_3}), (t_{c_3}, p_5), (p_6, t_{o_1}), (t_{o_1}, p_7), (p_7, t_{o_2}), (t_{o_2}, p_8)\}$
- $C = \{c\}, \quad O = \{o\}$

- $actassign = \{(c, N), (o, N)\}, \quad wpassign = \{(w, N)\}$
- $tperformer = \{(t_{c_1}, c), (t_{c_2}, c), (t_{c_3}, c), (t_{o_1}, o), (t_{o_2}, o)\}$
- $constr = \{((t_{c_1}, t_{o_1}), space), ((t_{o_1}, t_{c_2}), space), ((t_{c_2}, t_{o_2}), space), ((t_{o_2}, t_{c_3}), space)\}$

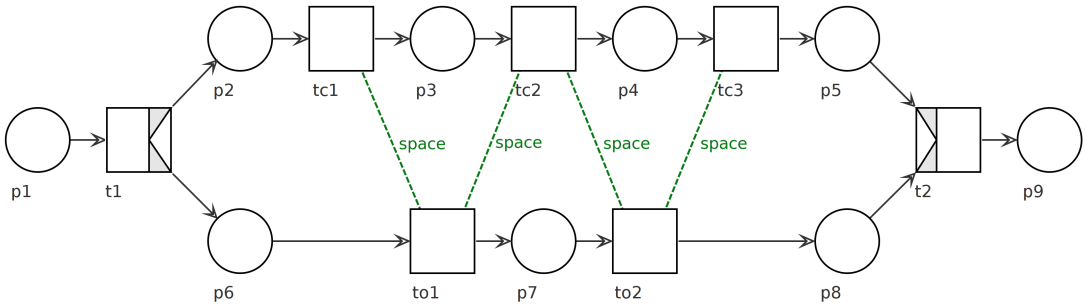


Figure 4.3.: Petri net N of the Polishing use case. The space constraints of the pattern are highlighted with green dashed lines.

Quality inspection: two operators and two cobots

Here, a quality inspection use case is defined that consists of four actors, namely two operators (o_1, o_2) and two cobots (c_1, c_2). The setting requires the inspection of eight different locations, two of them for each actor. It is designed so that one of the locations of each cobot overlaps with one inspection location of an operator. Also, one of the operator's inspection locations are overlapping with each other. In general, the order in which these locations are inspected does not matter. However, to increase efficiency, tasks are performed in a specific order that minimizes or eliminates overlapping of spatially constrained tasks.

Therefore, cobot c_1 starts with its unconstrained task $(t_{c_{1_1}})$, while operator o_1 is performing $t_{o_{1_1}}$, which is spatially constrained by $t_{c_{1_2}}$. Concurrently, operator o_2 performs $t_{o_{2_1}}$, which is spatially constrained by $t_{o_{1_2}}$, and c_2 works on $t_{c_{2_1}}$, which is constrained by $t_{o_{2_2}}$. Subsequently, each of the actors performs their other remaining task. This procedure effectively prevents unnecessary overlaps, which in turn saves time and reduces complexity.

Formalization Let the cooperative collaborative pattern be defined by $CP_{coop_{2O2C}} = (N, A, w, actassign, wpassign, tperformer, constr)$ with

- $P = \{p_1, p_2, p_3, p_4, p_5, p_6, p_7, p_8, p_9, p_{10}, p_{11}, p_{12}, p_{13}, p_{14}\}$
- $T = \{t_1, t_2, t_{c_{1_1}}, t_{c_{1_2}}, t_{o_{1_1}}, t_{o_{1_2}}, t_{c_{2_1}}, t_{c_{2_2}}, t_{o_{2_1}}, t_{o_{2_2}}\}$

- $F = \{(p_1, t_1), (t_1, p_2), (t_1, p_5), (t_1, p_8), (t_1, p_{11}),$
  $(p_2, t_{c_{1_1}}), (t_{c_{1_1}}, p_3), (p_3, t_{c_{1_2}}), (t_{c_{1_2}}, p_4),$
  $(p_5, t_{o_{1_1}}), (t_{o_{1_1}}, p_6), (p_6, t_{o_{1_2}}), (t_{o_{1_2}}, p_7),$
  $(p_8, t_{o_{2_1}}), (t_{o_{2_1}}, p_9), (p_9, t_{o_{2_2}}), (t_{o_{2_2}}, p_{10}),$
  $(p_{11}, t_{c_{2_1}}), (t_{c_{2_1}}, p_{12}), (p_{12}, t_{c_{2_2}}), (t_{c_{2_2}}, p_{13}),$
  $(p_4, t_2), (p_7, t_2), (p_{10}, t_2), (p_{13}, t_2), (t_2, p_{14})\}$
- $C = \{c_1, c_2\}, \quad O = \{o_1, o_2\}$
- $actassign = \{(c_1, N), (c_2, N), (o_1, N), (o_2, N)\}, \quad wpassign = \{(w, N)\}$
- $tperformer = \{(t_{c_{1_1}}, c_1), (t_{c_{1_2}}, c_1), (t_{c_{2_1}}, c_2), (t_{c_{2_2}}, c_2), (t_{o_{1_1}}, o_1), (t_{o_{1_2}}, o_1),$
  $(t_{o_{2_1}}, o_2), (t_{o_{2_2}}, o_2)\}$
- $constr = \{((t_{c_{1_2}}, t_{o_{1_1}}), space), ((t_{o_{1_2}}, t_{o_{2_1}}), space), ((t_{o_{2_2}}, t_{c_{2_1}}), space)\}$

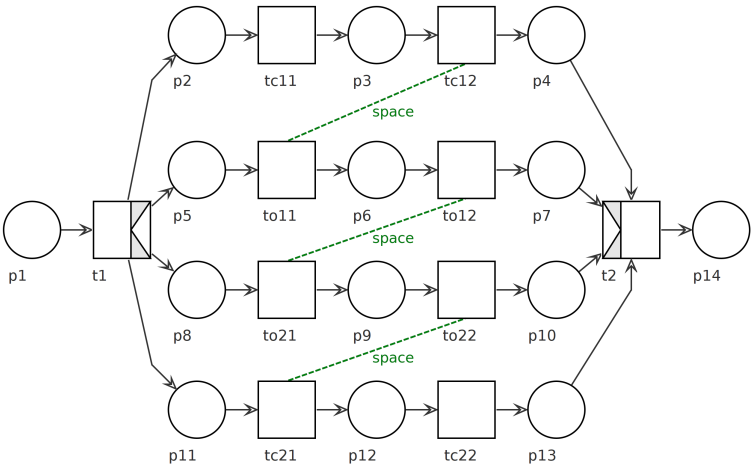


Figure 4.4.: Petri net N of the Quality inspection use case. The space constraints of the pattern are highlighted with green dashed lines.

4.3.3. Collaboration

Collaborative assembly: one operator and one cobot

A collaborative assembly use case depicts the collaborative pattern with a single operator and cobot. In this simplified example, the cobot leads the process and guides the operator through it. First, it grabs the toolbox (t_{c_1}) and gives the operator access to it (t_{c_2}). The operator then grabs the required tools (t_{o_1}), which are provided by the cobot. Afterwards, the cobot puts the toolbox aside (t_{c_3}), and then grabs (t_{c_4}) and holds the assembly part in an appropriate orientation (t_{c_5}), so that the operator is able to perform the assembly (t_{o_2}). Finally, the cobot puts the assembled workpiece in its place (t_{c_6}).

Formalization Let the collaborative pattern of type collaboration be defined by $CP_{coll_{1O1C}} = (N, A, w, actassign, wpassign, tperformer, constr)$ with

- $P = \{p_1, p_2, p_3, p_4, p_5, p_6, p_7, p_8, p_9, p_{10}, p_{11}, p_{12}\}$
- $T = \{t_1, t_2, t_{c_1}, t_{c_2}, t_{c_3}, t_{c_4}, t_{c_5}, t_{c_6}, t_{o_1}, t_{o_2}\}$
- $F = \{(p_1, t_1), (t_1, p_2), (t_1, p_9), (p_8, t_2), (p_{11}, t_2), (t_2, p_{12}),$
  $(p_2, t_{c_1}), (t_{c_1}, p_3), (p_3, t_{c_2}), (t_{c_2}, p_4), (p_4, t_{c_3}), (t_{c_3}, p_5),$
  $(p_5, t_{c_4}), (t_{c_4}, p_6), (p_6, t_{c_5}), (t_{c_5}, p_7), (p_7, t_{c_6}), (t_{c_6}, p_8),$
  $(p_9, t_{o_1}), (t_{o_1}, p_{10}), (p_{10}, t_{o_2}), (t_{o_2}, p_{11})\}$
- $C = \{c\}, \quad O = \{o\}$
- $actassign = \{(c, N), (o, N)\}, \quad wpassign = \{(w, N)\}$
- $tperformer = \{(t_{c_1}, c), (t_{c_2}, c), (t_{c_3}, c), (t_{c_4}, c), (t_{c_5}, c), (t_{c_6}, c), (t_{o_1}, o), (t_{o_2}, o)\}$
- $constr = \{((t_{c_2}, t_{o_1}), func), ((t_{c_5}, t_{o_2}), func)\}$

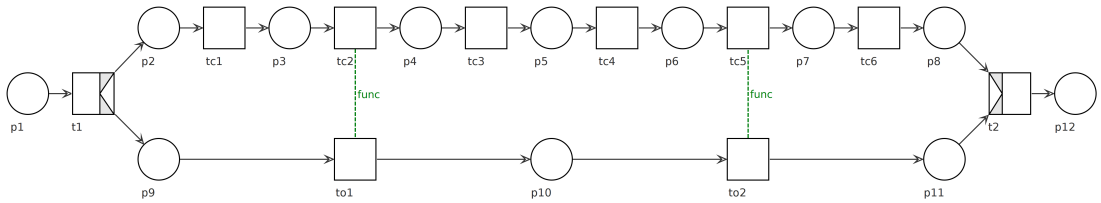


Figure 4.5.: Petri net N of the Collaborative assembly use case. The functional constraints of the pattern are highlighted with green dashed lines.

Assisted handling: one operator and two cobots

For a collaborative case where one operator (o_1) utilizes two cobots (c_1, c_2), the handling of a heavy workpiece can be looked at. The workpiece type in this example is a bumper (w), that needs to be attached to the frame of a car body. The two cobots grab the part on both ends ($t_{c_{1_1}}, t_{c_{2_1}}$). The operator's task is to guide the part to the correct location (t_{o_1}) and then perform the assembly (t_{o_2}). The cobots hold the weight of the object and follow the operator's movements ($t_{c_{1_2}}, t_{c_{2_2}}$). The way the operator guides the cobots is by simply pushing or pulling the workpiece into the desired direction. After the assembly is finished, the cobots move back into their home positions ($t_{c_{1_3}}, t_{c_{2_3}}$).

Formalization Let the collaborative pattern of type collaboration be defined by $CP_{coll_{1O2C}} = (N, A, w, actassign, wpassign, tperformer, constr)$ with

- $P = \{p_1, p_2, p_3, p_4, p_5, p_6, p_7, p_8, p_9, p_{10}, p_{11}, p_{12}, p_{13}\}$

- $T = \{t_1, t_2, t_{c1_1}, t_{c1_2}, t_{c1_3}, t_{c2_1}, t_{c2_2}, t_{c2_3}, t_{o1}, t_{o2}\}$
- $F = \{(p_1, t_1), (t_1, p_2), (t_1, p_6), (t_1, p_9),$
  $(p_2, t_{c1_1}), (t_{c1_1}, p_3), (p_3, t_{c1_2}), (t_{c1_2}, p_4), (p_4, t_{c1_3}), (t_{c1_3}, p_5),$
  $(p_6, t_{o1}), (t_{o1}, p_7), (p_7, t_{o2}), (t_{o2}, p_8),$
  $(p_9, t_{c2_1}), (t_{c2_1}, p_{10}), (p_{10}, t_{c2_2}), (t_{c2_2}, p_{11}), (p_{11}, t_{c2_3}), (t_{c2_3}, p_{12}),$
  $(p_5, t_2), (p_8, t_2), (p_{12}, t_2), (t_2, p_{13})\}$
- $C = \{c_1, c_2\}, \quad O = \{o\}$
- $actassign = \{(c_1, N), (c_2, N), (o, N)\}, \quad wpassign = \{(w, N)\}$
- $tperformer = \{(t_{c1_1}, c_1), (t_{c1_2}, c_1), (t_{c1_3}, c_1), (t_{c2_1}, c_2), (t_{c2_2}, c_2), (t_{c2_3}, c_2),$
  $(t_{o1}, o), (t_{o2}, o)\}$
- $constr = \{((t_{o1}, t_{c1_2}), func), ((t_{o1}, t_{c2_2}), func), ((t_{o2}, t_{c1_2}), func), ((t_{o2}, t_{c2_2}), func)\}$

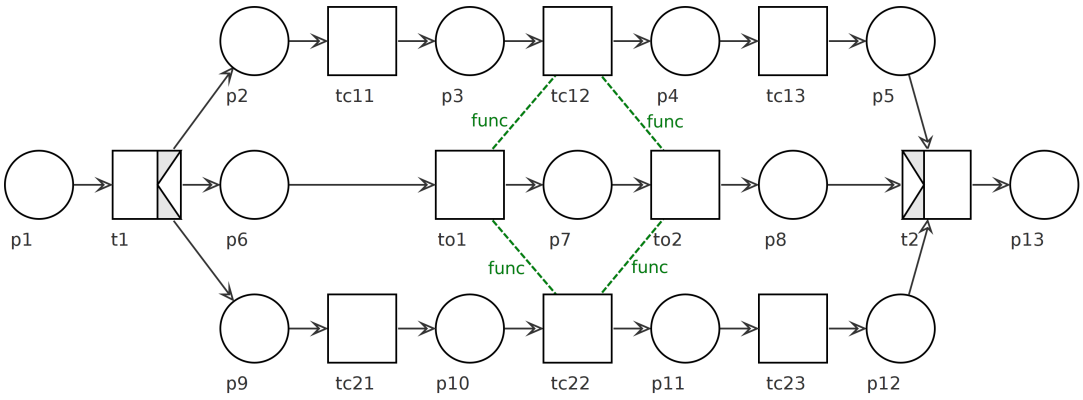


Figure 4.6.: Petri net N of the Assisted handling use case. The functional constraints of the pattern are highlighted with green dashed lines.

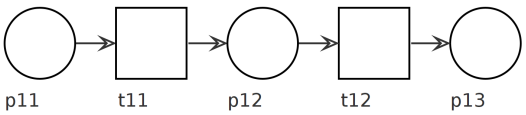
4.3.4. Coexistence

Pick & Place: one operator and one cobot

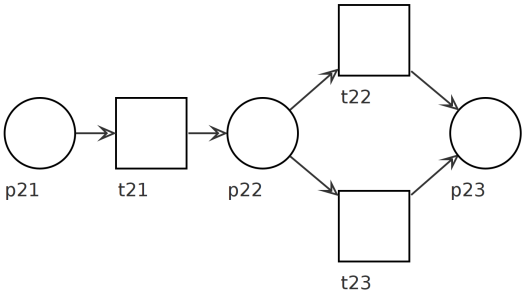
A simple pick and place use case demonstrates the coexistence pattern with a single operator and cobot. The workpiece types are parts that need to be picked from one location and then placed into another location (w_1), and boxes containing those parts (w_2). The two processes are: the picking (t_{1_1}) and placing (t_{1_2}) of the single parts (N_1), which is performed by the cobot, and the exchange of the part boxes (t_{2_2}) once they are empty (N_2), executed by the operator. In case the operator gets called but the checking of the box (t_{2_1}) reveals that it still contains parts, the operator's task is to determine and fix the underlying problem (t_{2_3}).

Formalization Let the coexistent collaborative pattern be defined by
$CP_{coex_{1O1C}} = \{CP_1, CP_2\} =$
$\{(N_1, A_1, w_1, actassign_1, wpassign_1, tperformer_1, constr_1),$
$(N_2, A_2, w_2, actassign_2, wpassign_2, tperformer_2, constr_2)\}$ with

- $P_1 = \{p_{1_1}, p_{1_2}, p_{1_3}\}, \quad T_1 = \{t_{1_1}, t_{1_2}\}$
- $F_1 = \{(p_{1_1}, t_{1_1}), (t_{1_1}, p_{1_2}), (p_{1_2}, t_{1_2}), (t_{1_2}, p_{1_3})\}$
- $C_1 = \{c\}, \quad O_1 = \emptyset$
- $actassign_1 = \{(c, N_1)\}, \quad wpassign_1 = \{(w_1, N_1)\}$
- $tperformer_1 = \{(t_{1_1}, c), (t_{1_2}, c)\}, \quad constr_1 = \emptyset$
- $P_2 = \{p_{2_1}, p_{2_2}, p_{2_3}\}, \quad T_2 = \{t_{2_1}, t_{2_2}, t_{2_3}\}$
- $F_2 = \{(p_{2_1}, t_{2_1}), (t_{2_1}, p_{2_2}), (p_{2_2}, t_{2_2}), (p_{2_2}, t_{2_3}), (t_{2_2}, p_{2_3}), (t_{2_3}, p_{2_3})\}$
- $C_2 = \emptyset, \quad O_2 = \{o\}$
- $actassign_2 = \{(o, N_2)\}, \quad wpassign_2 = \{(w_2, N_2)\}$
- $tperformer_2 = \{(t_{2_1}, o), (t_{2_2}, o), (t_{2_3}, o)\}, \quad constr_2 = \emptyset$



(a) Petri net N_1 depicts the process performed by the cobot c.



(b) Petri net N_2 depicts the process performed by the operator o.

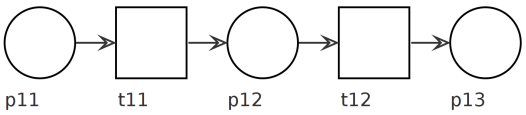
Figure 4.7.: Petri nets N_1, N_2 of the Pick & Place use case.

Packaging & Palletizing: two operators and one cobot

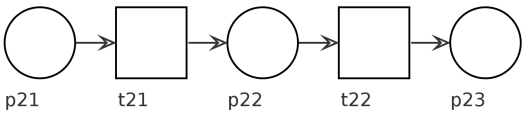
In the following case, two operators and one cobot are present. The use case consists of packaging smaller goods into boxes (N_1), palletizing those boxes (N_2), and finally transporting the pallets into their desired location (N_3). Hence, there are three types of workpieces (parts w_1, boxes w_2, pallets w_3). Operator o_1 grabs (t_{1_1}) and puts (t_{1_2}) specific amounts of parts into boxes, which are then buffered up to a certain amount. This way the operator's workflow does not get interrupted that easily, if the cobot has to wait for a new pallet, for example. Cobot c then palletizes those boxes onto pallets (t_{2_1}, t_{2_2}). Finally, operator o_2 fixates the stapled boxes on the pallets (t_{3_1}) and transports them to another location (t_{3_2}).

Formalization Let the coexistent collaborative pattern be defined by
$CP_{coex_{2O1C}} = \{CP_1, CP_2, CP_3\} =$
$\{(N_1, A_1, w_1, actassign_1, wpassign_1, tperformer_1, constr_1),$
$(N_2, A_2, w_2, actassign_2, wpassign_2, tperformer_2, constr_2),$
$(N_3, A_3, w_3, actassign_3, wpassign_3, tperformer_3, constr_3)\}$ with

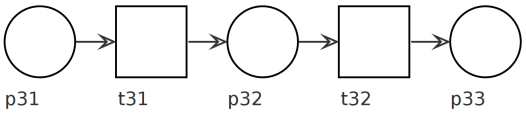
- $P_1 = \{p_{1_1}, p_{1_2}, p_{1_3}\}, \quad T_1 = \{t_{1_1}, t_{1_2}\}$
- $F_1 = \{(p_{1_1}, t_{1_1}), (t_{1_1}, p_{1_2}), (p_{1_2}, t_{1_2}), (t_{1_2}, p_{1_3})\}$
- $C_1 = \emptyset, \quad O_1 = \{o_1\}$
- $actassign_1 = \{(o_1, N_1)\}, \quad wpassign_1 = \{(w_1, N_1)\}$
- $tperformer_1 = \{(t_{1_1}, o_1), (t_{1_2}, o_1)\}, \quad constr_1 = \emptyset$
- $P_2 = \{p_{2_1}, p_{2_2}, p_{2_3}\}, \quad T_2 = \{t_{2_1}, t_{2_2}\}$
- $F_2 = \{(p_{2_1}, t_{2_1}), (t_{2_1}, p_{2_2}), (p_{2_2}, t_{2_2}), (t_{2_2}, p_{2_3})\}$
- $C_2 = \{c\}, \quad O_2 = \emptyset$
- $actassign_2 = \{(c, N_2)\}, \quad wpassign_2 = \{(w_2, N_2)\}$
- $tperformer_2 = \{(t_{2_1}, c), (t_{2_2}, c)\}, \quad constr_2 = \emptyset$
- $P_3 = \{p_{3_1}, p_{3_2}, p_{3_3}\}, \quad T_3 = \{t_{3_1}, t_{3_2}\}$
- $F_3 = \{(p_{3_1}, t_{3_1}), (t_{3_1}, p_{3_2}), (p_{3_2}, t_{3_2}), (t_{3_2}, p_{3_3})\}$
- $C_3 = \emptyset, \quad O_3 = \{o_2\}$
- $actassign_3 = \{(o_2, N_3)\}, \quad wpassign_3 = \{(w_3, N_3)\}$
- $tperformer_3 = \{(t_{3_1}, o_2), (t_{3_2}, o_2)\}, \quad constr_3 = \emptyset$



(a) Petri net N_1 depicts the process performed by the operator o_1.



(b) Petri net N_2 depicts the process performed by the cobot c.



(c) Petri net N_3 depicts the process performed by the operator o_2.

Figure 4.8.: Petri nets N_1, N_2, N_3 of the Packaging & Palletizing use case.

5. Technical analysis

In order to implement processes that utilize collaboration patterns, a test system is needed. This system needs the ability to combine a simulated cobot with a Workflow Management System. Before starting the actual implementation of such a test system, a few parts must be evaluated. Three main topics are looked into in detail in this chapter:

- Architectures for communication between robot simulation and Workflow Management System
- Offline robot simulation
- Workflow Management System (WfMS)

The goal is to provide a reasonable decision on which products and approaches to use for each of the above-mentioned parts of the setup.

5.1. Architectures

In order to set up the communication between robots and a Workflow Management System, different architectural approaches are possible. In this section, four architectures are discussed. Finally, a decision for one of the approaches is made, which will then be used in the implementation. The discussed approaches are, of course, not a complete list of possible ways to solve this problem. Very likely, there are additional variations of the discussed approaches possible, as well as completely different ones.

The first approach, as seen in Figure 5.1, is commonly seen in industrial automation. The right part of the picture shows a simplified representation of a production line. In this case, the production line consists of a central Programmable Logic Controller (PLC) and three separate robots. The connection between PLC and the robots is established via Industrial Ethernet.

Remark: Industrial Ethernet (IE) is the application of standard ethernet in the production industry. There are many additional requirements that the network must meet. The harsh environment demands for robust hardware, cabling and connectors, that can withstand increased amounts of electrical noise, vibration, temperature extremes, moisture and dust [6]. The demands on the network protocol are also different, as the communication is usually time-critical. Specialized ethernet protocols are used to deal with additional demands like real-time capability [6]. The most common ones are Profinet, Powerlink, EtherNet/IP and EtherCAT [6].

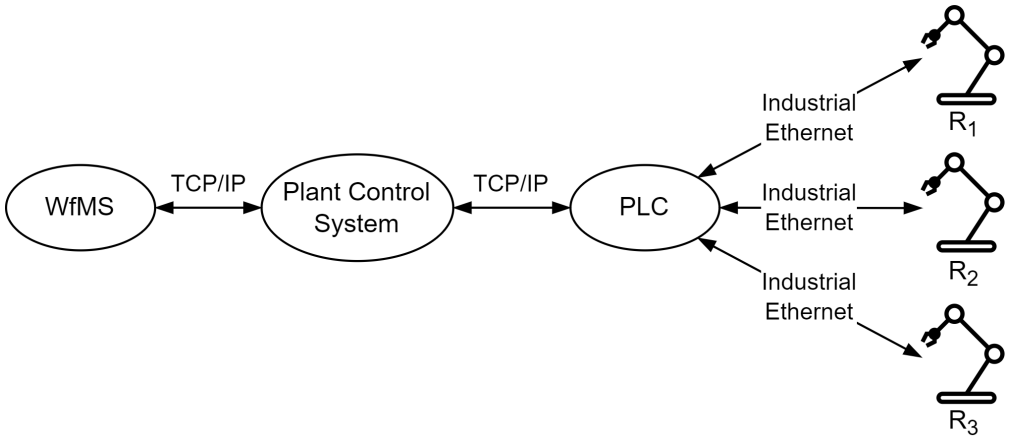


Figure 5.1.: Architecture with Plant Control System and PLC between WfMS and robots.

The PLC itself is connected to a higher-level control system (plant control system). Such systems are, for example, used to track performance measurements of the machines, and quality aspects of individual workpieces in certain production stages. The data received by the plant control system can be used by the WfMS to derive further actions. Subsequently, these decisions are sent back to the PLC via the plant control system. The production line then may handle each workpiece according to the decision from the WfMS.
Advantages of this approach:

- a standardized communication between PLC and robots
- connection from PLC to robots is widely used in automation industry
- good support on problems (PLC $\longleftrightarrow$ robot)

On the other side there are some drawbacks, especially when focusing directly on the communication between WfMS and robots:

- two superfluous instances (plant control system and PLC) need to be considered
- in-house development likely needed for the connection between plant control system and PLC
- data from and to the robots has to be dragged through the PLC, which also needs to be implemented separately

Another approach (cf. Fig. 5.2) is to establish a direct connection between the Workflow Management System and the robots. Robot manufacturers usually provide some data exchange interfaces. Different programming languages may be used to send data to the robot or retrieve information from it. This architecture would effectively get rid of any

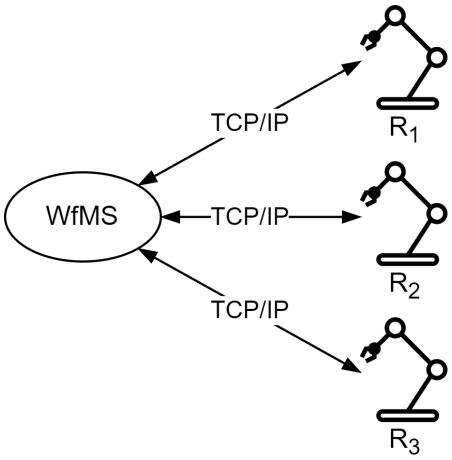


Figure 5.2.: Direct communication between WfMS and robots.

intermediaries. However, this method is highly specific and cannot be adapted to different situations without major rework. First of all, as no standard communication is used, each robot manufacturer needs to be handled differently. This may vary less with some manufacturers, but can make big differences for others or perhaps not be supported at all. On the other side of the communication channel, the WfMS also has to provide the functionality to implement the tools for the communication. This could result in compatibility problems between certain Workflow Management Systems and robots, which may lead to situations where this approach is basically unusable, e.g., use cases with different robots where only some of them are compatible with the WfMS.

The remaining two architectures (cf. Fig. 5.3 and 5.4) are quite similar, since they build upon the same technology choice. The communication between Workflow Management System and robots is achieved via the use of Open Platform Communications Unified Architecture (OPC UA) (cf. Sec. 5.2). The OPC server handles the communication on the robot side, whereas the OPC client is utilized by the WfMS to send and receive data. The main difference between these two approaches is the number of OPC servers used. In Figure 5.3, a single server is responsible for exchanging data with all robots, in contrast to Figure 5.4, where a separate server instance is deployed for each robot.

Both of them share the advantage of using a standardized communication (OPC UA), hence they are suited for a variety of different setups without the need for major rework. In both cases, however, OPC server and client have to be developed, unless the robot manufacturer already offers an implementation. But even if no implementation by the manufacturer is available, there is no need to start from scratch. Since OPC UA is an open standard, there are a variety of implementations in different programming languages to choose from. These can be used as a baseline for developing a server that supports individual requirements.

The implementation of the OPC client in Figure 5.3 is more lightweight, since it only

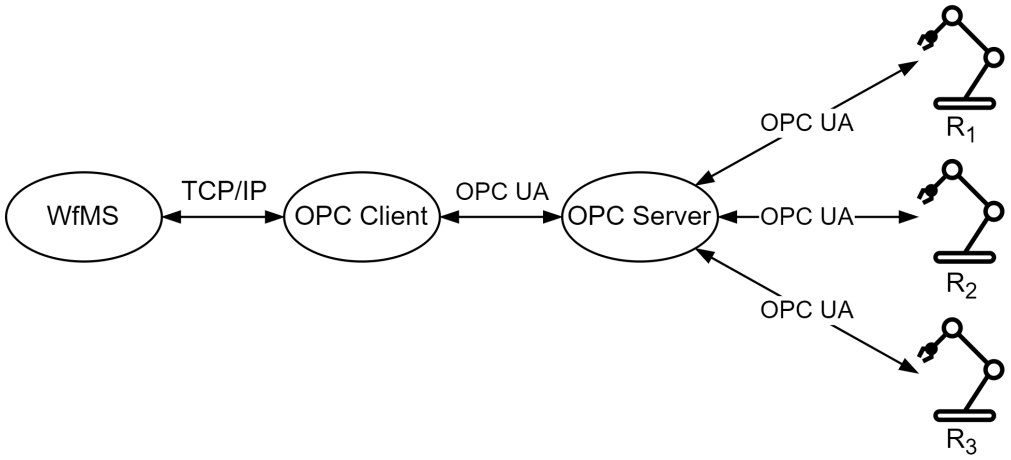


Figure 5.3.: Communication via single OPC Client and Server.

needs to connect to a single server instance. This, on the other hand, results in a potential bottleneck. Namely, the single OPC server instance, as it exchanges data with each robot in specific time intervals, leading to a loss of scalability. Also, the implementation of the server is much more complex in this case.

In Figure 5.4 the scalability problem is circumvented by the use of multiple server instances as mentioned above. This also simplifies the implementation of the server. The client needs to offer the functionality to connect to multiple servers when requested by the WfMS. This makes its implementation a bit more complex, in contrast to the client in Fig. 5.3. Unlike the single OPC server, the single OPC client does not need to constantly exchange data with each robot. Also, the amount of data is less. This prevents the client from becoming a bottleneck. For a simulated setup, Figure 5.4 suggests putting the robot simulation together with the corresponding OPC server inside a Virtual Machine (VM). This makes it particularly easy to add additional robots to the setup, as the VM can be easily duplicated.

As for deciding on an architectural approach, the architecture proposed in Figure 5.1 is not really suitable in this thesis. Although it is a common approach in industry, as larger production machines are usually PLC controlled, both the PLC and plant control system are an unnecessary intermediary in the context of this work. The second concept (Fig. 5.2) is not chosen due to limited compatibility and comparatively high effort in changing environments (e.g., different robot manufacturers). This leaves the decision between the two architectures using the open standard OPC UA as middleware (cf. Figures 5.3 and 5.4). From them, the latter is preferred, as the scalability problem is far less prominent and the embedding of OPC server and robot simulation inside a Virtual Machine offers great versatility regarding multiple robot instances.

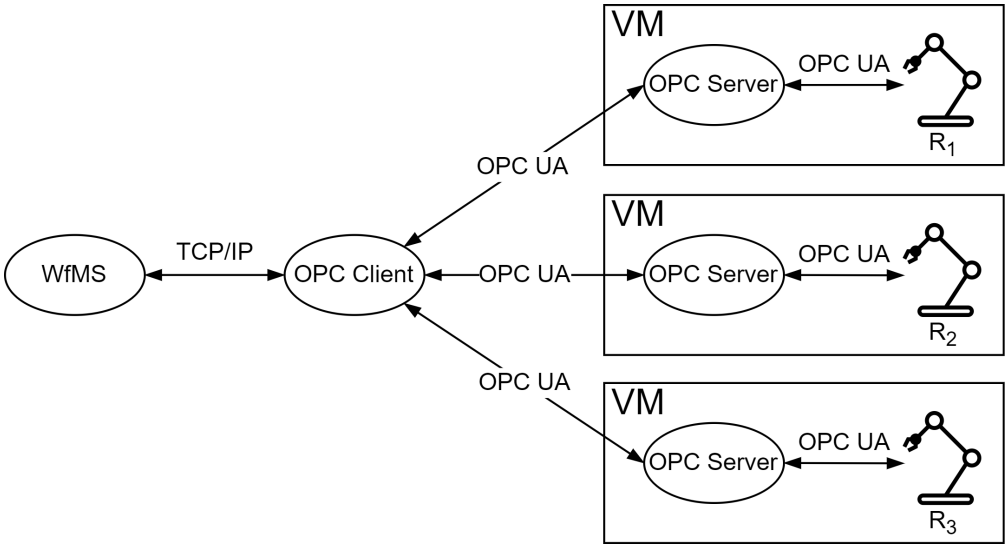


Figure 5.4.: Communication via a single OPC Client and one OPC Server for each robot. OPC Server and robot are set up inside a Virtual Machine.

5.2. OPC UA

Open Platform Communications Unified Architecture (OPC UA) is a platform independent interoperability standard for data exchange in industrial automation and other industries [72]. The initial standard was published in 1996 by a task force formed by several companies in 1995 [17], and is nowadays known as OPC Classic [69], [72]. It described a server-client software architecture to collect process data from devices to pass them to other systems [17]. Its purpose was to create a standardized interface for PLC specific protocols (e.g., Modbus, Profibus, etc.), to act as middleware for other systems (e.g., Human Machine Interface (HMI), Supervisory Control and Data Acquisition (SCADA)) [72]. In order to continue, maintain and market the standards, the OPC Foundation ([70]) was established [17]. OPC Classic was limited to Windows operating systems because it was based on Microsoft's OLE (Object Linking and Embedding), from which the original acronym (OLE for Process Control (OPC)) originated [72].

Even though the OPC specifications were widely used throughout the industry, there were some negative aspects [17] that led to a new major step with the release of OPC Unified Architecture (OPC UA) in 2008 [71]. It combines all of the individual OPC Classic specification's functionalities [69] (e.g., DA (Data Access), A&E (Alarms and Events), HDA (Historical Data Access), and Commands) in a single extensible framework [71]. Hence, it is functionally equivalent to OPC Classic [71]. OPC UA is platform independent. It supports different hardware platforms (e.g., traditional PC hardware, cloud-based servers, PLCs, microcontrollers, etc.) as well as various operating systems (e.g., Microsoft Windows, Apple OSX, Android, any Linux distribution, etc.). Hence, it provides the necessary infrastructure for interoperability across the enterprise [71]. A

major addition that came with OPC UA is security. In OPC UA, security aspects can be found at two different stages, which is the application layer and the communication layer [17]. As stated in [71], some of the features are:

- Session Encryption — secure message transmission
- Message Signing — origin and integrity of messages can be verified by the recipient
- Authentication — UA clients and servers are identified through certificates (X.509)
- Authorization — users can be granted access to or restricted from certain services or address spaces
- Auditing — user and system activities can be logged

OPC UA offers different types of data transmission (e.g., native binary, XML web-services, SOAP/HTTP with UA binary), which ensures that there is no reliance on specific operating systems or protocols [17].

OPC UA, with its multi-layered architecture, provides an extensible framework, i.e., new technologies like transport protocols, security algorithms, encoding standards, and application-services may be added, while also providing backward compatibility for existing products [71].

A fundamental element of OPC UA is its information modeling framework, which defines rules and base building blocks that are necessary to represent an information model [71]. The already defined core models may also be extended to describe more specific information in other domains [71]. The information model provides complete object-oriented capabilities so that even complex multi-level structures can be modeled appropriately [71]. The required access mechanisms are also defined [71]:

- browsing mechanism for instances and their semantics
- operations to read and write data
- method execution
- data and event notification

Alongside all these changes, the meaning of the acronym has also changed, and now stands for Open Platform Communications [72].

OPC UA offers a large amount of possibilities, and a lot of implementations in different programming languages exist. Therefore, building OPC UA applications can take a lot of effort. To simplify this process, Mangler J. developed the *opcua-smart*[1] library, which relies on the *open62541*[2] implementation of OPC UA. To achieve the goal of simplification, some constraints are put on the modeling functionality, and not all functions of OPC UA

[1] `https://github.com/etm/opcua-smart` (visited on 24th Nov. 2021)
[2] `https://open62541.org/` (visited on 24th Nov. 2021)

are offered. Together with Pauker F. an OPC UA server for Universal Robots e-series robots was developed (*URUA*[3]). It is written in the programming language Ruby and uses the *opcua-smart* and *ur-sock*[4] libraries. The implementation part of this work will utilize these tools to create a communication channel between a WfMS and the robot simulations.

5.3. Offline robot simulation

Most, if not all, major robot manufacturers provide some sort of offline programming and simulation software for their robot products. The terms Robotic Offline Programming (OLP) and offline robot simulation are often used interchangeably, although they don't exactly describe the same thing [44]. Technically, OLP can be used without simulation. Simply writing a program for a machine or robot in a text editor on a computer, and later uploading that program to the hardware, would qualify for the term offline programming [44]. Usually, though, OLP refers to programming a simulated robot [44]. Such simulation tools often offer the ability to upload CAD models of the environment of the robot, e.g., the workcell and workpieces [15]. The robot can then be programmed to fulfill its task in the simulated environment.

This approach has various advantages compared to online programming. First of all, safety. Since the program is developed and tested using a simulation, neither hardware nor humans are at risk when errors occur [15]. OLP can help to reduce downtimes when changes to existing production lines have to be made [15]. Furthermore, it ensures a quicker integration of new robots, since the program was already tested in a virtual environment. OLP can also be used to make proof of concepts [44] prior to buying the actual robot or building the workcell. Design flaws may get detected and fixed early on, resulting in cost savings in production [15], [44].

This section gives an overview on simulation tools of some of the well-known brands, namely ABB (RobotStudio [60]), KUKA (KUKA.Sim [66]), Universal Robots (URSim [74]), FANUC (Roboguide [65]), and YASKAWA (MotoSim EG-VRC [77]). As the purpose is to determine the applicability of the tool in collaborative use cases in combination with a Workflow Management System, there are some key aspects that have to be fulfilled, e.g., the reachability of the simulation from outside. According to Sections 5.1 and 5.2, this would, for example, be the support of OPC UA to enable communication.

5.3.1. Cobot support

Table 5.1 shows the rising popularity of cobots, as each of the listed manufacturers already has cobots in their product assortment. Although the cobots are available in the simulation tools, their features might be limited compared to their real counterparts. URSim states that, due to the fact that no real robot arm is connected, force control

[3] `https://github.com/fpauker/urua` (visited on 24th Nov. 2021)
[4] `https://github.com/fpauker/ur-sock` (visited on 24th Nov. 2021)

Manufacturer	Name	Cobot support	OPC UA	Offline Progr.	Simulated Environment	Availability
ABB	RobotStudio	+	+	+	+	30 day trial
KUKA	KUKA.Sim	+	+	+	+	4 week trial
Universal Robots	URSim	+	+	+	–	free
FANUC	Roboguide	+	+	+	+	30 day trial
YASKAWA	MotoSim EG-VRC	+	–	+	+	90 day demo

Table 5.1.: Comparison of robot offline simulation software tools.

is limited. Also, a specific feature of cobots, namely hand guiding (cf. Sec 2.2.7) is not possible in URSim. However, other simulators like RobotStudio offer something similar by dragging the robot with the mouse along specific joints or coordinate systems. It should be noted, though, that this functionality is not the same as real hand guiding, as it is also offered for robots that do not have that ability in reality.

5.3.2. OPC UA support

The robot manufacturers are also keen on supporting open standards like OPC UA (cf. Sec 5.2). Each of the listed simulation tools supports OPC UA in some way. The only exception here is MotoSim EG-VRC from YASKAWA. However, YASKAWA also offers an OPC server implementation for their actual products. There just doesn't seem to be any direct information on whether their simulation software also supports OPC UA. As already mentioned, the support for external communication is a vital part. The use of an open standard such as OPC UA is preferable, as it ensures that this approach is suitable for a broader range of applications and not only for a very specific scenario.

5.3.3. Simulated Environment

All of the tools offer functionality to create and execute robot programs, and while most of them also provide a simulation of the environment with uploadable 3D models, including collision detection between them and the simulated robot, URSim only provides a 3D simulation of the robot itself.

A simulated environment enables different kinds of analyses to be made.

Reachability analysis detects misplacements in the workcell layout (e.g., unreachable workpiece placements, interfering workcell parts, etc.) that impede proper task execution. It enables corrections to be made during the computer-aided design phase, rather than changing the mechanical construction once it is built, which can drastically reduce costs.

Cycle time predictions do not have to be estimated, but can be made with fairly high accuracy by simulating the actual task. This allows extensive changes to be implemented in advance in order to achieve specified cycle times.

Bottleneck identification helps to find and eliminate critical parts that slow down the process.

Digital Twin is the most extensive form of simulating a process. It is an exact copy of a production process, as a digital representation, which includes the components and also the dynamics of the physical system [57]. A digital twin of a system accompanies it along its lifecycle, from planning and design to production. It operates on the same data as the physical system, and is therefore a great tool to simulate changes of the process without affecting the actual production. ABB with RobotStudio and KUKA with KUKA.Sim explicitly state that their simulation tools provide the ability to implement digital twins.

5.3.4. Availability

Another important aspect for people interested in trying out robot simulations is the availability of said tools. The easiest one to obtain is URSim by Universal Robots, as it is basically free. The only prerequisite for the download of the software is an account on their website. ABB and FANUC offer 30 days of trial for their implementations, whereas KUKA provides a 4-week trial license. MotoSim EG-VRC by YASKAWA is available as a 90-day demo version. Easily accessible versions for testing purposes, whether they are free or timely limited, are especially important because the prices of the full versions are usually quite high.

5.3.5. Decision

The decision for the simulation tool was made for URSim from Universal Robots. The decision is based on a few different aspects. As discussed in Section 5.1 the desired setup should comprise separate robot instances, just like in a non-simulated setup. This is achieved by deploying multiple instances of URSim. To be more specific, multiple instances of a Virtual Machine, each standalone and with different network addresses. This already is a further advantage. It is easily possible to build setups with multiple cobots, which is one of the investigation points of this work.

Although the unavailability of a simulated environment is a downside of URSim, the focus is set on the collaboration between human and cobot and the combination with a WfMS. The environment simulations are better suited for exactly specified processes without the direct influence of humans. A simulated environment would also introduce a major workload in 3D modeling, which is further out of scope.

Another factor is the possibility to test some of the implementations on a real setup, as two UR cobots might be available at the Faculty of Computer Science at the University of Vienna. In this context, previous work has also been done regarding an OPC UA server

implementation for UR cobots (*URUA* and *opcua-smart*), which acts as a baseline for the implementation in this work, as already mentioned in Section 5.2.

5.4. Workflow Management System

Workflow Management Systems (WfMS) play an important role in workflow management. They aim to achieve business goals by sequencing work activities and then invoking the appropriate resources (e.g., human or information) for the corresponding activities [2]. A WfMS is a computer-aided system that supports the execution of business processes based on predefined models [3], [18]. The goals of Workflow Management Systems include consistent processing of processes, transparency of processes, and cycle time reductions [38]. A Workflow Management System consists of multiple components [38]:

- Modeling component — provides users a tool to design processes; multiple process execution languages are possible (e.g., Business Process Model and Notation (BPMN)[18], Petri net [4])
- Workflow engine — responsible for the process activation and execution
- Application component — triggers applications which are used to process activities (e.g., applications for human interaction, or automated services)
- Monitoring component — enables users to analyze the business processes and thus helps to improve them

The functional scope of Workflow Management Systems is usually very large. Therefore, only a few key aspects that are needed in the current context are of interest here. These include:

- the use of BPMN for modeling the processes
- the ability to integrate external services, preferably via Representational State Transfer (REST)
- an Application Programming Interface (API) to gain access to the processes (e.g., to send messages to a running process instance ("message correlation"))
- the availability of the tool, preferably freeware, as there is no need for a full-fledged enterprise product variant

The following is a short description of six Workflow Management Systems. Furthermore, Table 5.2 provides a brief overview regarding the aspects mentioned above.

Tool	BPMN	Ext. service connectors	API for ext. access	Availability
AristaFlow	+	+	+	free research and education license
Bizagi	+	+	+	free modeler and 30 day trial for studio/engine
Bonita	+	+	+	free community edition (open source)
Camunda	+	+	+	free community edition (open source)
jBPM	+	+	+	free, open source
YAWL	–	+	+	free, open source

Table 5.2.: Comparison of Workflow Management Systems.

AristaFlow

AristaFlow originally emerged from a project in 2004–2007, with partners including the University of Ulm, the University of Mannheim, Daimler AG Research & Development, and others [76]. AristaFlow's models are based on BPMN, like many others. The difference of this modeler to the majority of other tools is, that it is driven by the principle of *correctness by construction* [61]. That is, the process designer can not simply create arbitrary models, that may include unreachable nodes, for example [61]. Instead, the modeler only offers correct modifications on each node, of which the designer can choose [61]. The layout is then created by the modeling tool [61]. AristaFlow provides APIs (e.g., Java, web service, REST) for integration of other software stacks [61]. AristaFlow offers a free research & education license for their software solution[5].

Bizagi

The Bizagi software platform consists of three major components, the *Bizagi Modeler*, *Bizagi Studio*, and *Bizagi Automation* [62]. The process models are based on BPMN, and the modeling tool also supports the import of BPMN files. The platform offers connectors for many external services, as well as simple REST and Simple Object Access Protocol (SOAP) connectors [62]. For external access to the Bizagi system, a REST API, and a SOAP service is offered [62]. Although the modeling tool is free to use, in order to build and run processes, the other two tools are required [62]. The *Studio* and *Automation* tools can be used for 30 days as a trial version [62].

[5] `https://www.aristaflow.com/fuer-forschung-lehre.html` (visited on 10th Feb. 2022)

Bonita

Bonita is an open source platform based on Java and is used for business process automation and optimization [63]. It consists of three major components, *Bonita Studio*, *Bonita Runtime*, and *Bonita Continuous Delivery* [63]. *Studio* is the development environment, *Runtime* runs the engine and deployed applications, and *Continuous Delivery* supports the iterative deployment of projects [63]. Bonitasoft provides a free community edition of Bonita, which includes *Studio* and *Runtime* [63]. BPMN is used for the modeling of processes [63], and importing models of this format is also supported. For the integration of external services Bonita offers various connectors, also including REST and SOAP [63]. To gain access to processes a REST and Java API can be utilized [63].

Camunda

Camunda is a framework based on Java that supports BPMN among other standards [64]. Camunda is available for free as a community edition, which is provided under open source licenses [64]. It offers a REST API to provide access to the relevant interfaces of the engine, which is described in the documentation [64]. Camunda offers REST and SOAP connectors through third-party libraries (Apache Http Components), that enable processes to call external services [64].

jBPM

As the name indicates, jBPM is developed in Java. It is a light-weight, fully open source business process management suite [59]. The business processes are designed using the latest BPMN 2.0 specification [59]. So-called *Work Item Handlers* offer the ability to call external services, for example, using REST or SOAP [59]. jBPM also offers a REST API to gain external access to the workflow engine [59].

YAWL

The following information is from the official documentation of the tool [48]. YAWL stands for Yet Another Workflow Language and was developed by Wil van der Aalst and Arthur ter Hofstede in 2002. The language is an extension of Petri nets to fully support the Workflow Patterns[6]. The YAWL system is based on Java and comprises several web servlets and a Java-based desktop application. The tool works on Windows, Linux and Mac OSX platforms. YAWL is an open source workflow system, and as such available for free. As the name suggests, the processes build upon the language Yet Another Workflow Language (YAWL), and not on BPMN like many other tools. The integrated *Web Service Invoker Service (WSInvoker)* can be used to integrate external SOAP services into the designed processes. To access the system, custom external YAWL services are utilized that interact with the engine via XML/HTTP messages.

[6] `http://www.workflowpatterns.com/` (visited on 6th Feb. 2022)

The tool of choice in this work is Bonita. It offers good usability and contains all functions desired in the context of this work. Furthermore, Bonitasoft, the developer of Bonita, provides an excellent documentation on how to use each aspect of their system, with working examples as help points, e.g., a step-by-step guide on how to use the external API.

5.5. Modeling language choice

Historically, there is a big gap between business information systems, and the systems running on the shop floor. Systems on the shop floor are highly specialized for the purposes they have to fulfill. Among them are production lines and machines with special hardware controlling their actions (e.g., Programmable Logic Controller (PLC), CNC machines), robots and other systems. They are usually designed to only interact with selected other shop floor systems. For example, robots are expected to work in production lines in coordination with a PLC that orchestrates the whole production line. Therefore, manufacturers provide modules to integrate the robot into the PLC. This way the PLC is able to send data to the robot and also the other way around. However, they are not designed to interact with arbitrary systems. This means that communication between information systems of these two areas is not a trivial task, as it requires programmers with domain-knowledge on the highly specialized parts [42]. Creating solutions to this problem is an important task in research and also in industry itself. A term regarding this research area is Cyber-Physical System (CPS) and Cyber-Physical Production System (CPPS) [19], [39], [42]. This approach is expected to break with the typical automation pyramid, where communication is only available between modules on the same level of the pyramid [19], [42]. Instead, a sort of network establishes where functions of different layers communicate with each other [19], [42]. [39] and [42] use a BPMN-based approach for the automation with an orchestration framework that is based on BPMN processes. Since one of the goals in this work is to enable flexibility in the production use cases by the combination of cobots and Workflow Management Systems, the choice for the modeling language to be used is also made for BPMN.

BPMN is one of the most used and well-known modeling language standards for business processes. It was developed and released by IBM employee Stephen A. White and the Business Process Management Initiative (BPMI) [5]. The first specification, BPMN 1.0 was released in May 2004 [5]. In 2005 BPMI merged with the Object Management Group (OMG). Since then, BPMN is maintained by the OMG. The latest major version is BPMN 2.0, which was released in 2011. BPMN aims to provide a notation which is understandable by all business users [5]. It builds a common ground for business analysts who create process drafts, technical developers who implement the technology to run these processes, and business people who manage and monitor them [5]. Over time, and especially with the introduction of BPMN 2.0, the standard added new things and refined existing functionalities [14]. With version 2.0, execution semantics for all BPMN elements were added [14]. Furthermore, an extensibility mechanism for process model extensions

and graphical extensions was defined [14]. The definition of human interactions was extended, and known inconsistencies and ambiguities of previous versions were resolved [14]. Furthermore, a metamodel alongside XML based interchange formats has been defined [14]. BPMN models are composed of elements from four basic categories [5]:

- Flow Objects
- Connecting Objects
- Swimlanes
- Artifacts

Flow Objects include, for example, Events, Activities, and Gateways [5]. Events are represented as circles and denote something that happens during the execution of the process [5]. Events affect the flow of the process (e.g., error events) [5]. There are basically three different types of events, Start, Intermediate, and End [5], with additional subtypes as of BPMN 2.0. Processes modeled in BPMN always start with an event, and also end with an event. In between, the process is built mainly by series of activities. An activity is a generic term for work that is performed in the process, and can either be an atomic task or a compound of tasks, a so-called Subprocess [5]. Activities are displayed as rectangles with rounded corners [5]. Subprocess activities are distinguished by a small plus sign at the bottom center of the rectangle [5]. In current versions of BPMN, activities can be of different task types, e.g., user tasks, service tasks, script tasks, etc. These types specify the nature of the performed task. The divergence and convergence of the sequence flow is controlled by gateways [5]. Hence, they determine traditional decisions, and the forking and merging of paths [5]. Gateways are represented as diamond shapes [5]. The two most common types of gateways are exclusive and parallel gateways. They are distinguished by internal markers in the diamond shape, usually a plus sign for parallel gateways and either nothing or an "X" for exclusive gateways. There are also other types of gateways, such as inclusive, event-based and complex gateways.

In order to connect the different Flow Objects in a process model, there are different types of Connecting Objects [5]. These connectors are represented as arrows of different types (e.g., solid line, dashed line, dotted line) [5]. The sequence flow is modeled by arrows with solid lines and arrowheads [5]. It is used to show the order in which the activities of a process are performed [5], and is therefore the main type of connector seen in most BPMN models. The next connector type is the message flow, and is displayed by a dashed line with an open arrowhead [5]. This connector is used to show the flow of messages between different process participants [5]. The third type is an association, and is denoted by dotted lines with a line arrowhead [5]. This connector is, for example, used to associate data and text with flow objects, and shows inputs and outputs of activities [5].

The third category of elements in BPMN are Swimlanes. They are a mechanism to group the activities and to show the responsibilities of performers [5]. In BPMN, there are two types of swimlanes, namely Pools and Lanes [5]. Pools represent participants of

a process, whereas Lanes are sub-partitions within Pools, and are used to organize and categorize activities within a Pool [5]. Pools are used when there are multiple business entities or participants involved in a process. They are physically separated in the diagram, and the activities within each Pool are distinct processes [5]. The sequence flows must not cross the boundaries of a Pool. To show messages between participants or Pools, the aforementioned message flow connector is used [5].

The final category are Artifacts. These elements are used to provide additional process information that does not affect the flow of the process [14]. Artifact types are Data Objects, Groups, and Annotations [5]. A Data Object denotes required or produced data by an activity and is represented as a paper sheet [5]. Groups are depicted by dashed-dotted rectangles with rounded corners and are used to group elements for documentation or analysis purposes [5]. Annotations are purely used to provide the modeler a mechanism to add additional textual information for the reader of the diagram [5].

A clear summary of all elements of the BPMN 2.0 standard is provided by this poster[7].

[7] `http://www.bpmb.de/images/BPMN2_0_Poster_EN.pdf` (visited on 17th Mar. 2022)

6. Technical implementation

This chapter provides details on the technical implementation of the test setup, which is used to realize several use cases of different collaboration patterns. It comprises multiple distinct parts:

- Cobot simulation
- OPC UA server
- OPC UA client
- Workflow Management System

For the overall architectural design choices and software solutions that are used here, please refer to Chapter 5. The goal of this implementation is to demonstrate the usability of the patterns with the chosen approach of combining WfMS, cobots, and OPC UA.

This chapter starts by introducing the general architectural approach the system follows. Afterwards, a guide follows on how the different parts are set up and which potential problems may occur. Subsequently, some important aspects of the OPC UA client service and the Workflow Management System are discussed. Finally, the basic structure of an exemplary cobot program is shown.

6.1. General architecture

As proposed in Section 5.1, the general architecture focuses on the use of Virtual Machines. This approach has some important advantages. First, it ensures that the different components are actually independent of each other, just as in real world applications. This makes it easier to compare it to such cases, and also to deploy this implementation into a real world application with distributed services. In addition, problems with system dependencies, such as different versions of libraries interfering with each other, can be easily avoided. Another advantage in the simulated environment is that within a very short amount of time, multiple cobot simulations with already configured setups and OPC UA server implementations can be deployed.

As depicted in Fig. 6.1, at least two instances of VMs must be deployed. One of them contains the Workflow Management System and the OPC UA client implementation. The other one is responsible for the simulation of the cobot and also the OPC UA server implementation. In case more cobots are needed for a use case, the second VM can be cloned. With just minor adjustments, the second cobot simulation is ready to be used.

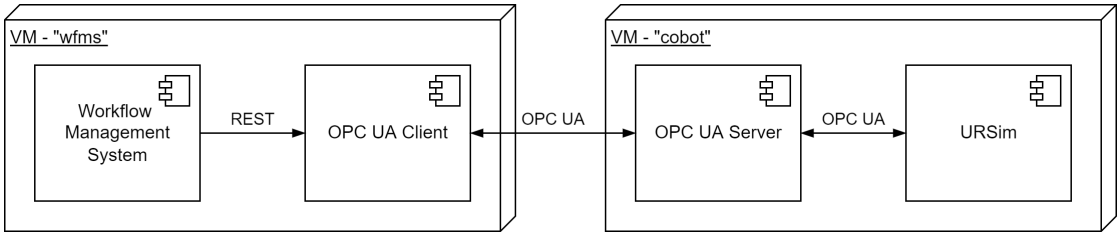


Figure 6.1.: General architecture of the implementation setup. Virtual Machine *wfms* contains the Workflow Management System and the OPC UA client, and Virtual Machine *cobot* the OPC UA server and the cobot simulation.

6.2. Setup guide

This section guides through the process of setting up the various parts of the deployment structure. Software version numbers are mentioned in some places in the guide. These should be kept in mind during the installation process, otherwise incompatibilities could occur. Additionally, encountered problems and potential solutions and workarounds are discussed. As previously stated, the individual services are running inside Virtual Machines. Hence, they are the starting point of the setup process.

6.2.1. Virtual Machine installation

There are various possibilities of setting up and using Virtual Machines. One of them is a software called *VirtualBox (VBox)*[1], which is free to use. The following steps guide through the setup process of Virtual Machines in VBox. This procedure is the same for both Virtual Machines.

1. Create a new Virtual Machine in VBox, by clicking on *Machine→New or (Ctrl+N).*
2. In the newly opened window, the name of the VM will be set (*wfms* and *cobot*).
3. Furthermore, the type will be set to *Linux* and version to *Ubuntu (64-bit).*
4. In the following window, *Memory size* depends on the VM. For the *wfms* set at least 4 GB, for the *cobot* 2 GB are sufficient. Note that RAM settings can be changed at any time if needed.
5. In the next window, keep the default settings of *Create a virtual hard disk now.*
6. The same applies to the *Hard disk file type* at *VDI (VirtualBox Disk Image).*
7. *Storage on physical hard disk* can also be left at *Dynamically allocated.*
8. Change *File size* to 15 GB.

[1] `https://www.virtualbox.org/` (visited on 7th Dec. 2021)

9. Finally, under *Settings* and *Network*, *Attached to* should be changed from *NAT* to *Bridged Adapter*.

10. Optionally, under *System→Processor*, the cores can be adjusted based on the hardware used.

11. If the Virtual Machine is running on a solid-state drive (SSD), the checkmark for it should be set at *Storage→Controller:SATA→FILENAME.vdi*.

After creating a new Virtual Machine in VBox, it is time to install an operating system (OS). In this guide, and also the whole implementation setup, *Ubuntu 18.04.6 LTS (Bionic Beaver)* will be used. The required ISO image for the installation can be downloaded here[2]. After starting the Virtual Machine inside VBox, a prompt to select an image to start from will appear. Here, the downloaded ISO image file must be selected. Consequently, the Ubuntu installation process will start.

1. Choose *English* and *Install Ubuntu*.

2. Select preferred keyboard layout.

3. For *Updates and other software* choose *Minimal installation*.

4. *Installation type* left at default of *Erase disk and install Ubuntu*.

5. Select timezone.

6. For computer's name enter *wfms-VM* and *cobot-VM*, respectively.

7. Username to *wfms* or *cobot*.

8. Set passwords (for simplicity, only for the test setup, *wfms* and *cobot* were used as passwords).

9. In addition, the *Log in automatically* option can be selected to log in automatically at startup.

After completing the installation of the OS as well as the updates that will pop up shortly after startup, the first thing to do is to install the *VirtualBox Guest Additions*. It consists of drivers and applications for optimization of performance and usability, and enables a few convenience features, such as shared folders, automated logins, shared clipboard, drag & drop between host and guest, mouse pointer integration, and better video support. Prior to installing them, a few commands have to be executed, in order to be able to successfully complete the installation process (cf. Listing 6.1). Otherwise, the installation will partly fail with the following error message:

- *This system is currently not set up to build kernel modules.*
  Please install the gcc make perl packages from your distribution.

[2] `https://releases.ubuntu.com/bionic/` (visited on 8th Dec. 2021)

To fix that error, the leftovers of the process need to be completely removed, and then started from the beginning.

```
sudo apt update
sudo apt upgrade
sudo apt install build-essential gcc make perl dkms
reboot
```

Listing 6.1: Commands to execute before installing *Guest Additions* (terminal).

After that, install the *Guest Additions* by clicking on *Devices→Insert Guest Additions CD Image* and click run in the appearing prompt window. Finally, reboot the VM once again to finish the installation process.

Earlier in the initialization of the Virtual Machine (VM), the network adapter was set to *Bridged Adapter*. That way, the VM will show up as an individual device in the network the host computer is connected to. The default settings use DHCP to obtain an IP-address automatically. This is usually fine, but it would pose a problem later on for the settings used in the OPC UA client. To circumvent this problem, a static address will be used. In the settings menu of Ubuntu under *Network*, change the active connection of IPv4 to the desired address. Depending on the network, it should be checked that these addresses are not already in use by other devices, and preferably outside of the DHCP range. In this test setup, the address allocation is as follows:

- *wfms*-VM – *192.168.0.200*
- *cobot*-VMs – from *192.168.0.210* onwards

6.2.2. URSim

Installing the robot simulation software URSim is only relevant to the *cobot*-VM. Two prerequisites have to be fulfilled. URSim needs *Python 2.6* (version 2.7 works fine as well), and *Java JDK 1.8*.

Python 2

In this version of Ubuntu, the installation process of *Python 2* is quite simple. Run the commands as seen in Listing 6.2.

```
sudo apt install python-minimal
python -V
# Output similar to:
# Python 2.7.17
```

Listing 6.2: Installing Python 2 on Ubuntu 18.04.6 LTS (terminal).

For newer versions of Ubuntu, the default interpreter for Python changed to *Python 3*. To install *Python 2* run the commands in Listing 6.3.

```
sudo apt install python2
python2 -V
```

Listing 6.3: Installing Python 2 on Ubuntu 20.04.3 LTS (terminal).

After successfully installing *Python 2* change the default version as depicted in Listing 6.4.

```
sudo update-alternatives --install /usr/bin/python python /usr/bin/
    python2 1
sudo update-alternatives --install /usr/bin/python python /usr/bin/
    python3 2
sudo update-alternatives --config python   # type selection number: 1
python -V
```

Listing 6.4: Set Python 2 as default version on Ubuntu 20.04.3 LTS (terminal).

Java 1.8 JDK

To install and check the Java version, run the commands in Listing 6.5.

```
sudo apt install openjdk-8-jdk
java -version
# Output similar to:
# openjdk version "1.8.0_292"
# OpenJDK Runtime Environment (build 1.8.0_292-8u292-b10-0ubuntu1~18.04-
    b10)
# OpenJDK 64-Bit Server VM (build 25.292-b10, mixed mode)
```

Listing 6.5: Install and check JDK (terminal).

URSim 5.11.5

First of all, it is preferred to install this software in a dedicated VM. Otherwise, the installation script provided by the manufacturer may overwrite and replace other dependencies[3].

Download the software from the UR website[4] (free account is required) and move the file to the *Home* directory. After that, follow the commands in Listing 6.6.

```
tar xvzf  URSim_Linux-5.11.5.1010327.tar.gz
cd ursim-5.11.5.1010327/
./install.sh # without sudo, otherwise it won't be able to create desktop
     shortcuts
./start-ursim.sh UR5 # start the simulator once in the terminal,
    otherwise the desktop shortcuts won't work
```

Listing 6.6: Extract and install URSim (terminal).

Now URSim will also successfully launch using the desktop shortcuts. Upon starting the simulation, these two errors will likely pop up:

- *Error reading state (Unable to read state)*
- *Error updating state (Unable to set Remote Control)*

[3] https://forum.universal-robots.com/t/offline-simulator-e-series-ur-sim-for-linux-5-11-1-removes-all-installed-files/15384 (visited on 9th Dec. 2021)

[4] https://www.universal-robots.com/download/software-e-series/simulator-linux/offline-simulator-e-series-ur-sim-for-linux-5115/ (visited on 9th Dec. 2021)

In order to resolve these errors, a few libraries of URSim need to be copied into */usr/bin* (source[5]). Open a new terminal and follow Listing 6.7.

```
cd ursim-5.11.5.1010327/
sudo cp -avr ./usr/bin /usr
```

Listing 6.7: Copy URSim libraries into correct folder (terminal).

There is also another approach to fix these errors, by adding the folder of the URSim libraries to the PATH variable (cf. Listing 6.8) (source[6]). In the setup used in this work, this fix only worked for the second error message, but not for the first one. Hence, the variant of copying the files was preferred.

```
echo -e '\nexport PATH="$PATH:$HOME/ursim-5.11.5.1010327/usr/bin"' >> ~/.
    profile
```

Listing 6.8: Adding URSim libraries folder to PATH (terminal).

For newer versions of Ubuntu, the following line in *install.sh* (cf. Listing 6.9) produces an error.

```
commonDependencies='libcurl3 libjava3d-* ttf-dejavu* fonts-ipafont fonts-
    baekmuk fonts-nanum fonts-arphic-uming fonts-arphic-ukai'
```

Listing 6.9: Dependency problem in URSim install script on newer Ubuntu version (e.g. 20.04 LTS).

To prevent the installation script from aborting, change *libcurl3* to *libcurl4*. However, no tests, on whether this change could create problems with URSim later on, were made.

6.2.3. OPC UA

For the communication between Workflow Management System and cobot simulation, OPC UA is used. This section is relevant to the *cobot*-VM, where the server will be running, and the first parts of it, up to and including *opcua-smart*, also apply to the *wfms*-VM, since the client will be running there.

To start out, Git and Ruby need to be installed. Afterwards, add the *gem user directory* to the PATH variable and reboot the VM. Otherwise, installing Ruby gems will throw a warning message similar to:

- *You don't have /home/cobot/.gem/ruby/2.5.0/bin in your PATH, gem executables will not run.*

```
sudo apt install git
sudo apt install ruby-dev

echo -e '\nexport GEM_HOME="$(ruby -e '\''puts Gem.user_dir'\'')"' >> ~/.
    profile
```

[5] https://github.com/githubuser0xFFFF/URSim_Install_Guides#copy-libraries-to-usrbin (visited on 9th Dec. 2021)

[6] https://github.com/ljden/URSim_Install_Guides#add-binaries-to-path (visited on 9th Dec. 2021)

```
echo 'export PATH="$PATH:$GEM_HOME/bin"' >> ~/.profile
reboot
```

Listing 6.10: Installing Git and Ruby (terminal).

opcua-smart

The *opcua-smart* library aims to simplify the development of OPC UA applications (server and client). It provides Ruby bindings of the *open62541* implementation[7] of OPC UA, which is written in C. Therefore, *opcua-smart* relies on *open62541*. Hence, *open62541* will be installed at first. The instructions in this section are adapted from the *opcua-smart* GitHub[8].

For the manual installation of *open62541*, *cmake* is needed. On Ubuntu 18.04.6 LTS, the command *apt install cmake* will install version *3.10.2*. However, the current version of *open62541* requires at least *cmake 3.12*, otherwise the installation will fail. A simple workaround to use a newer version of *cmake* on Ubuntu 18.04.6 LTS is shown in Listing 6.11. Also, a few other libraries are required for the installation. The command to install those is also seen in Listing 6.11.

```
sudo snap install cmake --classic
cmake --version # check version (should be >= 3.12)

sudo apt install build-essential libmbedtls-dev libxml2-dev libxslt-dev
    libz-dev libssl-dev libicu-dev
```

Listing 6.11: Installing Cmake (terminal).

Now, *open62541* and *opcua-smart* can be downloaded and installed with the commands in Listing 6.12.

```
git clone https://github.com/open62541/open62541.git
cd open62541
mkdir build && cd build
cmake -DBUILD_SHARED_LIBS=ON -DCMAKE_BUILD_TYPE=RelWithDebInfo -
    DUA_ENABLE_AMALGAMATION=ON -DUA_ENABLE_ENCRYPTION=ON -
    DUA_ENABLE_ENCRYPTION_MBEDTLS=ON ..
make
sudo make install
sudo ldconfig # update libs
sudo ldconfig -p | grep libopen62541 # check libs
# Output should look like this:
#   libopen62541.so.1 (libc6,x86-64) => /usr/local/lib/libopen62541.so.1
#   libopen62541.so (libc6,x86-64) => /usr/local/lib/libopen62541.so
cd
git clone https://github.com/etm/opcua-smart
gem install --user-install opcua
```

Listing 6.12: Installing open62541 and opcua-smart (terminal).

[7] https://open62541.org/ (visited on 24th Nov. 2021)
[8] https://github.com/etm/opcua-smart (visited on 24th Nov. 2021)

Up to this point, the instructions were relevant to both Virtual Machines. The following two parts, *ur-sock* and *URUA*, only apply to the *cobot*-VM.

ur-sock

UR robots provide different interfaces for data exchange. The *ur-sock* library provides functions to interact with them, especially with the UR e-series robots, which it was designed for (source [9]). The installation process is shown in Listing 6.13.

```
git clone https://github.com/fpauker/ur-sock
gem install --user-install xml-smart
gem install --user-install ur-sock
```

Listing 6.13: Installing ur-sock (terminal).

URUA

For the OPC UA server implementation, *URUA*[10] is used. *URUA* uses the libraries *ur-sock* and *opcua-smart* to implement its functionalities. The installation process is similar to *ur-sock* (cf. Listing 6.14).

```
git clone https://github.com/fpauker/urua
gem install --user-install urua
sudo apt install openssh-server

# Go into URUA folder and switch to branch "registertest" for the newest
    version, as of writing
cd urua/
git checkout registertest
```

Listing 6.14: Installing URUA (terminal).

Although *URUA* is now installed and practically ready to be used, there are a few additional setup steps needed. First, a small fix needs to be applied. Otherwise, the server won't update the values for some output registers, which are needed in the implementation. Listing 6.15 shows the added lines of code, and also the line numbers where they are placed inside the file *urua/lib/urua.rb*, which range from 553 to 566.

```
553 #register
554 # Output Register
555 # Bit
556 64.upto(127) do |i|
557   opts['o_bit'][i-64].value = data["output_bit_register_#{i}"]
558 end
559 # Int
560 0.upto(47) do |i|
561   opts['o_int'][i].value = data["output_int_register_#{i}"]
562 end
563 # Double
```

[9] https://github.com/fpauker/ur-sock (visited on 24th Nov. 2021)
[10] https://github.com/fpauker/urua (visited on 24th Nov. 2021)

```
564 0.upto(47) do |i|
565   opts['o_doub'][i].value = data["output_double_register_#{i}"]
566 end
```

Listing 6.15: Fix for updating output registers (in file: urua/lib/urua.rb).

The next step for the input and output registers to work is a simple configuration task. It has to be applied in the file *urua/lib/rtde.conf.xml*. Depending on the needed input and output bit, int and double registers, the corresponding entries need to be added (cf. Listing 6.16). For the implemented use cases in the context of this work, the required entries are:

- Input and output bit 64, which are used for a handshake
- Input integers 24 and 25, which are used to control the program flow of the cobot
- Input doubles 24 to 35, which are used to write position coordinates to the cobot

```
...
<recipe key="inbit">
  <!--Only put Input Bit Registers here which are used (64-127 free)-->
  <field name="input_bit_register_64" type="BOOL"/>
</recipe>
<recipe key="inint">
  <!--Only put Input Int Registers here which are used (0-23 reserverd
    for Fieldbus/PLC | 24-47 free)-->
  <field name="input_int_register_24" type="INT32"/>
  <field name="input_int_register_25" type="INT32"/>
</recipe>
<recipe key="indoub">
  <!--Only put Input Double Registers here which are used (0-23 reserverd
     for Fieldbus/PLC | 24-47 free)-->
  <field name="input_double_register_24" type="DOUBLE"/>
  ...
  <field name="input_double_register_35" type="DOUBLE"/>
</recipe>
<recipe key="out">
  ...
  <!--Only put Output Bit Registers here which are used (64-127 free)-->
  <field name="output_bit_register_64" type="BOOL"/>
  ...
  <!--Only put Output Int Registers here which are used (0-23 reserverd
    for Fieldbus/PLC | 24-47 free)-->
  <field name="output_int_register_24" type="INT32"/>
  ...
  <!--Only put Output Double Registers here which are used (0-23
    reserverd for Fieldbus/PLC | 24-47 free)-->
  <field name="output_double_register_24" type="DOUBLE"/>
  ...
</recipe>
```

Listing 6.16: Configuration of required registers (in file: urua/lib/rtde.conf.xml).

Finally, the configuration of the server has to be updated. These settings are inside the files *urua/server/devserver.conf* and *urua/server/uruaserver.conf*. Due to the changes made in Listing 6.15, the *devserver* entrypoint has to be used, as this version uses the changed *urua.rb* file. Hence, the following changes in Listing 6.17 are only needed in the *devserver.conf* file. Depending on the current *cobot-VM* the cobot number in line 3 needs to be adjusted. The rest of the file is identical for all cobots.

```
1  ### OPC UA specific config
2  ### ----------------------------------------------------------------
3  namespace: https://centurio.work/cobot1
4
5  ### UR simulation specific config
6  ### ----------------------------------------------------------------
7  ### if no password, password has to entered when server is started
8  ipadress: localhost
9  sshport: 22
10 username: cobot
11 password: cobot
12 url: /home/cobot/ursim-5.11.5.1010327/programs
```

Listing 6.17: URUA server config (in file: urua/server/devserver.conf).

URUA can be started directly from the terminal (cf. Listing 6.18) or via a desktop shortcut provided here[11]. Note that URSim needs to be running before starting URUA.

```
cd urua/server
./devserver.rb start -v
```

Listing 6.18: Starting URUA from terminal.

The desktop shortcut must first be made executable, in order to use it (cf. Listing 6.19).

```
cd Desktop/
chmod +x URUA.desktop
```

Listing 6.19: Making the desktop shortcut for URUA executable (terminal).

Client Service

On the opposing side of the OPC UA server, an OPC UA client is needed. This client is running in the *wfms-VM*, so this part is only relevant to this VM. The client establishes a connection to the server in order to read and write values from and to the cobot. For this purpose, it uses the functionality of the *opcua-smart* library. Furthermore, this service must offer access points for the Workflow Management System, where the processes that need to communicate with the cobots are running. Back in Section 5.4, REST was mentioned as a preferred method to integrate an external service into the WfMS. Hence, the client service is served as a web API offering REST endpoints for various purposes. There are many existing frameworks in different programming languages, that offer relatively simple development of a web API, e.g., *Spring Boot*[12] (Java), *Django REST*[13]

[11] `https://github.com/samhabers92/CollabPatternsCobots` (visited 29th Apr. 2022)
[12] `https://spring.io/projects/spring-boot` (visited on 21st Feb. 2022)
[13] `https://www.django-rest-framework.org/` (visited on 21st Feb. 2022)

and *Flask*[14] (Python), *Express Js*[15] and *Fastify*[16] (JavaScript), *Ruby on Rails*[17] and *Sinatra*[18] (Ruby), and many more. Since the OPC UA server and client implementation are written in Ruby, the decision was made for Sinatra, as it is a much lighter framework compared to Ruby on Rails, and perfectly sufficient for this case.

Another function this service must offer, is to make callbacks to running process instances of the WfMS. For this purpose, Bonita offers APIs, including a REST API, which this service will utilize. There are many possibilities in Ruby to make calls to a REST API. This implementation uses a Ruby gem, called *rest-client*[19].

Hence, the dependencies of this service are *opcua-smart*, *Sinatra*, and *rest-client*. The installation procedure for *opcua-smart* is described in Section 6.2.3. For the other two, see Listing 6.20.

```
gem install --user-install sinatra
gem install --user-install rest-client
```

Listing 6.20: Installing Sinatra and rest-client (terminal).

The client service consists of a single file (*client_service.rb*), which is located at path */home/wfms/client_service* in the *wmfs-VM*. It can either be started in the terminal or via a desktop shortcut. Both of those files can be downloaded here[20]. Before being able to start the service via the desktop shortcut, the shortcut file must first be made executable (cf. Listing 6.21).

```
cd Desktop/
chmod +x client_service.desktop
```

Listing 6.21: Making the desktop shortcut of the client service executable (terminal).

In case a different location is used for the file *client_service.rb*, the path under *Exec* in the desktop shortcut (cf. Listing 6.22) needs to be adapted accordingly.

```
1 [Desktop Entry]
2 Version=1.0
3 Type=Application
4 Terminal=true
5 Name=client_service
6 Exec=/home/wfms/client_service/client_service.rb
```

Listing 6.22: Desktop shortcut for starting the client service (client_service.desktop).

A more detailed description of the client service follows in Section 6.3.

6.2.4. Workflow Management System – Bonita

As Workflow Management System, Bonita is used. More specifically, the community edition of Bonita (cf. Section 5.4) in version 2021.2. The installation process of this tool

[14] `https://flask.palletsprojects.com/en/2.0.x/` (visited on 21st Feb. 2022)
[15] `https://expressjs.com/` (visited on 21st Feb. 2022)
[16] `https://www.fastify.io/` (visited on 21st Feb. 2022)
[17] `https://rubyonrails.org/` (visited on 21st Feb. 2022)
[18] `http://sinatrarb.com/` (visited on 21st Feb. 2022)
[19] `https://rubygems.org/gems/rest-client/` (visited on 22nd Feb. 2022)
[20] `https://github.com/samhabers92/CollabPatternsCobots` (visited on 29th Apr. 2022)

is straightforward. The first step is to download the installer, which can be acquired here[21]. As the *wfms-VM* is running on a Linux operating system, the Linux version of Bonita must be downloaded. Once this download has finished, run the file and follow the instructions of the installer. After that, Bonita can be started via the desktop shortcut. Note that it may be necessary to make the downloaded file executable, before being able to start it. To do this, open the terminal and go to the directory where the file is located, usually the Downloads folder. Then perform the following command shown in Listing 6.23 (edit the filename in the command to match the downloaded file if needed).

```
chmod +x BonitaStudioCommunity-2021.2-u0-x86_64.run
```

Listing 6.23: Making the Bonita install file executable (terminal).

After the installation has completed, a REST connector has to be installed. Otherwise the import of the process models won't work, since they use this connector. The required connector can be installed via the Bonita Marketplace. The marketplace can be opened by clicking on *New→Extension* and then on *Open Marketplace*. This opens a new window with a list of available extensions. Search for "REST" and install Bonitasoft's *REST* connector.

6.2.5. Import

The Bonita processes and the corresponding cobot programs that have been developed in the context of this work, can be downloaded here[22]. The import of them is pretty straightforward and explained in the following.

URSim programs

The cobot programs were developed for the UR5e type. Each program consists of five files, all named after the program's name and having different file extensions (*.installation, *.script, *.txt, *.urp, *.variables). To import the programs, simply copy all of those files directly into the subdirectory called *programs.UR5* of the URSim installation (e.g., */home/cobot/ursim-5.11.5.1010327/programs.UR5*). After that, URSim (UR5) is able to open and execute them.

Bonita processes

The process models and additional data (e.g., organizational data, business data model, forms, etc.) can be imported in Bonita using a so-called "BOS-archive".

1. Start Bonita
2. *File→Import→BOS archive*
3. Select the downloaded file (e.g., CollaborativeRobots.bos)

[21] `https://www.bonitasoft.com/downloads` (visited on 23rd Feb. 2022)
[22] `https://github.com/samhabers92/CollabPatternsCobots` (visited on 29th Apr. 2022)

4. Click button *Details* >
5. Click button *Extensions* >
6. Click button *Import*

Bonita will show a bunch of validation errors. These can be ignored for the moment, since they will resolve themselves after deploying the Business Data Model. Click on *File→Deploy*, and then on the deploy-button. The deploy-settings should be fine as is. Now the validation errors can be resolved by clicking on *File→Validate*.

6.3. Client service

In this section, different aspects of the client service program are described in more detail. At first, a simple example of an endpoint is shown, followed by an explanation of the OPC UA connection from client to server. After that, different types of OPC UA nodes and their operations are shown. Finally, the procedure of the handshake between client service and cobot, as well as the callback from client service to Bonita is explained.

Endpoint

As stated in Section 6.2.3, the program makes use of the Sinatra framework. Endpoints can be created relatively simple using this framework. Listing 6.24 shows a simple GET endpoint that returns the string "Hello world!".

```
get '/' do
  'Hello world!'
end
```

Listing 6.24: Example GET endpoint.

The client service offers various endpoints using different HTTP request methods (e.g., GET, POST, PUT, etc.). The default endpoint on '/' lists all available endpoints, how to call them, and a short description (cf. Listing 6.25).

```
# Returns available endpoints offered by the service as json.
get '/' do
  content_type :json
  endpoints = { endpoints: [
    { url: '/cobots', method: 'GET',
      description: 'Lists available cobots.' },
    { url: '/cobots/:cobot_id/power_on', method: 'POST',
      description: 'Sends power_on to a cobot.' },
    { url: '/cobots/:cobot_id/node/:node_id', method: 'GET',
      description: 'Returns the value of the corresponding node.' },
    { url: '/cobots/:cobot_id/node/:node_id', method: 'PUT',
      description: 'Sets the value of the corresponding node.' },
    ...
  ] }
  endpoints.to_json
```

```
end
```

Listing 6.25: Default endpoint lists all available endpoints.

OPC Connection to servers

A dedicated class manages the connections to the OPC UA servers of the cobots (cf. Listing 6.26). The class holds all active connections, so that they can be used across all service calls. Whenever a call needs to communicate with a cobot, and there is no established connection to that cobot yet, a new one is created. However, the connections are not closed after each call, as this would add a load of overhead, since there are often multiple calls to a cobot in succession. Therefore, a connection to a cobot is only closed when sending the power off signal to this cobot.

```
class OpcConnection
  ...
  def self.connect(cobot_id, address)
    @@conns.fetch(cobot_id) do |id|
      @@conns[id] = OPCUA::Client.new(address)
      ...
      @@conns[id]
    end
  end
  ...
  def self.disconnect(cobot_id)
    @@conns[cobot_id]&.disconnect
    @@conns.delete(cobot_id)
  end
end
```

Listing 6.26: Class OpcConnection manages the connections to the OPC UA servers.

OPC UA nodes

The OPC UA connection can be used to read and write values, and also to call methods the OPC server offers, such as power on/off, or manage programs (load/start/stop/pause). This follows a general procedure. The values or methods are called *nodes*. These nodes have a unique identifier in the form of a URL path (e.g., */cobot1/PowerOn*). There are different types of nodes, e.g., read/write values, read-only, methods with and without parameters. The "connection" to a node is shown in Listing 6.27, where the variable *client* holds an active OPC connection. If the specified node could not be retrieved, an error message stating that the identifier is invalid is returned. Furthermore, the mentioned operations that can be performed on the nodes are shown.

```
node = client.get(node_id)
halt(404, { message: "Invalid NodeID: #{node_id}" }.to_json) unless node
val = node.value # read value of node
node.value = val # write value of node
node.call        # method call without parameter
```

```
node.call(param) # method call with parameter
```

Listing 6.27: "Connecting" to a specific node and performing read, write, and method calls. Note that this is just an example. Not all of those operations work on every node (different node types).

Cobot handshake

In order to control the program flow of a cobot, a 4-way handshake, as seen in Figure 6.2a, is implemented. The bit registers (cf. Listing 6.16) are used for this procedure. Additionally, the client service is able to determine when the cobot's program progressed to a certain point, and can then perform a callback to the WfMS, signaling that the cobot reached that specific state.

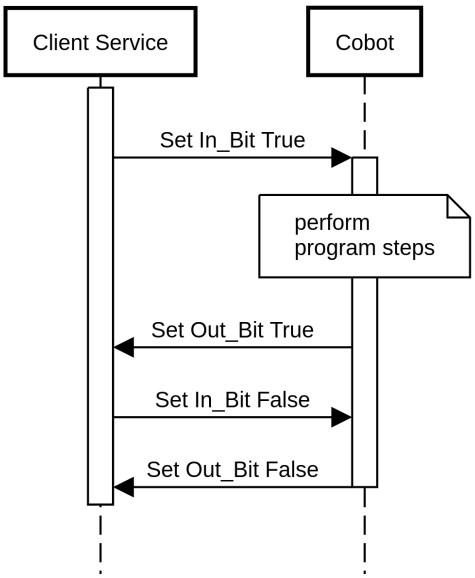
The cobot's program waits, until the value of the input bit register is true. This is triggered by the client service setting that bit, which in turn happens when a process in the WfMS makes a specific request to the client service. After that, the client service waits until the output bit register is true. Meanwhile, the cobot performs its program, which eventually leads to setting that output bit's value to true. Then it waits for the input bit's value to change back to false, after which it finally resets the output bit back to false. On the other side, once the output bit changed to true, the client service resets the input bit to false and waits for the cobot to reset the output bit. This results in the same state of the bit registers as prior to the handshake procedure.

Bonita message-event callback

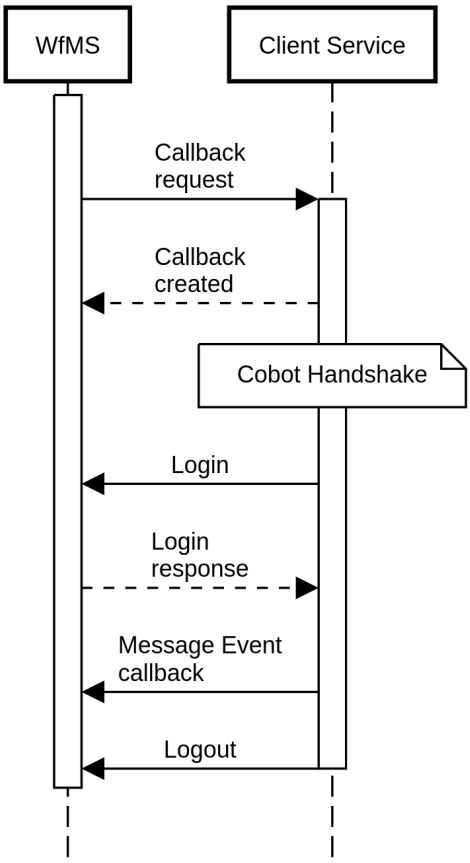
A process diagram can contain elements that model incoming messages. In Bonita, it is called a *catch message event*. Once the process execution reaches such an element, this branch of the process waits for an incoming message that is targeted at this message event. As soon as such a message arrives, the process branch continues. For Bonita to correctly correlate an incoming message to the desired *catch message event*, the message must contain specific information, which is described in the following. The sequence of a message-event callback can be seen in Figure 6.2b. In this work it is used, for example, when a process starts a program step of a cobot and needs to know when the cobot has reached a certain point in its execution. In this situation, the client service immediately answers the request of the process, and later sends a specific message to Bonita's REST API. Detailed information on the authentication process[23] and possible message contents[24] can be found in the Bonita documentation [63].

[23] `https://documentation.bonitasoft.com/bonita/2021.2/api/rest-api-authentication` (visited on 24th Feb. 2022)

[24] `https://documentation.bonitasoft.com/bonita/2021.2/api/bpm-api#message` (visited on 24th Feb. 2022)



(a) 4-way handshake between the Client service and a cobot.



(b) Message-Event callback from Client Service to Bonita WfMS.

Figure 6.2.: Sequence diagrams of the handshake with a cobot and a callback to Bonita.

Login

To authenticate to Bonita's REST API, there must be a user in Bonita's active *Organization* that the client service logs in as. In the example in Listing 6.28, the credentials used are *API* for *username*, and *bpm* for *pwd* (password). To login, a POST request, containing the URL-encoded credentials in the request body, is sent to the address where the Bonita service is running plus the path */bonita/loginservice*. In case of success, the response contains cookies with information that is needed for further requests to the API. Therefore, the login method returns the cookies for use in other requests.

```
# Handles the login process to the REST API of the Bonita Workflow
    Management System.
# Returns the cookies created by the request.
def bonita_login
  username = 'API'
  pwd = 'bpm'
```

```
  payload = "username=#{username}&password=#{pwd}"
  resp = RestClient.post("#{BONITA_URL}/bonita/loginservice", payload)
  resp.cookies
end
```

Listing 6.28: Bonita REST authentication login.

Message callback

Listing 6.29 shows the process of sending a message to Bonita's message API. A POST request is sent to this API, which has the path */bonita/API/bpm/message*. There are certain requirements the request has to fulfill. The headers need to specify the content-type, which must be in JSON format. Furthermore, the so-called *X-Bonita-API-Token*, an information provided in the cookies, has to be set there. And finally, the headers must include the complete cookies of the login request.

```
def bonita_message_event_callback(cookies, message_name, target_process,
    process_instance_id, role_id)
  payload = {
    messageName: message_name,
    targetProcess: target_process,
    correlations: {
      processInstanceIdKey: {
        value: process_instance_id,
        type: 'java.lang.Long'},
      roleIdKey: {
        value: role_id,
        type: 'java.lang.String'}}}
  headers = { content_type: :json,
    'X-Bonita-API-Token': cookies.fetch('X-Bonita-API-Token'),
    cookies: cookies}
  resp = RestClient.post("#{BONITA_URL}/bonita/API/bpm/message", payload.
    to_json, headers)
end
```

Listing 6.29: Bonita message callback.

In order for Bonita to assign the message to the correct *catch message event*, the data provided in the request body is essential. The field *messageName* needs to match the name set in the *Catch message* property of the targeted *catch message event* in Bonita. The value set in *targetProcess*, specifies the name of the process where this event is located at. With this information, Bonita is able to assign the message to the corresponding message element. However, this may not be enough information in certain circumstances, for example when multiple instances of such a process are running. To add further elements to distinguish between events, data can be added under *correlations*. The matching of these aspects is user-defined, and can be different for each *catch message event*. In this example, there is additional matching on the process instance identifier, and another string called *roleIdKey*. The first one is used to distinguish between different process instances, whereas the second one is used to identify messages with the same name across different process actors. The maximum number of supported keys in *correlations* is five.

It is also possible to send data in these messages. This can be done by adding the element *messageContent* and inserting the information there. More detailed information, e.g. on supported value types, can be found in the documentation [63].

Logout

The logout procedure is straightforward and is performed by a GET request to the logout service (Bonita URL + */bonita/logoutservice*). This request needs to contain a header called *cookies* that holds the cookies generated at the login request.

```
# Handles the logout process from the REST API of the Bonita Workflow
    Management System.
def bonita_logout(cookies)
  headers = { cookies: cookies }
  resp = RestClient.get("#{BONITA_URL}/bonita/logoutservice", headers)
end
```

Listing 6.30: Bonita REST authentication logout.

6.4. Bonita

In the following, some aspects of the Bonita Workflow Management System are discussed. The main elements mentioned here are the *Organizations*, *Business Data Model*, and *Forms*.

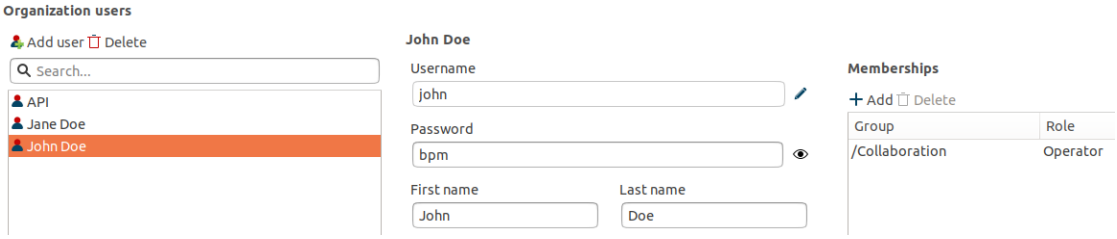
Organization

The *Organization* depicts the structure of the business. The actors that take part in the processes are modeled here. There are three major elements. *Organization groups* are used to specify different parts of the organization, e.g., Production, Administration, IT, etc. The next element are *Organization roles*, which show the roles that people can have inside different groups. And then there are *Organization users*, which depict actual users of the system. Each user can be assigned to multiple groups and fulfill different roles for each of them. Two examples can be seen in Figure 6.3.

Business Data Model

The data objects that are used and shared by processes are defined in the Business Data Model. These data objects are persistent and stored in a database. There are some Bonita specific peculiarities, e.g. regarding composition and aggregation, that can be looked up in the documentation [63]. In the implemented processes, business data objects are for example used to save and later work with the responses of REST calls to the client service. Another use is to hold general information about the process and process settings like the process name, URL of the client service, names of corresponding cobots and cobot programs, etc.

(a) Bonita *Organization user* the client service uses in the callbacks.



(b) Bonita *Organization user* of a human process actor (operator).

Figure 6.3.: Screenshots of two Bonita *Organization users*.

Forms

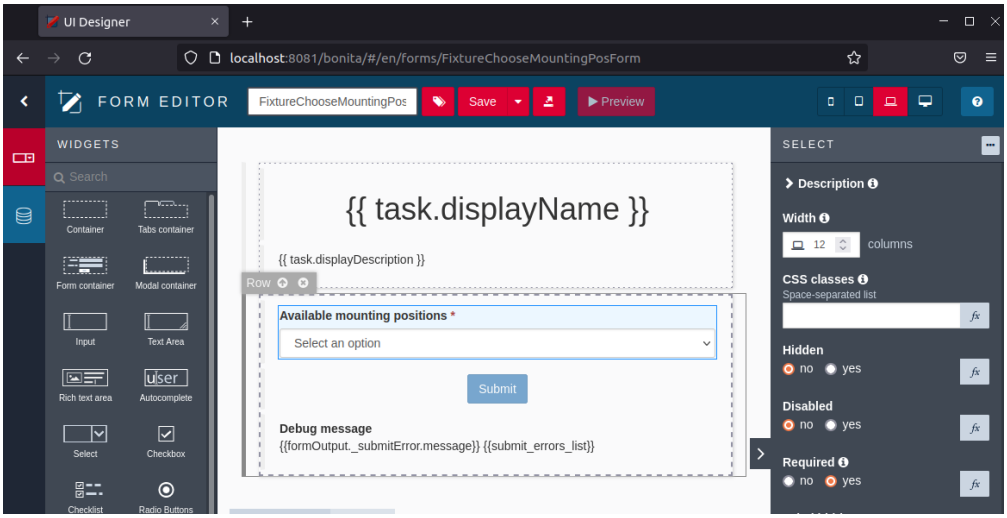
Forms are an important part in the execution of processes in Bonita. There are three types of forms in Bonita [63]. The first form a user sees is the process instantiation form. This form is shown when a user wants to start a new process instance, a so-called case. Once the user submits this form, Bonita creates a new case of the chosen process. The next form type is the human task execution form. These forms are used to execute human tasks. They display information, and can provide different types of input (e.g., textual input, checkboxes, select fields, etc.). The data input from the user then may influence the further execution of the case. The third type is the case overview form. This form shows a chronological history of the case (i.e., which tasks were executed, when and by whom were they performed), the business data used in the case and current values of them, and the documents attached to the case [63].

Bonita provides a tool to design those forms. This tool is called *UI Designer* and comes with different predesigned widgets and access to the Business Data Model. The forms are shown in a browser, and as such they are based on basic web forms (i.e., HTML, JavaScript, CSS). The default design is based on Bootstrap[25] [63]. This design can be changed with other themes, or completely customized CSS classes. Figure 6.4 shows an example of a form in the UI Designer and the same form during process execution.

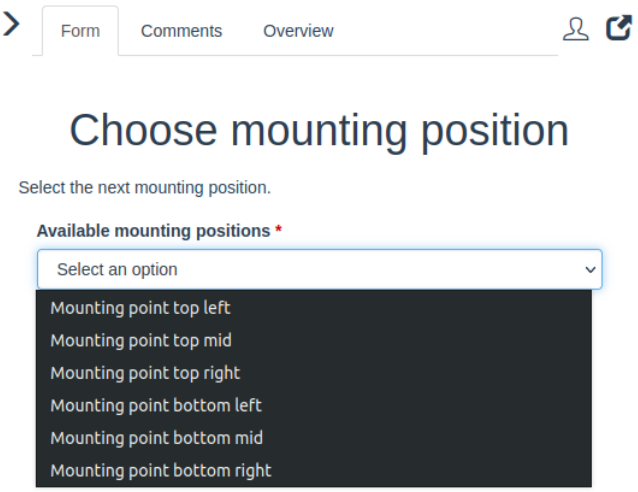
6.5. Cobot program in URSim

The program depicted in Figure 6.5 shows the general structure of the cobot programs implemented in this work. The whole program is set to loop forever until it is externally

[25] `https://getbootstrap.com/` (visited on 1st Mar. 2022)



(a) Form in Bonita's UI Designer.



(b) Form during execution of a process.

Figure 6.4.: Screenshots of a form in Bonita's UI Designer and during process execution.

stopped. In lines 2 and 40–42, the handshake procedure, described in Section 6.3 and Figure 6.2a, can be seen on the side of the cobot. The program waits on line 2 until the input bit register 64 is set to high (true). After the execution of the current program step, the output bit register 64 is set to true, and then the program waits for the input bit register to be reset. Subsequently, the output bit register is reset and the cycle continues. In between the handshake, the actual program takes place. To allow the process in the WfMS to dynamically control the cobot's program, a program step variable is used. This variable is a simple integer value and is updated by the process prior to starting the

handshake procedure. Based on the current value of it, a different part of the program is executed.

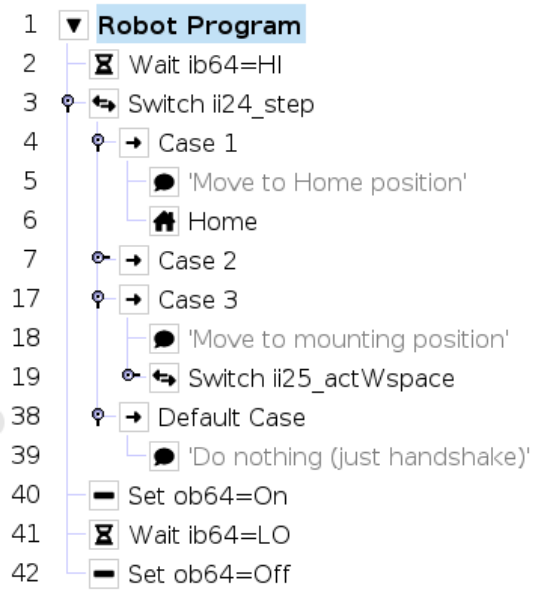


Figure 6.5.: Example of a simple cobot program in *URSim*.

6.6. Startup procedure

In order to use the setup to simulate the implemented processes, the corresponding programs must be started beforehand. For the *wfms*-VM, these are the *client_service* program and the Bonita engine. The *client_service* can be started with the desktop shortcut or directly via terminal. Either way, the application runs in a terminal window. The Bonita engine starts automatically when opening the Bonita Studio application (desktop shortcut). After that, open a browser window and go to `http://localhost:8080` and log in with the username *john* or *jane* and password *bpm* (cf. Fig. 6.3b).

On the side of the *cobot*-VM, *URSim* and the OPC UA server must be started. Here the order does matter, i.e., the cobot simulation must be started first. This can be done with the desktop shortcut for the *UR5* version of the cobot (e.g., *ursim-5.11.5.1010327 UR5*). It is important to choose the *UR5*, because the programs have been created for this variant, so they only show up when this cobot type is running. Once the cobot simulation is up and running, the OPC UA server can be started via the desktop shortcut *URUA Devserver*. Now the test setup is ready to execute processes.

7. Evaluation

For evaluation, seven exemplary use cases have been implemented. This chapter gives an overview on all the implemented use cases. First, the modeling approach is showcased by discussing the process model of the "Spot taping" use case. Furthermore, problems and challenges that may arise when including multiple human and robotic actors in the process model are discussed, and a potential solution to the problem is proposed. Consequently, the remaining use case implementations are presented. As previously mentioned, BPMN is used for modeling the processes. The process models are created in Bonita Studio. For more information about this tool, refer to Section 5.4.

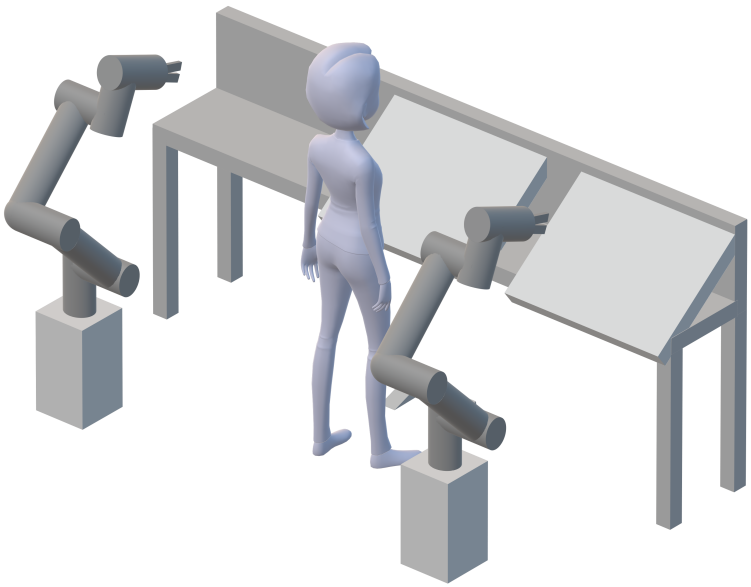
7.1. Use case implementation

All the discussed collaborative patterns were successfully demonstrated using the combination of Bonita (WfMS), URSim (cobot simulation), and OPC UA (communication). The test setup described in Chapters 5 and 6 has been used for the implementation of seven use cases across these patterns and demonstrates the applicability of the approach. There is one implemented example for the coexistence pattern, and two for each of the patterns synchronized, cooperation and collaboration.

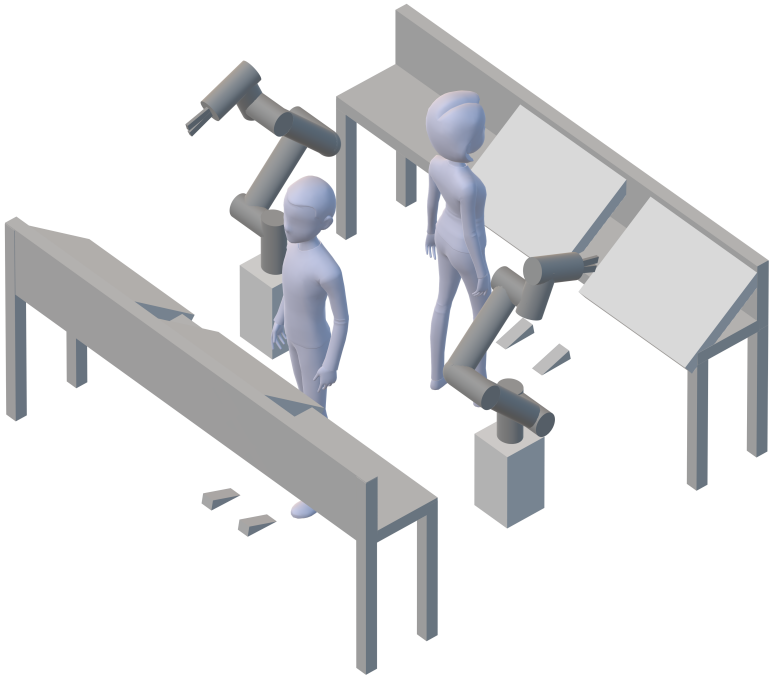
- Synchronized — Spot taping, Extended spot taping (cf. Sec. 7.1.1)
- Coexistence — Pick & Place (cf. Sec. 7.1.2)
- Cooperation — Quality inspection (cf. Sec. 7.1.3), Polishing (cf. Sec. 7.1.4)
- Collaboration — Fixture (cf. Sec. 7.1.5), Collaborative assembly (cf. Sec. 7.1.6)

This section provides an overview of these implementations, including aspects such as:

- user interactions with the Bonita User Application
- process models
- some details on the activities used in the process models (e.g., how to make REST calls, read/manipulate data, call subprocesses, etc.)
- cobot simulation pictures and references to the cobot programs in Appendix A.1



(a) Design of the spot taping work space described in [58] (own representation).



(b) Design of the spot taping work space scaled to produce four wire harnesses at the same time as described in [58] (own representation).

Figure 7.1.: Custom work cell layout developed for the Spot taping process by the authors of [58]. The two images are own representations created in Microsoft Paint3D. The operator figures are standard 3D models provided by Microsoft Paint3D.

7.1.1. Spot taping & Extended spot taping

The Spot taping process modeled in this section is a simplified variation of a use case described and implemented in [58]. Their goal was to improve the current process for spot taping wire harnesses, which was performed manually, by adding robotic arms that collaborate with the human worker. A custom work cell was developed, and for the implementation a prototype of the cell was created (cf. Figure 7.1a) [58]. The human worker sits in front of a desk, where two working areas are mounted on a linear axis. The linear axis has two defined positions. One where the left working area is in front of the worker, and one where the other area is in front of the worker. The operator has two foot pedals that can be operated to send a signal indicating that the linear axis can be moved to the left or to the right. On both sides of the worker, a cobot is placed. These

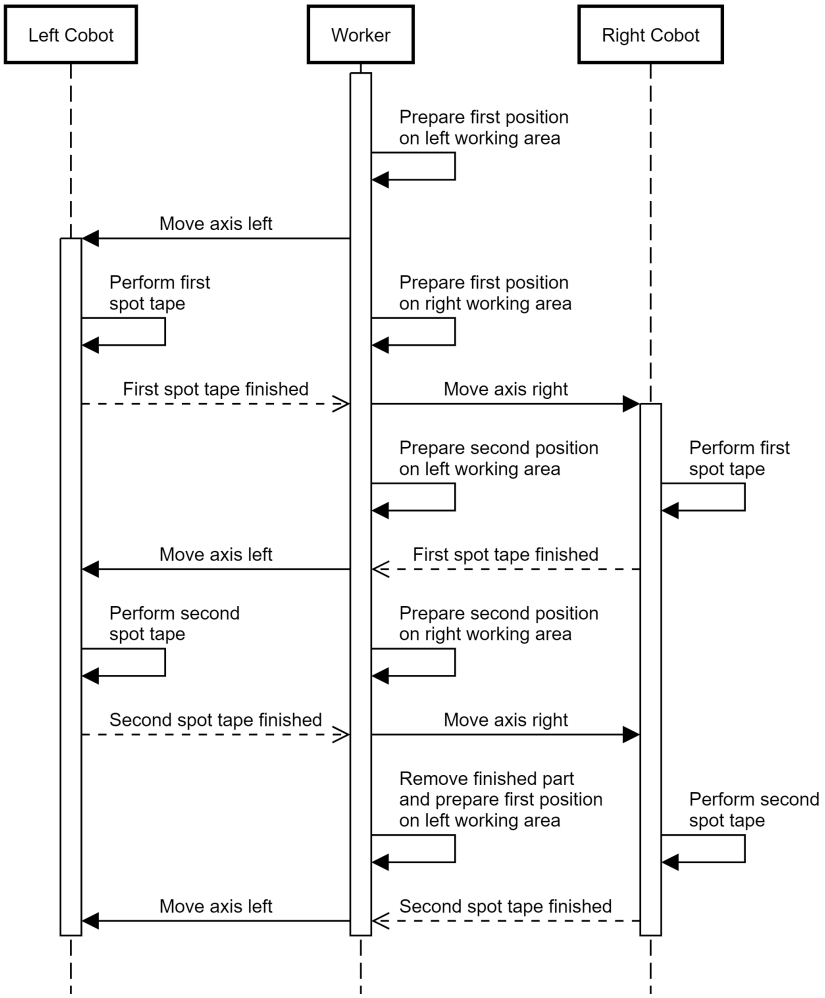


Figure 7.2.: Sequence diagram of the Spot taping process described in [58] (own representation).

cobots perform the spot taping of the wire harnesses. In each production cycle, two wire harnesses are processed. One on each of the two working areas. When the linear axis is positioned on the left side, the left cobot has access to its working area, while the worker prepares things on the right working area. Once both are finished, the axis can move to the right, and the worker performs tasks on the left working area, while the right cobot has access to the right working area. A more detailed description of this production cycle's sequence is shown in the diagram in Figure 7.2. The cobot programs are depicted in Figures A.4 and A.5.

Since the BPMN diagram is too large to be displayed completely on one page and still be easily readable, different parts of the diagram are displayed in individual figures. The complete process model is then shown in Figure 7.7. First of all, there is a slight difference in the naming of pools in Bonita, as opposed to the standard BPMN convention. Usually, a pool represents a process participant, e.g., name of the company, name of the department, etc. In Bonita, however, the name of the pool has to be unique within a Bonita project, and represents the name of the process modeled within the pool [63]. In this case, the process, and therefore the pool, is called *Synchronized_SpotTaping*. The pool contains four lanes. The left and right cobots, the operator and the linear axis.

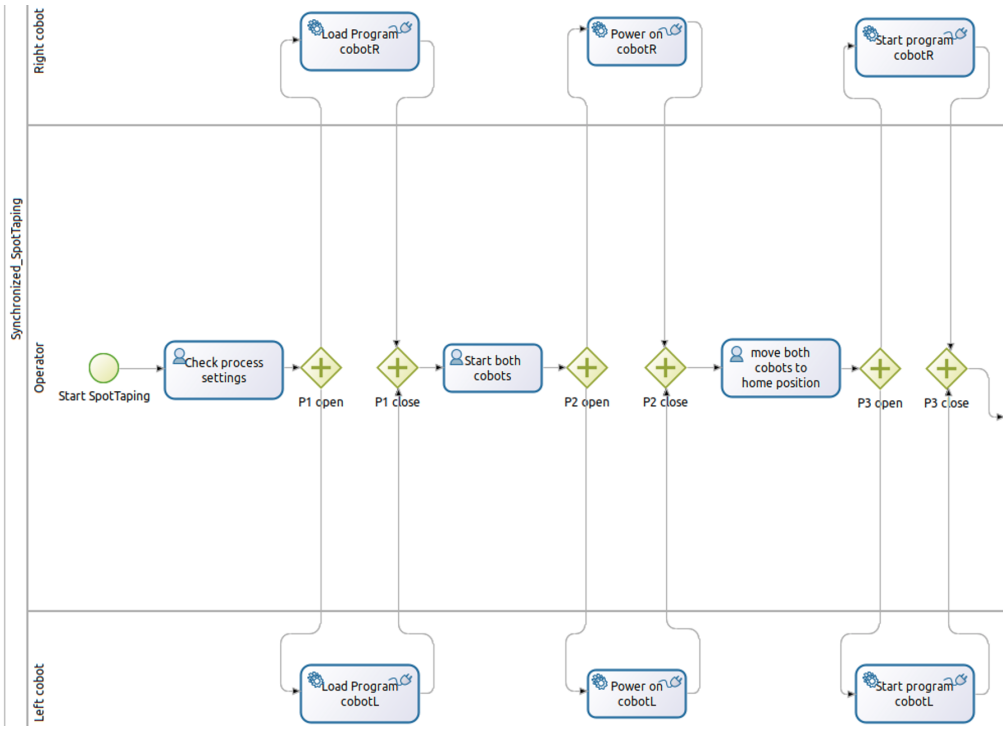


Figure 7.3.: Initial tasks of the Spot taping process model.

The beginning of the process model is similar across most of the implemented use cases in the context of this work (cf. Fig. 7.3). As a first step after starting the process, there is a user task called *Check process settings*. This task provides a form with various

settings that are used in the process, e.g., URL of the client service, names of the cobots, and other process relevant settings like position offsets, list of tasks, etc. These settings are predefined and can be submitted as is, but it is also possible to make changes to them when performing this task. In the following activity, which is a service task, the corresponding cobot program is loaded.

Figure 7.3 shows that there is an activity for loading the program for each of the two cobots, which are executed in parallel. These two activities perform a REST call to the OPC client service, specifying the program to load for the corresponding cobot simulation.

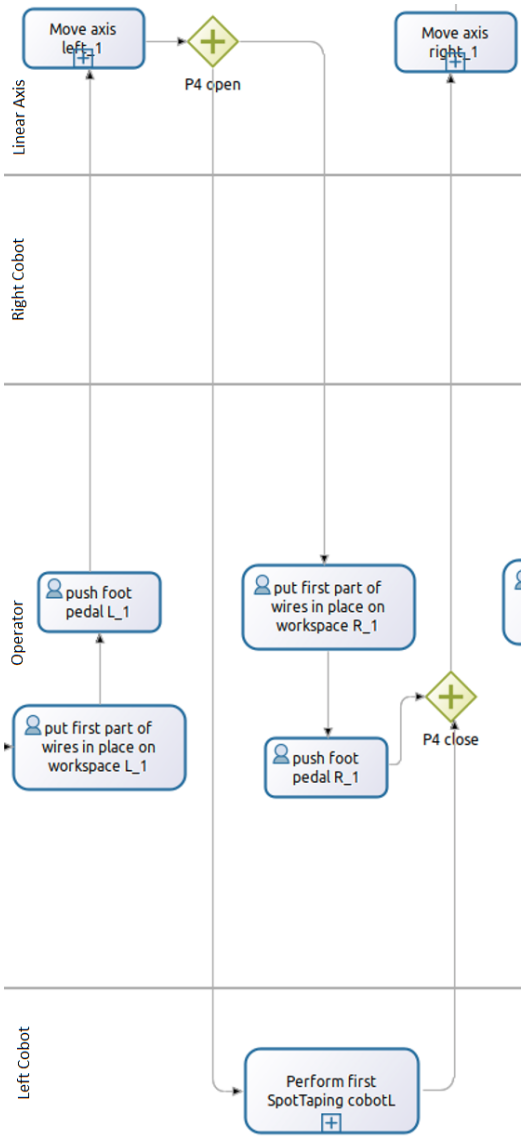


Figure 7.4.: First part of the Spot taping production cycle.

The client then sends a command to the OPC server of each cobot, which results in the cobot loading the program. The next step is likely only relevant in the simulated test setup of this work. The operator has to switch on the cobot simulations manually once a new program is loaded, which puts them in idle mode. After that, they can be fully powered on automatically via the OPC server. This happens similar to the loading of the programs, by executing the *Power on* service tasks. The following task, which is also only relevant to this test setup, is to manually move the cobots to their home position. This has to be done, because the cobot program can only be started when the cobot is in a predefined position. Since this position varies between the different programs, this step has to be done to ensure that the program can start successfully. Afterwards, the workflow engine can send the command to start the cobot programs. Once this step has been performed, the actual spot taping process can start.

The first steps of a production cycle in the Spot Taping process are shown in Figure 7.4. The process model shows the same behavior as the sequence in Figure 7.2 suggests. At first, the operator has to perform two tasks, the preparation of the left workspace for the first spot taping, and after this task is finished, operating the left foot pedal. Subsequently, the linear axis moves to the left. Now the operator and the left cobot can work on their tasks simultaneously (parallel gateway *P4 open* opens two paths). The operator repeats his two previous tasks, with the difference that this time the tasks are performed for the right workspace and the right foot pedal must be pushed. In the meantime, the left cobot performs the first spot taping on the left workspace. As soon as both actors finished their respective tasks, the two paths are joined by the closing parallel gateway *P4 close*, and the linear axis can move back to the right.

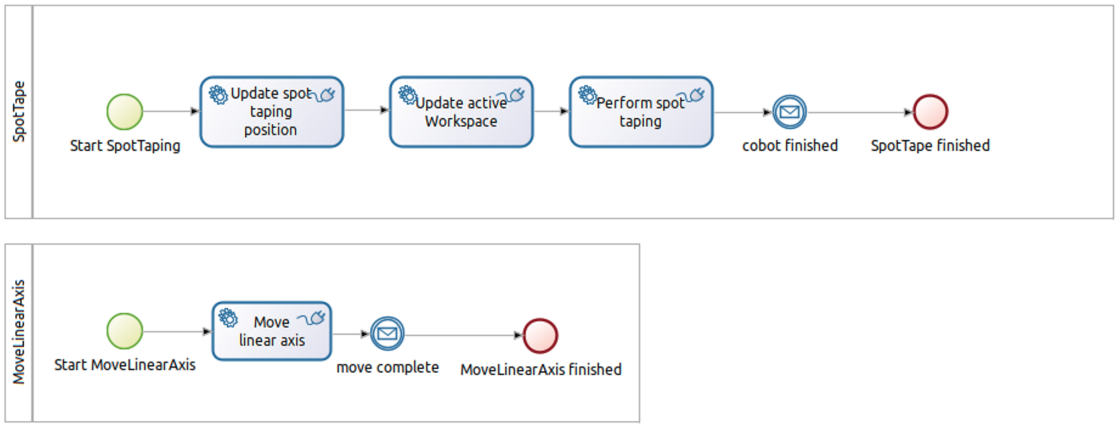


Figure 7.5.: Depiction of the two subprocesses used in the Spot taping process model.

As one may have noticed, the activities of the left cobot and the linear axis in Figure 7.4 are subprocesses. In Bonita, the name of this type of task is *call activity*, since it calls a specific subprocess. In this process model, two different subprocesses are used. One reason to use such subprocesses is to simplify the process model. The *SpotTape* subprocess is called five times in the spot taping process model, twice for the right cobot and three

times for the left one. *MoveLinearAxis* is called six times, three times to move the axis to the left and three times to the right. Another reason for the use of subprocesses is the added reusability. In case a certain functionality requires a specific set of tasks, that may only differ through the variables used within those tasks, a subprocess for this functionality can save lots of time during the development of process models. Additionally, if something changes in this set of tasks, it only needs to be modified at one place, and not in every instance it is used separately.

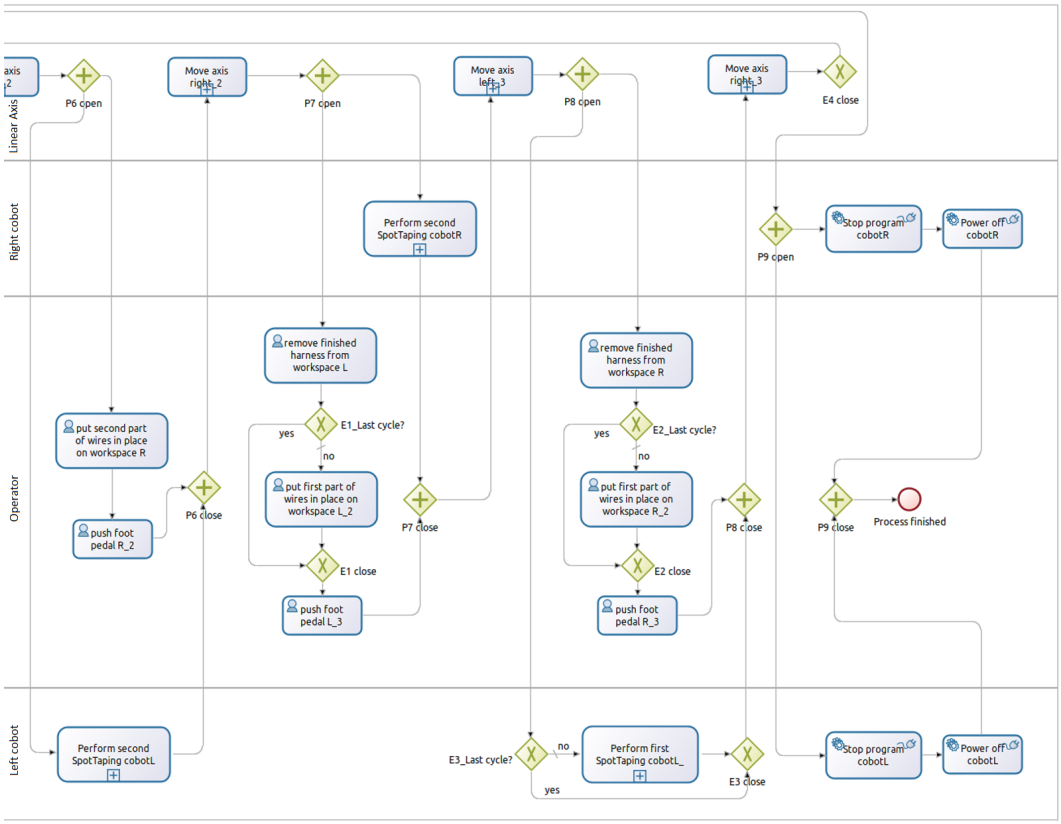


Figure 7.6.: Larger segment of the Spot taping process model. It already shows a lot of sequence flow connectors across the different actors in the process, but it is still easily readable.

The two subprocesses, shown in Figure 7.5, look and behave rather similar. After starting them, there is either one service task, in case of *MoveLinearAxis*, or a sequence of three service tasks in the *SpotTape* subprocess. The first two tasks of *SpotTape* update some variables of the cobot simulation, which are a position offset and the active workspace. These two in combination determine the position where the cobot needs to perform the spot taping. The third task tells the cobot to start the spot taping operation. In more detail, this task triggers a specific handshake, which is described in Section 6.3. Then the subprocess waits for an incoming message on the *catch message event* called *cobot finished*.

Once the corresponding message arrives, the subprocess finishes and with that, the call activity (e.g., *Perform first SpotTaping cobotL*) in the main process finishes as well. The execution of *MoveLinearAxis* is very similar, except another endpoint of the client service is called. This one also waits for an incoming message to finish its execution.

The process continues by alternating work steps between the left and right cobot. This behavior can be seen in the sequence diagram in Figure 7.2 and also in the snippet of the BPMN model in Figure 7.6. So the first question that arises immediately is, how can those downtimes of the two cobots can be utilized to further improve the efficiency of the process. And indeed, this aspect was already taken into account in the development of the custom work cell [58]. The design has been made with scalability in mind and allows for more complex setups with just slight modifications to it [58]. The authors propose two modified versions. The first one doubles the amount of simultaneously processed wire harnesses to four units [58]. This is achieved by adding an additional operator to the process, and adding a mirrored work cell back to back, so that the cobots can reach both of the work cells, which can be seen in Figure 7.1b [58]. This way, the downtime of each cobot is utilized, and the productivity is improved while the cycle time remains roughly the same [58]. All of that is possible without adding more cobots to the design [58]. The second proposed approach, includes four cobots and four operators to further increase the throughput [58]. This approach arranges the work spaces in a square with one cobot on each corner [58].

Extending the process - a challenge

In order to approach the domain of multiple operators and cobots in the modeling of a collaborative use case using BPMN to create a process model, the first approach of introducing a second operator to the spot taping process has been implemented, which leads to the **Extended spot taping** use case.

During the development of an initial draft of the new process model, a few problems quickly became apparent. Naturally, the first approach when extending a process by additional actors is to modify the already existing process model. For this, two new lanes have to be added. That is, a second operator as well as a second linear axis, along with their required elements (e.g., activities, gateways, etc.). However, this raises the following problems:

- Naming — In some tools, the names of the different elements inside a process model have to be unique. This introduces a problem when there are multiple of the same tasks in different locations of the model. The names must be changed in some way to be unique (e.g., *push foot pedal L_ 1*, *push foot pedal L_ 2*, *push foot pedal L_ 3* in Fig. 7.4 and 7.6). This decreases the readability of the model. When adding the new elements for the extension of the process, basically all of them would require names that are already in use. So, a lot of changes to the naming must be made. Additionally, also the names of the existing elements have to be adapted to clarify the meaning, e.g., what does *workspace L* or *workspace R* mean when there are four possible workspaces now.

- Space — A consequence of extending the current process model by additional actors is, that there are several new elements that have to be added to the model. In the lanes of the new actors, there is plenty of space available. However, also the lanes of the existing actors are influenced, e.g., new things like gateways have to be added. To fit those additional elements, more space is needed. Modeling tools may provide the means to achieve this. In Bonita, one can click on a specific spot inside a lane and insert or remove new modeling space by dragging the mouse in the corresponding direction. This moves all elements to the right of the mouse cursor into this direction. The problem is that this is only possible per lane and not for all lanes at once. Hence, doing this completely messes up the layout of the sequence flow connectors. Adjusting all other lanes to fit the changed one only fixes this to a certain extent. Ultimately, the modeler has to fix the connectors manually. However, this might be more of a problem related to the modeling tool, as other tools might provide better functionality to deal with this.

- Overlapping connectors — Depending on the process, the addition of new actors will introduce overlaps in the different types of connectors, especially the sequence flow connectors between flow objects (activities, gateways, etc.). For very simple processes and when only a small number of actors are added, this may not be the case or a problem. But for scalability, this is definitely a major problem. In the example of the spot taping process (cf. Fig 7.6), adding only one additional operator leads to a good amount of unavoidable overlaps, due to sequence flows between tasks of each actor to both cobots. Dealing with these overlaps to keep the model reasonably readable is a tedious task for the modeler. The larger the scale becomes, the more overlaps occur, and eventually the model becomes near impossible to read and understand.

This approach is not suitable for scalability because it leads to immensely complex models that are difficult to read and understand, which is also a major problem for the maintainability of the process. Also, the process of extending the model itself is a tedious and error-prone task.

To address these issues, the extension of the spot taping process was achieved by splitting the process into two separate process models. In order to coordinate the two models, they are synchronized at certain steps using *message events*. There are two different kinds of synchronization points. The first one is regarding the cobots performing their process steps. For example, when *cobot1* starts by working on the left working area of *operator1*, then the process model of *operator2* can only start requesting *cobot1* after it is finished with the current operation. The second type deals with determining the last process cycle. Here, the first operator takes over the command and indicates whether the current cycle is the last one. If that is the case, the following user tasks are different for both operators. Hence, the second operator needs the decision of *operator1* to advance beyond this point in the process. This approach allowed to reuse the original process model with only minor modifications (mainly adding the *message events*), which can be seen in Figure 7.8. The process model for the second operator is basically a copy of the

first one, with modified process settings and removed initial and final activities of starting and stopping the cobots (cf. Figure 7.9).

Splitting the process into separate models made it fairly easy for the modeler to extend the process to two operators and two cobots. In the same way, it can be used to further scale up the process to four operators and four cobots, for example. However, it also has certain limitations and introduces some potential problems. The fact that one physical actor is present in multiple separate models which are executed together, demands for specific handling in the process flow. A single cobot, for example, can't perform multiple activities concurrently. Therefore, the implementation needs to take care of this situation. There are multiple ways to deal with this problem. In the current example of the spot taping process, synchronization points in the process models were used to avoid creating a situation where a request is sent to a busy cobot. Another downside of the current process is the rather strict dependency between the two operators. If both operators work at a similar speed, there is no problem. But as soon as one of them is significantly slower, the faster operator is going to experience waiting times. This could be solved by decoupling the process models further, i.e., not using synchronization messages for the cobot tasks, and instead let each model request the cobot as soon as possible. This, of course, requires a mechanism to pool the waiting cobot tasks.

7.1.2. Pick & Place

This implementation of a Pick & Place process refers to the use case described in Section 3.1. It consists of a single operator and a cobot. The process starts with the operator checking the part box, from which the cobot is picking parts. The operator's task is to change or refill the box, and to check for other possible problems in case that the box is not empty. Once this is done, the operator gives the cobot a start signal. With this, the operator can move on to work on something different. From now on, the process is performed by the cobot. It starts by moving into a predefined position to scan the available parts. The detection system then scans the box and sends a pick-up position to the cobot. After receiving the position, the cobot picks up this specific part and deposits it at the destination. This is repeated as long as the part detection system is able to detect pickable parts. As soon as this isn't the case anymore, the cobot moves into its home position and stops the program, and the operator is notified to check the situation.

The process model of this example is shown in Figure 7.10. Most of the activities in this and the following process models are either *Human tasks* or *Service tasks*. Note that the BPMN standard divides *Human tasks* into *User tasks* and *Manual tasks*, whereas in Bonita, there is only the combined *Human task*. The business data, which is produced and used by the process instances, must be added to the *Business variables* of the pool (cf. Figure 7.11). The data types that are used here, are defined in the *Business Data Model*, which is described in Section 6.4. The initialization of these variables can also be set here, which is performed on every new process instantiation (cf. Figure 7.12). The script window in Figure 7.12b is used in different locations in Bonita (e.g., data handling, REST connectors, etc.) and allows adding code in Groovy, which has a similar syntax as Java [63].

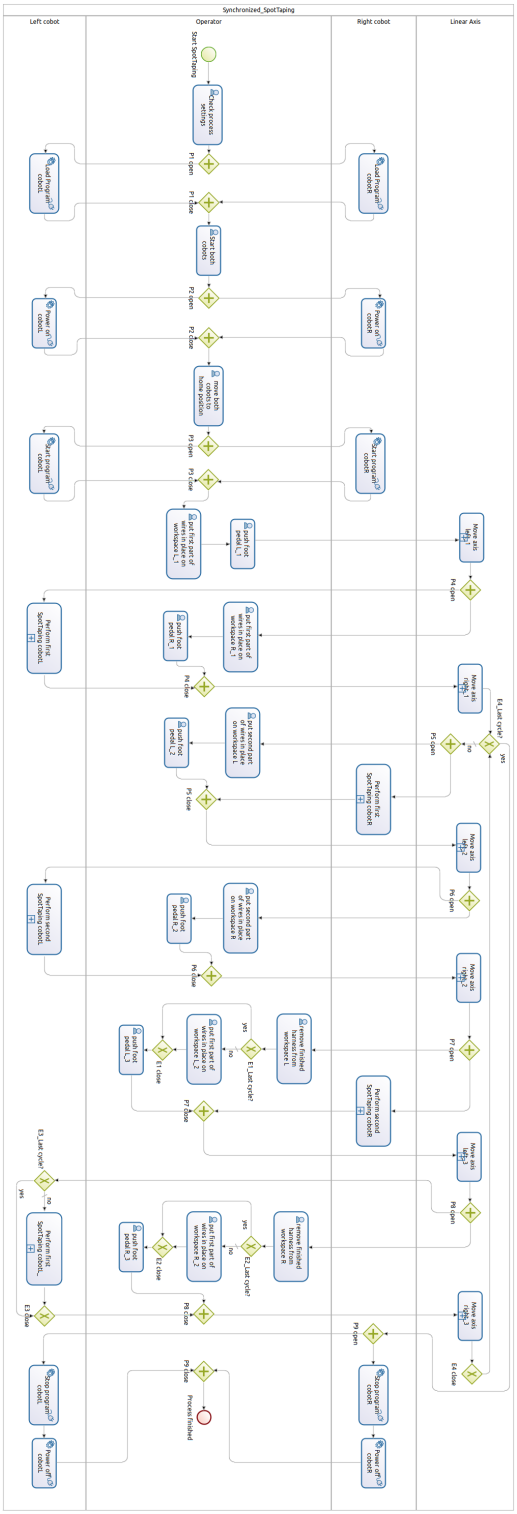


Figure 7.7.: Complete process model of the Spot taping process.

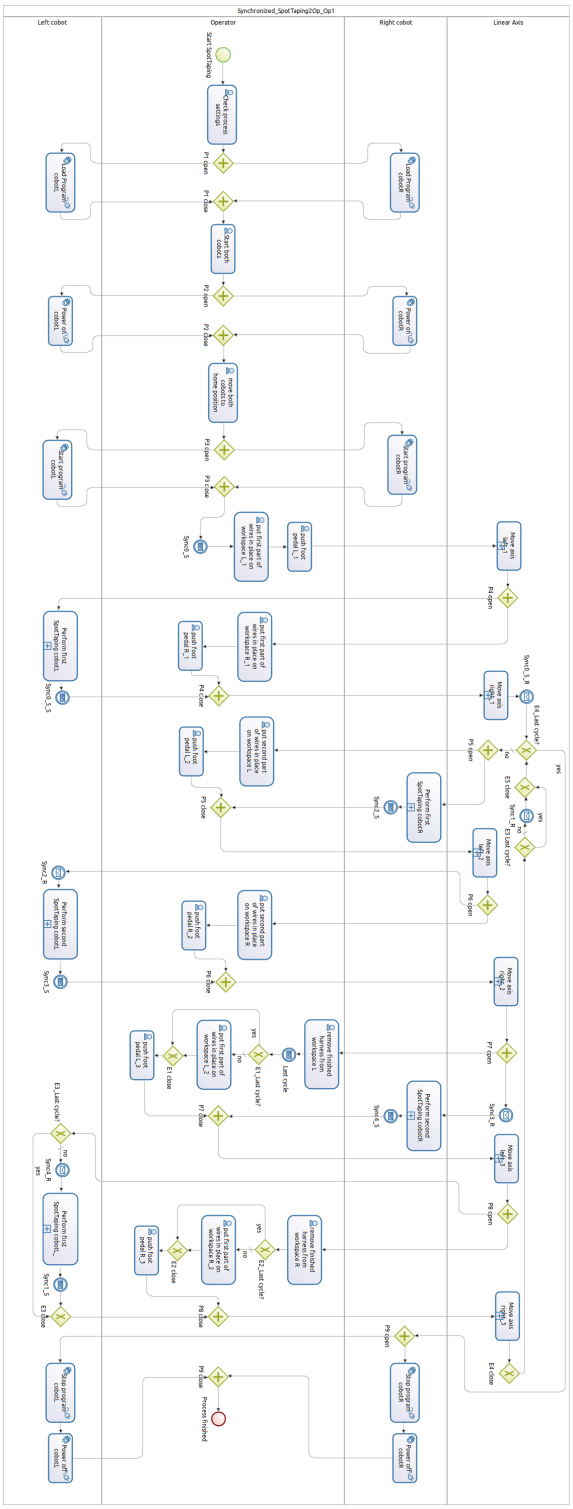


Figure 7.8.: Process model for the first operator of the Extended spot taping process.

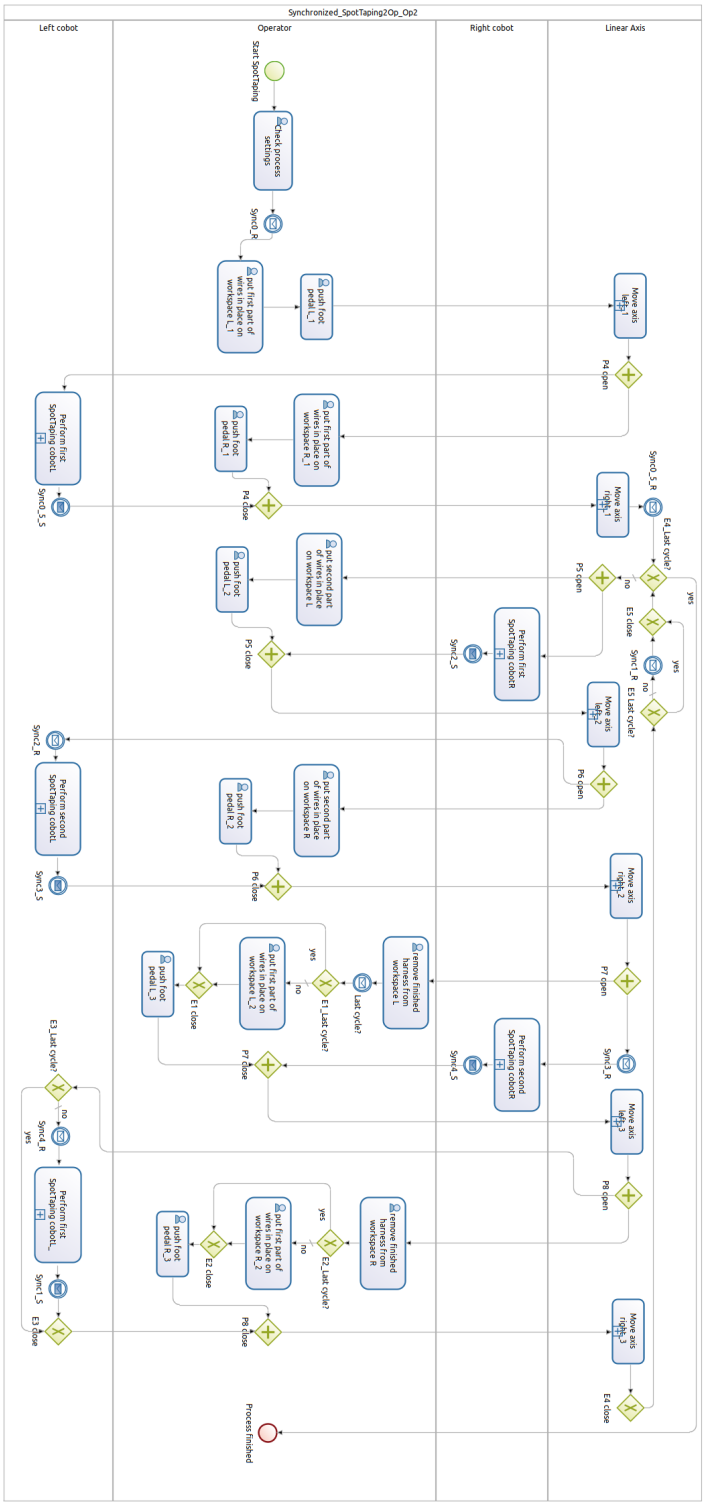


Figure 7.9.: Process model for the second operator of the Extended spot taping process.

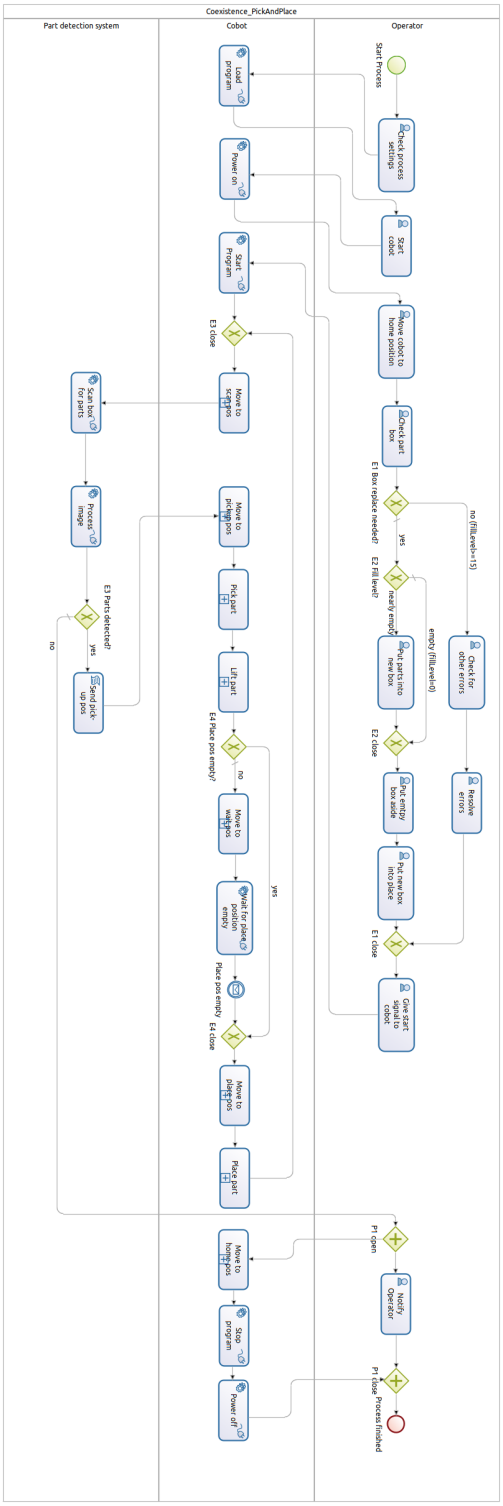


Figure 7.10.: Process model of the Pick & Place process.

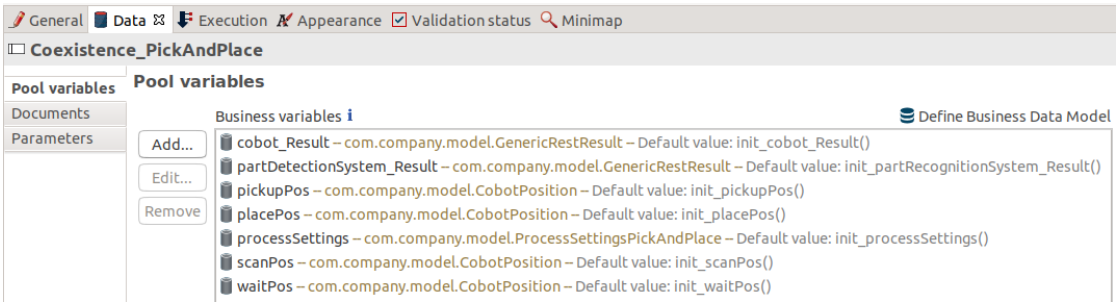
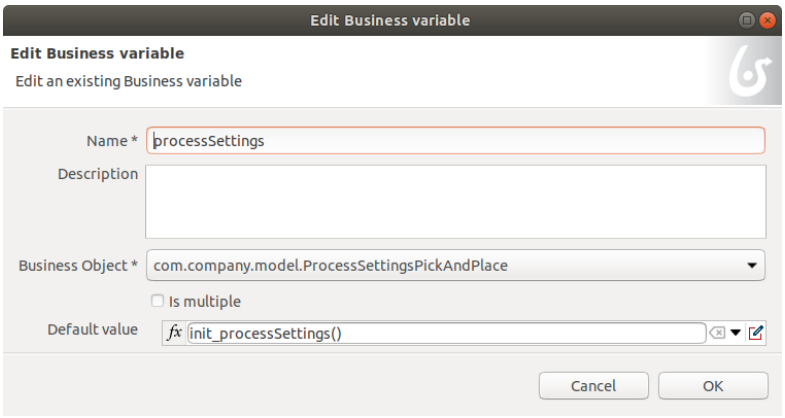
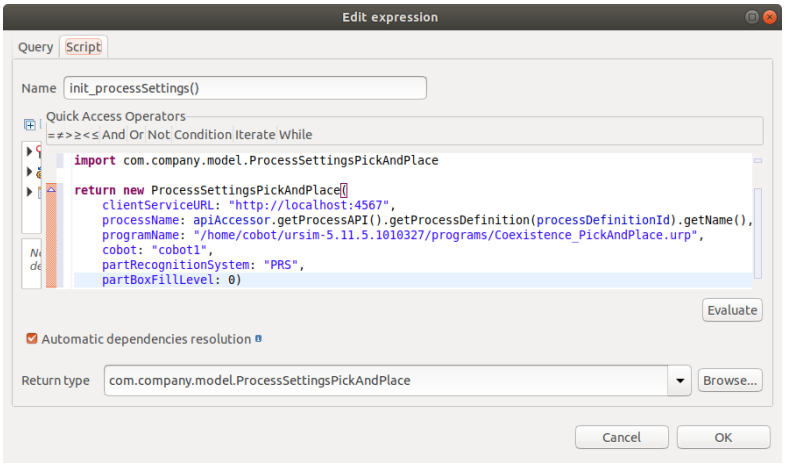


Figure 7.11.: Business variables of a process model in Bonita (screenshot).



(a) Business variable edit window (screenshot).



(b) Business variable initialization function (screenshot).

Figure 7.12.: Initialization of business variables in Bonita Studio.

In order for Bonita to determine how to map the actors of the lanes of the process model to the actors defined in the *Organization*, each process model needs to configure an actor mapping (cf. Figure 7.13). The configuration of this mapping is quite flexible and can be based on *Organization groups*, *roles* or distinct *users*.

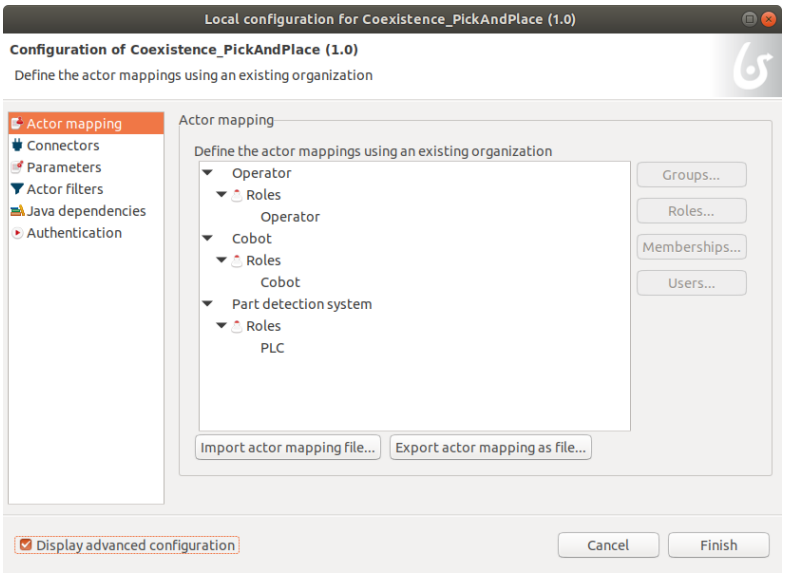


Figure 7.13.: Configuration of the actor mapping of a process model (screenshot).

Human tasks are submitted through forms in the *Bonita User Application*, which is a web browser application. The form used by an activity must be linked to it. To do this, the target form of the activity has to be selected. This can be seen in Figure 7.14. If there is data that should be shown in the form, it has to be specified in a *Contract* (cf. Figure 7.15). In case that data can also be entered or manipulated, additional *Operations* must be added, which map the form inputs to the actual business data objects (cf. Figure 7.16).

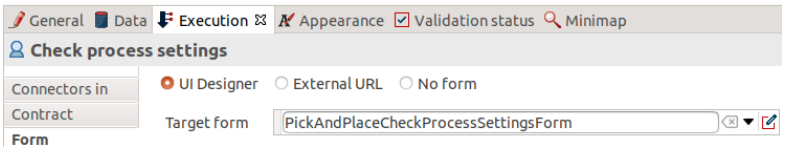


Figure 7.14.: Target form selection of an activity (screenshot).

Subprocesses are used across all process models of the implemented use cases. Bonita provides an easy way to utilize these subprocesses with the *call activity* (cf. Figure 7.17). The process to call can be entered directly or can be selected via a dropdown menu (cf. Figure 7.17a). Furthermore, data can be easily exchanged between the calling process and the called subprocess. This can be configured in the *Execution* tab of the *call activity* under *Data to send*/*Data to receive* (cf. Figure 7.17b).

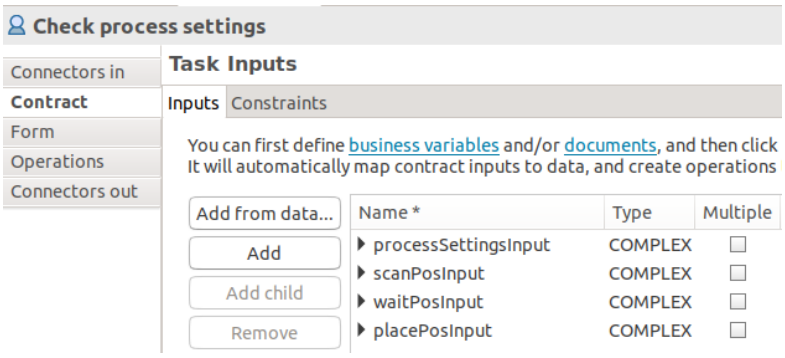


Figure 7.15.: Activity contract definition (screenshot).

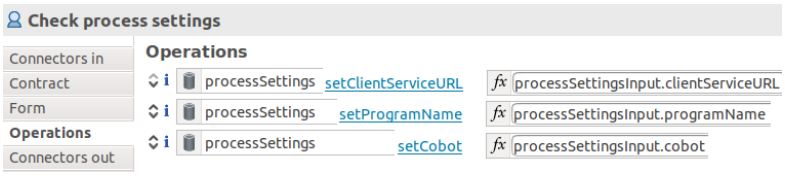
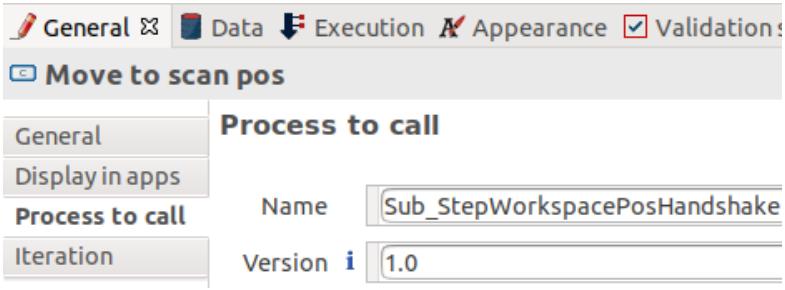
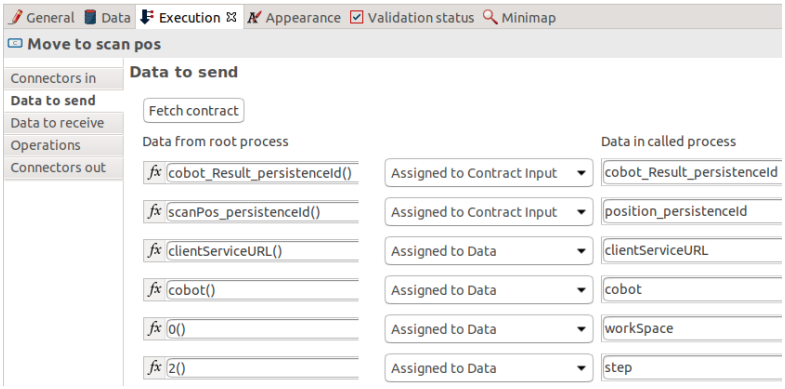


Figure 7.16.: Activity operation definition (screenshot).



(a) Specifying the called process (screenshot).



(b) Sending data to a subprocess in Bonita (screenshot).

Figure 7.17.: Calling a subprocess in Bonita.

In the Pick & Place process model, and also in the subprocesses, *catch message events* are used. These events wait for specific incoming messages, which need to be configured. The configuration can be seen in Figure 7.18. The message must have a specific name, which is unique in the scope of a single process model. Optionally, it can have additional correlation keys, for example, a process instance identifier. This identifier is needed if multiple instances of a process are running at once, so that the Bonita engine can correlate messages to the correct process instance.

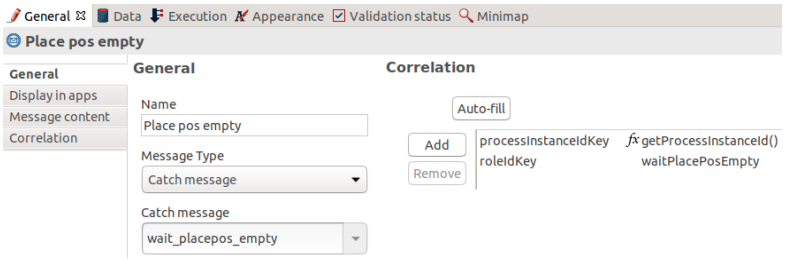
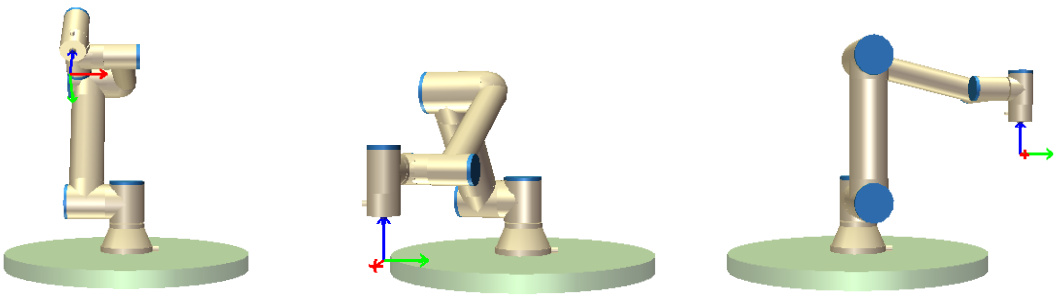


Figure 7.18.: Configuration of a catch message event (screenshot).

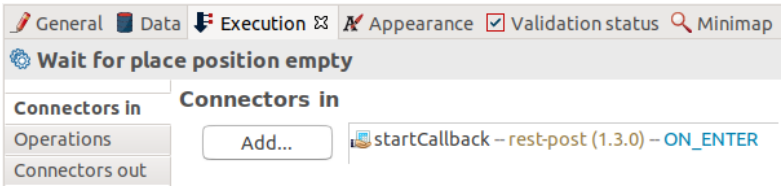
A frequently used type of activity is the *Service task*. These activities can be used to utilize different connectors, for example the REST connector. Using this type of connector, requests are sent to the *client service*, where they are processed and forwarded to the cobots via the OPC UA connection. The REST connector is added to an activity under the *Execution* tab in *Connectors in* (cf. Figure 7.20a). The connector offers a lot of settings, but the most important ones are the URL, the content type and payload (cf. Figure 7.20b). The data returned by the request is processed in Figure 7.20c, where it is stored in a business variable.



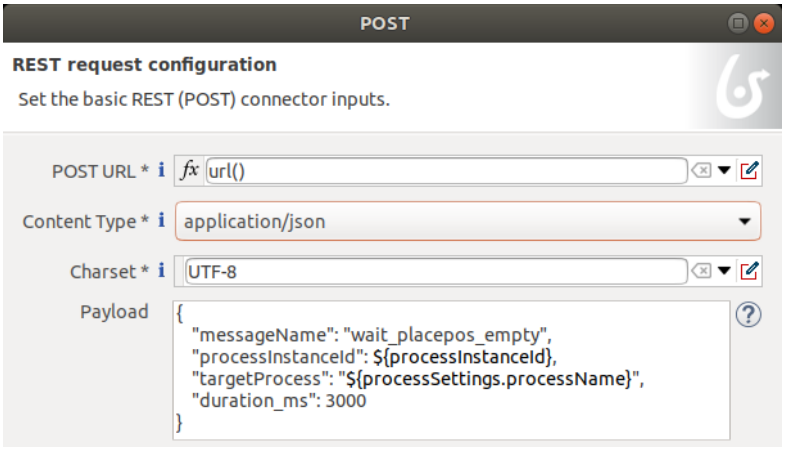
(a) *Home* position. (b) *Pick-up* position. (c) *Place* position.

Figure 7.19.: Example of cobot positions in the Pick & Place process (screenshots).

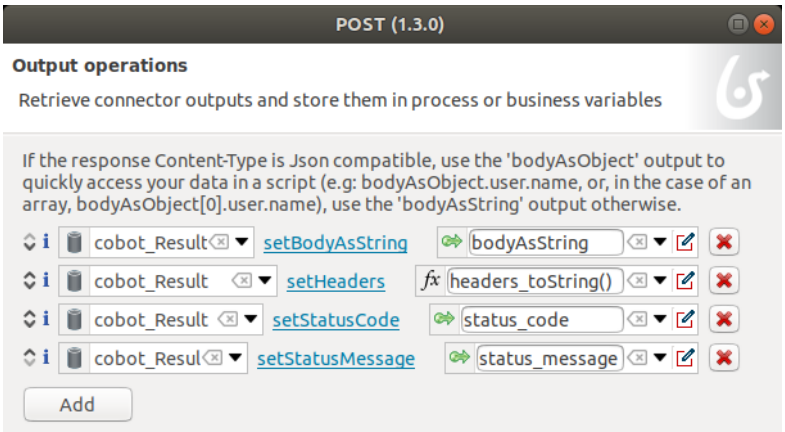
The *Bonita User Application* provides the performing actors an overview page where the current state of the process can be viewed (cf. Figure 7.21). It includes different aspects like the *Business Data* and their current values, and the timeline of the process instance. The timeline shows which activities have already been performed in the process instance, the timestamp when they were executed, and the actor who performed them.



(a) Adding a REST connector to an activity (screenshot).



(b) Configuring the request URL and body (screenshot).



(c) Processing the response of the request (screenshot).

Figure 7.20.: Using a REST connector in Bonita.

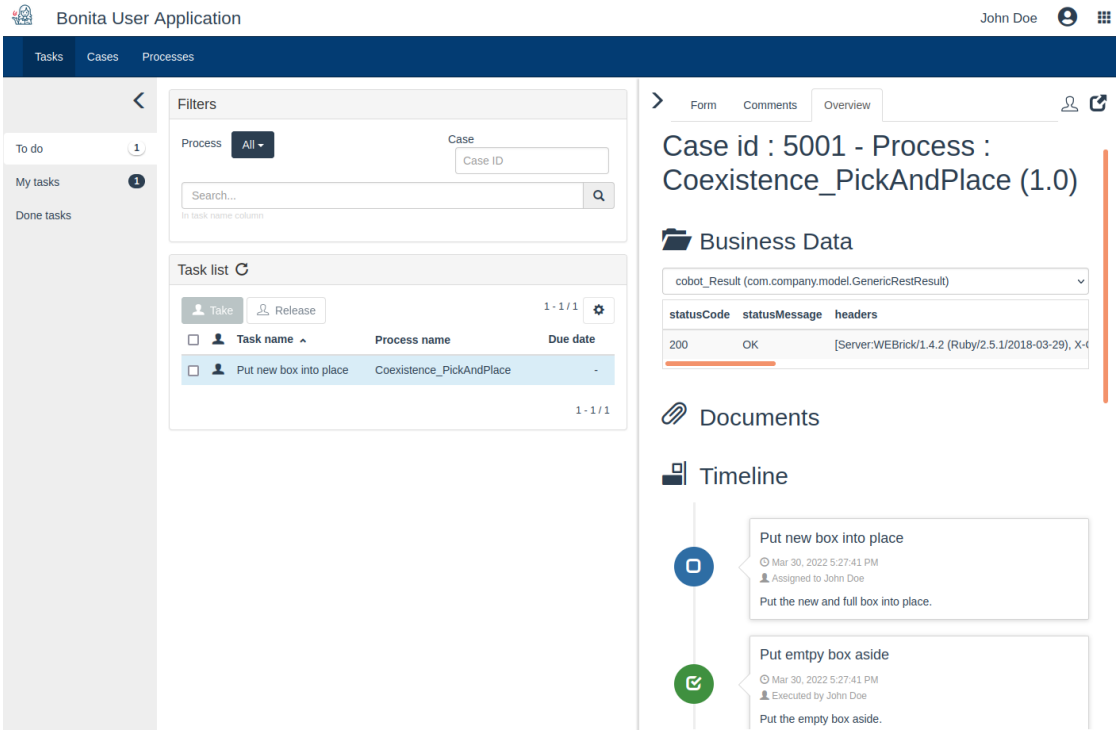


Figure 7.21.: Process instance overview page provided by the Bonita User Application (screenshot).

In the process, the human operator has different tasks to perform. Figure 7.22 shows the *Task List* for an operator in the Bonita User Application. On the right side, also a form of an activity can be seen, where the user needs to input process relevant data for example. From here, users can work on tasks that are available to them. The example in this picture shows the activity *Check part box*, where the user needs to determine the fill level of the box. After specifying the fill level in the input field, the activity can be finished by submitting the form.

In the Pick & Place use case, there are a few key positions of the cobot. Three of them are shown in Figure 7.19. The start and end orientation of the cobot is the *home* position, depicted in Fig. 7.19a. Figure 7.19b shows a *pick-up* position, which varies depending on the located part. The position in which to place the picked parts can be seen in Figure 7.19c. The whole cobot program used in this implementation can be seen in Figure A.3.

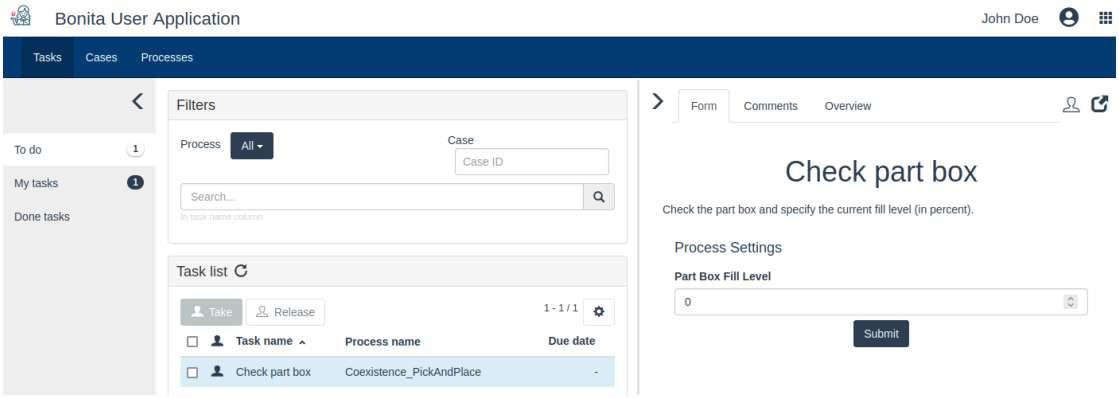
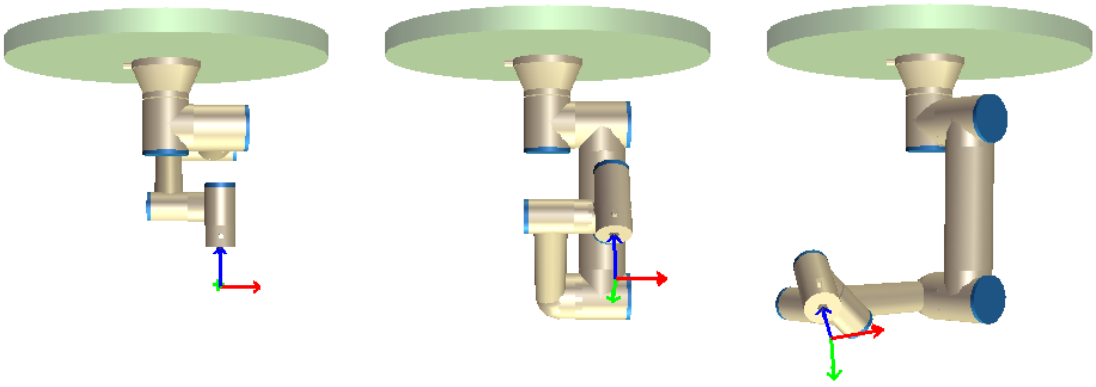


Figure 7.22.: Task overview and form page in Bonita User Application (screenshot).

7.1.3. Quality inspection

The quality inspection process implemented in the test environment refers to the use case in Section 3.7. The process model can be seen in Figure 7.25, and the cobot program in Figure A.2. One operator works alongside two cobots and their task is to check the quality of certain spots of a workpiece. There are fixed spots to check for each of the actors, and they are arranged in a way that none of the actors interferes with the space of another actor. This way, they can work concurrently on their tasks without the need to coordinate their steps with other actors. The quality inspection spots are specified in the process settings business variable as a list of positions. They have a name, a performing actor, a boolean variable indicating whether the spot should be checked manually or automatically, and in case of the automatic check it also includes the coordinates of the position. Figure 7.24 shows a slice of these position variables during process execution. The three actors work in parallel and determine the quality for each of the spots. It is possible that some automatic inspections can not deliver a reliable result. In that case, the result is set to “RECHECK”. After all three actors finished their tasks, all of the “RECHECK” spots are shown to the operator, who then makes a final decision on the quality result. This completes the quality inspection process for a workpiece.



(a) *Home* position. (b) Waiting position between inspections. (c) One of the inspection positions.

Figure 7.23.: Example of cobot positions in the Quality inspection process (screenshots).

Business Data

inspectionPosList (com.company.model.QualityInspectionPosition)

description	cobotTask	cobot	qualityResult
FrontLeftBottom	false		OK
FrontLeftTop	false		OK
FrontRightBottom	false		OK
FrontRightTop	false		OK
BackLeftBot_1	true	cobot1	RECHECK
BackLeftBot_2	true	cobot1	OK
BackLeftTop_1	true	cobot1	OK
BackLeftTop_2	true	cobot1	OK
BackRightBot_1	true	cobot2	OK
BackRightBot_2	true	cobot2	NOK
BackRightTop_1	true	cobot2	NOK
BackRightTop_2	true	cobot2	OK

Figure 7.24.: Overview of the quality inspection spots in the business variables during process execution (screenshot).

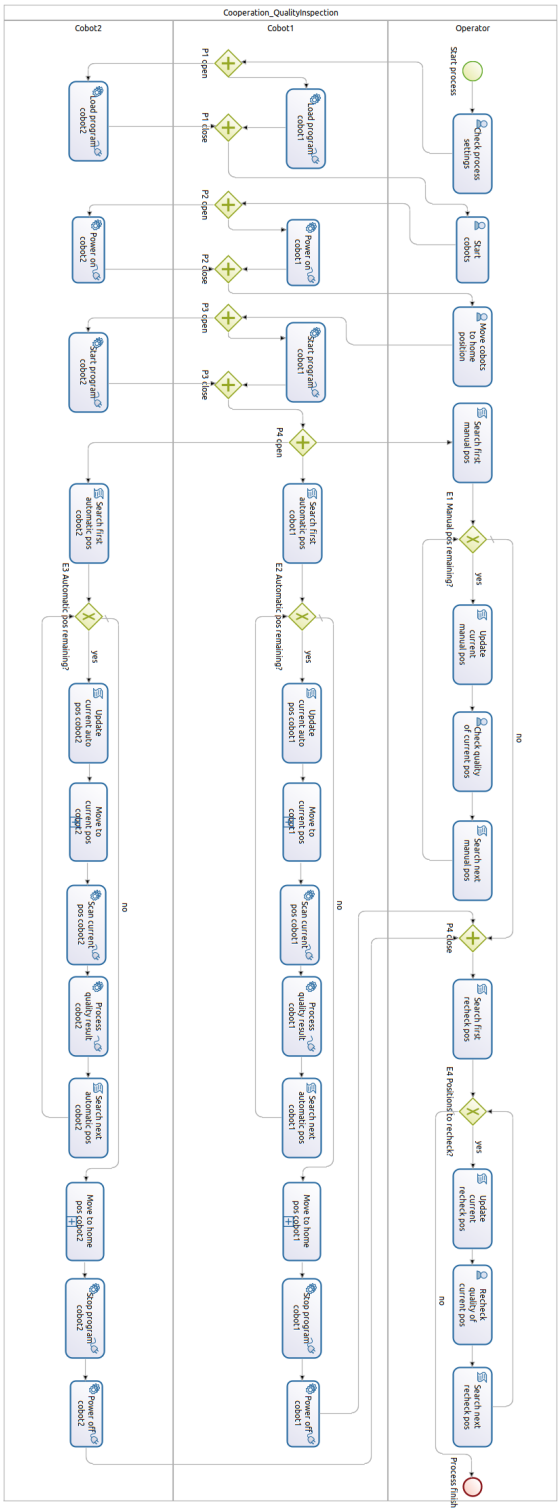
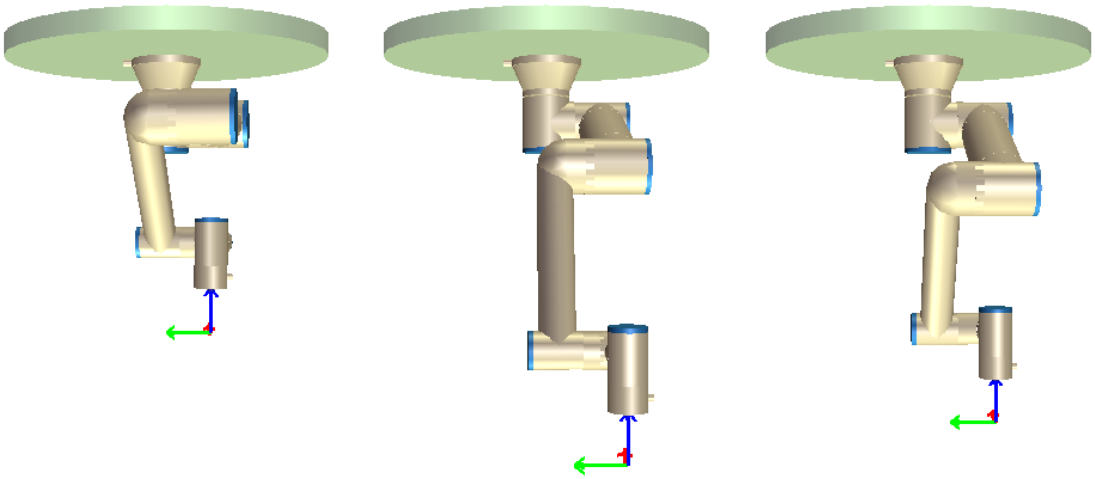


Figure 7.25.: Process model of the Quality inspection process.

7.1.4. Polishing

In this example of a polishing process, one operator and one cobot are finishing the surface of a large workpiece (cf. Fig. 7.27). The cobot, in this case, is top mounted, and performs the main polishing work. The polishing surface is split into several lines, each with start and end position coordinates (cf Figure 7.26). The cobot polishes line after line, and once it is finished with the last one, it moves into the predefined home position. The cobot program used to accomplish this is depicted in Figure A.1. The operator's task is to check the cobot's work and perform fine polishing if necessary. The tasks of the two actors are spatially constrained. The operator can check and fine polish all the lines that the cobot has already processed. This process is an implementation of a finishing task described in Section 3.6.



(a) *Home* position.

(b) Start position of one of the polishing lines.

(c) End position of one of the polishing lines.

Figure 7.26.: Example of cobot positions in the Polishing process (screenshots).

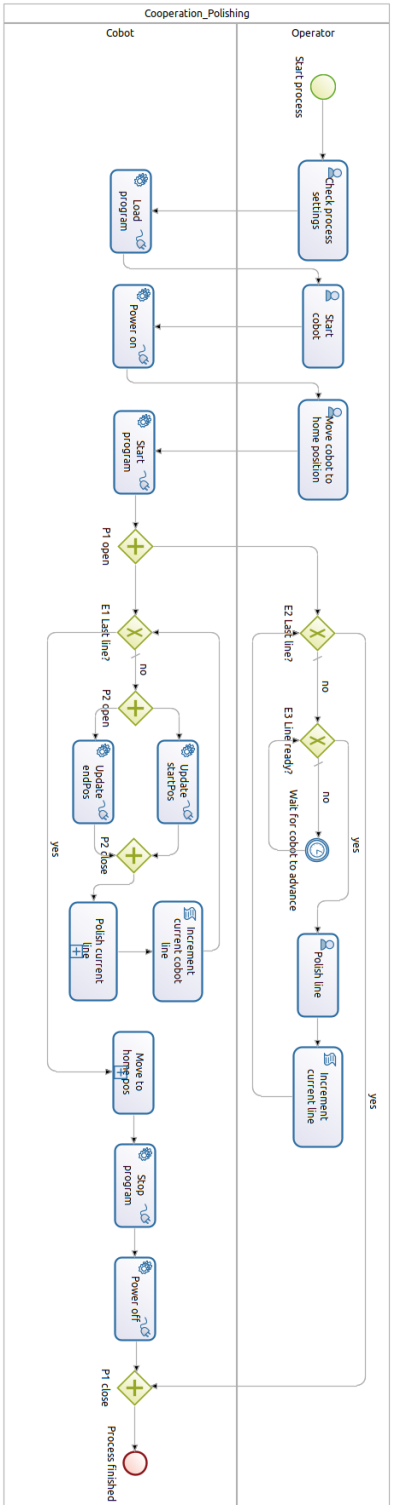
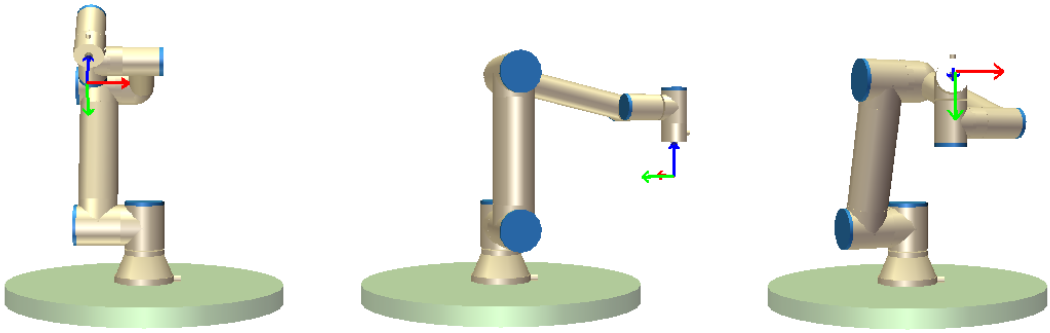


Figure 7.27.: Process model of the Polishing process.

7.1.5. Fixture

This process implements a use case described in Section 3.11. Figure 7.29 depicts the process model of this fixture use case. Some example cobot positions of the use case are shown in Figure 7.28, and the cobot program implementation is depicted in Figure A.6. The goal of this process is to mount several parts onto a larger workpiece. As these parts need to be attached from one side of the large workpiece, but mounted from the other side, two actors are needed to perform the process. Also, these parts are quite heavy, hence the cobot performs the task of holding the parts into place. The operator can then mount each part. The order in which the parts are installed does not matter, so the operator can arbitrarily choose which of the remaining positions is selected next. Once all parts are installed on the workpiece, the cobot moves to its home position and the process instance finishes.



(a) *Home* position. (b) *Pick-up* position. (c) One of the mounting positions.

Figure 7.28.: Example of cobot positions in the Fixture process (screenshots).

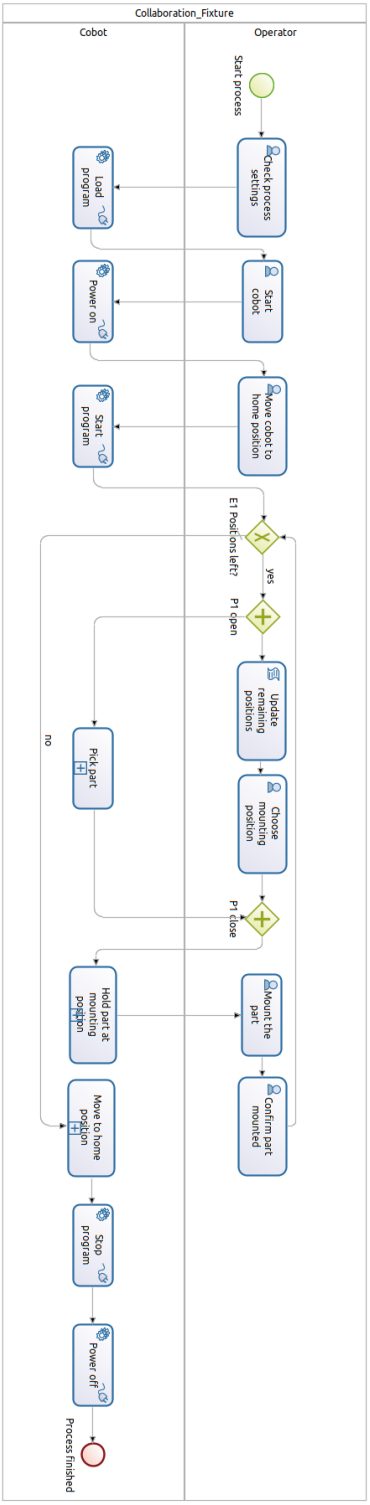
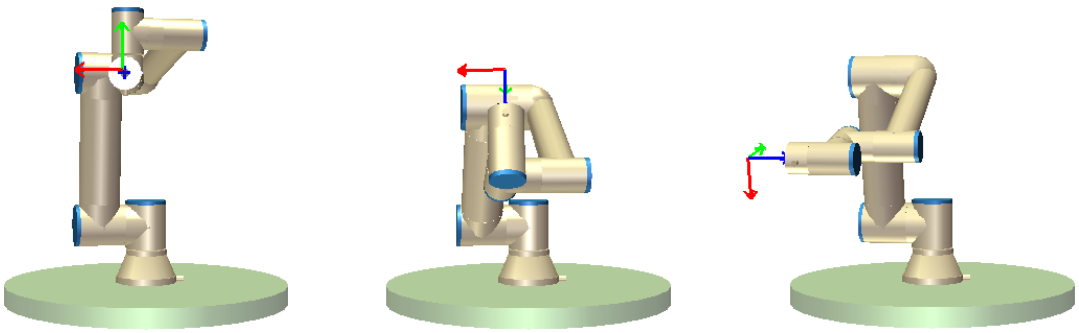


Figure 7.29.: Process model of the Fixture process.

7.1.6. Collaborative assembly

The collaborative assembly process described in Section 3.2 has been implemented and can be seen in Figure 7.32. It is based on the architecture developed in [29], although a simplified version of their process. Figure 7.30 depicts a few positions, which are required in the execution of this process. The process settings hold a list of tasks that have to be performed in a specific order. Based on this list, the cobot guides the operator through the assembly of a box, where some plates must be attached with four screws each. The cobot also checks the assembly of each installed plate for missing screws. If there is a problem with the assembled plate, the cobot tells the operator to check again and install the missing screws. To do this, the cobot uses a display, which acts as his face. Due to the simulated setup, a terminal window is used to mimic this. This window opens automatically, shows the message, and closes again after a few seconds. The cobot is also equipped with a gesture and face recognition system (simulated by randomized results). The gesture recognition system is used for the communication from the operator to the cobot. At the beginning of the process, the face recognition system checks if the current user is authorized to perform the assembly. The process only continues when the user is recognized as an authorized person. Otherwise, the cobot displays the message "User not authorized". The cobot program of this use case can be seen in Figures A.7 , A.8 and A.9.



(a) Handover position of the assembly plates.

(b) Assembly position for the bottom plate.

(c) Quality check position for the bottom plate.

Figure 7.30.: Example of cobot positions in the Collaborative assembly process (screenshots).

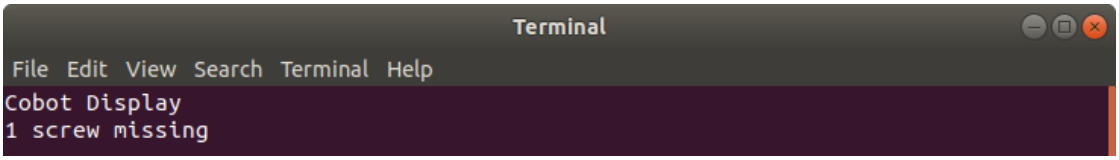


Figure 7.31.: Message from the cobot that the quality inspection result shows a missing screw.

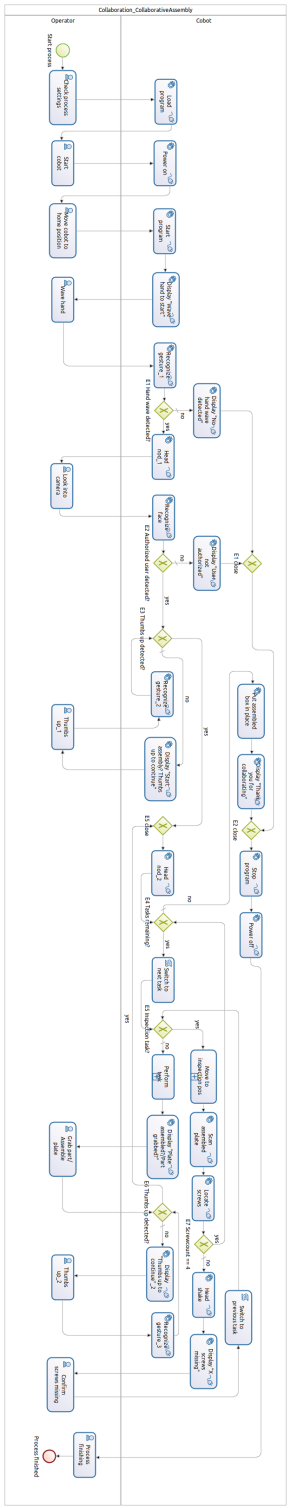


Figure 7.32.: Process model of the Collaborative assembly process.

In summary, Section 7.1 demonstrates that the chosen setup approach is suitable for the implementation of the use cases discussed in Chapter 3. Furthermore, the collaborative patterns (Synchronized, Coexistence, Cooperation, and Collaboration) are successfully demonstrated by implementations of use cases, with the use of Bonita, URSim, and OPC UA.

8. Conclusion

The aim of this research was to investigate collaboration patterns in the Human-Robot Collaboration domain, to explore potential definitions of these patterns, and finally to examine how they can be implemented in combination with Workflow Management Systems in order to support the flexibility needed in modern production environments. The main findings are summarized in the following.

Q1. This work started off with an analysis of literature on Human-Robot Collaboration and collaborative robots. Different types of collaboration scenarios were discussed. Functionalities and abilities of cobots, which enable the execution of different use cases, were identified, and potential types of communication for the interaction between humans and cobots were analyzed. Alongside, typical use cases from industry and research, which involve the use of cobots, were investigated and analyzed with regard to those aspects. This resulted in four collaboration patterns (Synchronized, Cooperation, Collaboration, Coexistence) and a collection of 14 different use cases that fit into those patterns.

Q2. The resulting collaborative patterns were then further analyzed, along with the proposal of a formal definition to provide a way to systematically describe these patterns. This definition also takes the possibility of multiple process actors into account. The definition was then used to outline general characteristics of the four patterns. Additionally, some examples from the use case analysis were instantiated using the proposed definition.

Q3. In order to create a test system, which enables the implementation of use cases with a simulated cobot in combination with a WfMS, the technical aspects of potential solutions were analyzed and evaluated. The results were then used to develop a test environment, which was then utilized to successfully demonstrate the implementation of the four identified collaborative patterns. The architecture of the test system is based on multiple Virtual Machines. It comprises one VM for the WfMS, Bonita in this case, and one distinct VM for each instance of cobot simulation (URSim). For the communication between them, the open-source standard Open Platform Communications Unified Architecture (OPC UA) is utilized.

Future work

Although the current approach for the communication between the WfMS and the cobot utilizes the industry standard Open Platform Communications Unified Architecture (OPC UA), it shows some limitations regarding the time needed for message transmission.

In time-critical situations, especially when the process involves the exchange of many messages, the current selection of tools introduces delays that may lead to unacceptable increases in cycle times, for example. Further research could explore different implementations of the OPC UA standard or also other ways of message transmission, to reduce transmission times.

Another point of interest is a visualization for the formal description of the collaboration patterns. During the exploration of different visualization approaches, it became apparent, that once a certain level of detail should be depicted, i.e., process instance level including multiple actors, a number of potential problems regarding the design of modeling notations arise. Since this was far beyond the scope of this research project, the focus was returned to the formal notation. Therefore, developing a visualization of the proposed formal definition would be a highly interesting topic for future research.

Bibliography

[1] T. Park and B. Lee, 'An approach to robot motion analysis and planning for conveyor tracking', *IEEE Transactions on Systems, Man, and Cybernetics*, vol. 22, no. 2, pp. 378–384, 1992. DOI: 10.1109/21.148413.

[2] S. Meilin, Y. Guangxin, X. Yong and W. Shangguang, 'Workflow management systems: A survey', in *ICCT'98. 1998 International Conference on Communication Technology. Proceedings (IEEE Cat. No.98EX243)*, vol. 2, 1998, 6–pp. DOI: 10.1109/ICCT.1998.740974.

[3] M. Reichert, 'Dynamische ablaufänderungen in workflow-management-systemen', Ph.D. dissertation, Uni Ulm, 2000. [Online]. Available: http://dbis.eprints.uni-ulm.de/433/.

[4] K. Salimifard and M. Wright, 'Petri net-based modelling of workflow systems: An overview', *European Journal of Operational Research*, vol. 134, no. 3, pp. 664–676, 2001, ISSN: 0377-2217. DOI: https://doi.org/10.1016/S0377-2217(00)00292-7. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0377221700002927.

[5] S. A. White, 'Introduction to BPMN', *Ibm Cooperation*, vol. 2, no. 0, 2004.

[6] P. Marshall and J. Rinaldi, *Industrial Ethernet: How to Plan, Install, and Maintain TCP/IP Ethernet Networks : the Basic Reference Guide for Automation and Process Control Engineers*. ISA, 2005, ISBN: 9781556178924. [Online]. Available: https://books.google.at/books?id=280HAAAACAAJ.

[7] A. BAUER, D. WOLLHERR and M. BUSS, 'Human–robot collaboration: A survey', *International Journal of Humanoid Robotics*, vol. 05, no. 01, pp. 47–66, 2008. DOI: 10.1142/S0219843608001303. eprint: https://doi.org/10.1142/S0219843608001303. [Online]. Available: https://doi.org/10.1142/S0219843608001303.

[8] J. Krüger, T. Lien and A. Verl, 'Cooperation of human and machines in assembly lines', *CIRP Annals*, vol. 58, no. 2, pp. 628–646, 2009, ISSN: 0007-8506. DOI: https://doi.org/10.1016/j.cirp.2009.09.009. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0007850609001760.

[9] C. Lenz, M. Rickert, G. Panin and A. Knoll, 'Constraint task-based control in industrial settings', in *2009 IEEE/RSJ International Conference on Intelligent Robots and Systems*, Oct. 2009, pp. 3058–3063. DOI: 10.1109/IROS.2009.5354631.

[10] S. Lichiardopol, N. van de Wouw and H. Nijmeijer, 'Control scheme for human-robot co-manipulation of uncertain, time-varying loads', in *2009 American Control Conference*, Jun. 2009, pp. 1485–1490. DOI: 10.1109/ACC.2009.5160062.

[11] Y. Chua, K. P. Tee and R. Yan, 'Human-robot motion synchronization using reactive and predictive controllers', in *2010 IEEE International Conference on Robotics and Biomimetics*, Dec. 2010, pp. 223–228. DOI: 10.1109/ROBIO.2010.5723331.

[12] C.-C. Lan, J.-H. Wang and Y.-H. Chen, 'A compliant constant-force mechanism for adaptive robot end-effector operations', in *2010 IEEE International Conference on Robotics and Automation*, May 2010, pp. 2131–2136. DOI: 10.1109/ROBOT.2010.5509928.

[13] W. Aalst, van der, *Process mining : discovery, conformance and enhancement of business processes*, English. Germany: Springer, 2011, ISBN: 978-3-642-19344-6. DOI: 10.1007/978-3-642-19345-3.

[14] M. Chinosi and A. Trombetta, 'BPMN: An introduction to the standard', *Computer Standards & Interfaces*, vol. 34, no. 1, pp. 124–134, 2012, ISSN: 0920-5489. DOI: https://doi.org/10.1016/j.csi.2011.06.002. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0920548911000766.

[15] Z. Pan, J. Polden, N. Larkin, S. Van Duin and J. Norrish, 'Recent progress on programming methods for industrial robots', *Robotics and Computer-Integrated Manufacturing*, vol. 28, no. 2, pp. 87–94, 2012, ISSN: 0736-5845. DOI: https://doi.org/10.1016/j.rcim.2011.08.004. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0736584511001001.

[16] KUKA Systems GmbH, *Industrial robots welding car axles in an automotive factory*, [Image], Licensed under CC BY-SA 3.0 https://creativecommons.org/licenses/by-sa/3.0/deed.en, 29th Jul. 2013. [Online]. Available: https://commons.wikimedia.org/wiki/File:Application_field_automotive.jpg (visited on 26th Nov. 2021).

[17] M. H. Schwarz and J. Börcsök, 'A survey on opc and opc-ua: About the standard, developments and investigations', in *2013 XXIV International Conference on Information, Communication and Automation Technologies (ICAT)*, 2013, pp. 1–6. DOI: 10.1109/ICAT.2013.6684065.

[18] D. Karagiannis, 'Workflow-management-system', *Enzyklopädie der Wirtschaftsinformatik. Online-Lexikon. Oldenbourg, München*, 2014. [Online]. Available: https://www.enzyklopaedie-der-wirtschaftsinformatik.de/wi-enzyklopaedie/lexikon/is-management/Systementwicklung/Softwarearchitektur/Middleware/Workflow-Management-System/ (visited on 29th Nov. 2021).

[19] L. Monostori, 'Cyber-physical production systems: Roots, expectations and r&d challenges', *Procedia CIRP*, vol. 17, pp. 9–13, 2014, Variety Management in Manufacturing, ISSN: 2212-8271. DOI: https://doi.org/10.1016/j.procir.2014.03.115. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S2212827114003497.

[20] B. Chandrasekaran and J. M. Conrad, 'Human-robot collaboration: A survey', in *SoutheastCon 2015*, 2015, pp. 1–8. DOI: 10.1109/SECON.2015.7132964.

[21] G. Charalambous, S. Fletcher and P. Webb, 'Identifying the key organisational human factors for introducing human-robot collaboration in industry: An exploratory study', *The International Journal of Advanced Manufacturing Technology*, vol. 81, no. 9, pp. 2143–2155, 2015. [Online]. Available: https://doi.org/10.1007/s00170-015-7335-4.

[22] G. Dumonteil, G. Manfredi, M. Devy, A. Confetti and D. Sidobre, 'Reactive planning on a collaborative robot for industrial applications', in *2015 12th International Conference on Informatics in Control, Automation and Robotics (ICINCO)*, vol. 02, 2015, pp. 450–457.

[23] A. Hermann, F. Mauch, K. Fischnaller, S. Klemm, A. Roennau and R. Dillmann, 'Anticipate your surroundings: Predictive collision detection between dynamic obstacles and planned robot trajectories on the gpu', in *2015 European Conference on Mobile Robots (ECMR)*, 2015, pp. 1–8. DOI: 10.1109/ECMR.2015.7324047.

[24] C.-M. Huang, M. Cakmak and B. Mutlu, 'Adaptive coordination strategies for human-robot handovers.', in *Robotics: science and systems*, Rome, Italy, vol. 11, 2015.

[25] S.-D. Lee, M.-C. Kim and J.-B. Song, 'Sensorless collision detection for safe human-robot collaboration', in *2015 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, 2015, pp. 2392–2397. DOI: 10.1109/IROS.2015.7353701.

[26] R. Müller, M. Vette and O. Mailahn, 'Process-oriented task assignment for assembly processes with human-robot interaction', *Procedia CIRP*, vol. 44, pp. 210–215, 2016, 6th CIRP Conference on Assembly Technologies and Systems (CATS), ISSN: 2212-8271. DOI: https://doi.org/10.1016/j.procir.2016.02.080. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S2212827116003620.

[27] X. Xie and L. Sun, 'Force control based robotic grinding system and application', in *2016 12th World Congress on Intelligent Control and Automation (WCICA)*, Jun. 2016, pp. 2552–2555. DOI: 10.1109/WCICA.2016.7578828.

[28] Aurora Flight Sciences. (16th May 2017). 'Robotic co-pilot autonomously flies and lands a simulated boeing 737', [Online]. Available: https://www.aurora.aero/wp-content/uploads/2015/09/APR-343_ALIAS-737-Simulator-1.pdf (visited on 6th May 2021).

[29] I. El Makrini, K. Merckaert, D. Lefeber and B. Vanderborght, 'Design of a collaborative architecture for human-robot assembly tasks', in *2017 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, Sep. 2017, pp. 1624–1629. DOI: 10.1109/IROS.2017.8205971.

[30] I. El Makrini, S. Elprama, J. Van den Bergh, D. Lefeber, B. Vanderborght, C. Jewell and A. Jacobs, 'Design and investigation of social robotic coworkers in factories', Sep. 2017. [Online]. Available: https://www.researchgate.net/publication/329782314_Design_and_Investigation_of_Social_Robotic_Coworkers_in_Factories.

[31] R. Müller, M. Vette and A. Geenen, 'Skill-based dynamic task allocation in human-robot-cooperation with the example of welding application', *Procedia Manufacturing*, vol. 11, pp. 13–21, 2017, 27th International Conference on Flexible Automation and Intelligent Manufacturing, FAIM2017, 27-30 June 2017, Modena, Italy, ISSN: 2351-9789. DOI: https://doi.org/10.1016/j.promfg.2017.07.113. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S2351978917303177.

[32] S. Robla-Gómez, V. M. Becerra, J. R. Llata, E. González-Sarabia, C. Torre-Ferrero and J. Pérez-Oria, 'Working together: A review on safe human-robot collaboration in industrial environments', *IEEE Access*, vol. 5, pp. 26 754–26 773, 2017. DOI: 10.1109/ACCESS.2017.2773127.

[33] Tlach, Vladimir, Cisar, Miroslav, Kuric, Ivan and Zajacko, Ivan, 'Determination of the industrial robot positioning performance', *MATEC Web Conf.*, vol. 137, 2017. DOI: 10.1051/matecconf/201713701004. [Online]. Available: https://doi.org/10.1051/matecconf/201713701004.

[34] B. Vanderborght, J. Berte, G. Coppel, I. El Makrini, S. Elprama, C. Jewell, A.-J. Knevels, J. Potargent, I. Ravyse, K. Schroyen, F. Stals, J. Van den Bergh, T. Van der Auwermeulen, T. Waegeman and A. Jacobs, 'Towards an acceptable socially collaborative robot for the manufacturing industry', Jul. 2017. [Online]. Available: https://www.researchgate.net/publication/329782294_Towards_an_Acceptable_Socially_Collaborative_Robot_for_the_Manufacturing_Industry.

[35] X. V. Wang, Z. Kemény, J. Váncza and L. Wang, 'Human–robot collaborative assembly in cyber-physical production: Classification framework and implementation', *CIRP Annals*, vol. 66, no. 1, pp. 5–8, 2017, ISSN: 0007-8506. DOI: https://doi.org/10.1016/j.cirp.2017.04.101. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0007850617301014.

[36] X. Zhang, H. Chen, N. Yang, H. Lin and K. He, 'A structure and control design of constant force polishing end actuator based on polishing robot', in *2017 IEEE International Conference on Information and Automation (ICIA)*, Jul. 2017, pp. 764–768. DOI: 10.1109/ICInfA.2017.8079007.

[37] I. El Makrini, S. A. Elprama, J. Van den Bergh, B. Vanderborght, A. Knevels, C. I. C. Jewell, F. Stals, G. De Coppel, I. Ravyse, J. Potargent, J. Berte, B. Diericx, T. Waegeman and A. Jacobs, 'Working with walt: How a cobot was developed and inserted on an auto assembly line', *IEEE Robotics Automation Magazine*, vol. 25, no. 2, pp. 51–58, 2018. DOI: 10.1109/MRA.2018.2815947. [Online]. Available: https://ieeexplore.ieee.org/document/8360084.

[38] R. Lackes and M. Siepermann. (19th Feb. 2018). 'Workflow management system - ausführliche definition im online-lexikon', [Online]. Available: https://wirtschaftslexikon.gabler.de/definition/workflow-management-system-50957/version-274165 (visited on 29th Nov. 2021).

[39] F. Pauker, J. Mangler, S. Rinderle-Ma and C. Pollak, 'Centurio.work - modular secure manufacturing orchestration', in *16th International Conference on Business Process Management 2018*, ser. Dissertation Award, Demonstration, and Industrial Track, BPM 2018, Sep. 2018, pp. 164–171. [Online]. Available: http://eprints.cs.univie.ac.at/5770/.

[40] V. Villani, F. Pini, F. Leali and C. Secchi, 'Survey on human–robot collaboration in industrial settings: Safety, intuitive interfaces and applications', *Mechatronics*, vol. 55, pp. 248–266, 2018, ISSN: 0957-4158. DOI: https://doi.org/10.1016/j.mechatronics.2018.02.009. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0957415818300321.

[41] S. El Zaatari, M. Marei, W. Li and Z. Usman, 'Cobot programming for collaborative industrial tasks: An overview', *Robotics and Autonomous Systems*, vol. 116, pp. 162–180, 2019, ISSN: 0921-8890. DOI: https://doi.org/10.1016/j.robot.2019.03.003. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S092188901830602X.

[42] J. Mangler, F. Pauker, S. Rinderle-Ma and M. Ehrendorfer, 'Centurio.work - industry 4.0 integration assessment and evolution', in *17th Int'l Conference on Business Process Management*, 2019, pp. 106–117. [Online]. Available: http://eprints.cs.univie.ac.at/6099/.

[43] E. Matheson, R. Minto, E. G. G. Zampieri, M. Faccio and G. Rosati, 'Human–robot collaboration in manufacturing applications: A review', *Robotics*, vol. 8, no. 4, 2019, ISSN: 2218-6581. DOI: 10.3390/robotics8040100. [Online]. Available: https://www.mdpi.com/2218-6581/8/4/100.

[44] A. Owen-Hill. (Jun. 2019). 'What's the difference between offline programming and simulation?', [Online]. Available: https://robodk.com/blog/difference-simulation-offline-programming/ (visited on 15th Nov. 2021).

[45] Z. Samadikhoshkho, K. Zareinia and F. Janabi-Sharifi, 'A brief review on robotic grippers classifications', in *2019 IEEE Canadian Conference of Electrical and Computer Engineering (CCECE)*, 2019, pp. 1–4. DOI: 10.1109/CCECE.2019.8861780.

[46] The Robot Report Staff. (14th Jan. 2019). 'Scoliobot aims to improve spinal surgery accuracy', [Online]. Available: https://www.therobotreport.com/scoliobot-spinal-surgery-accuracy/ (visited on 13th Apr. 2021).

[47] L. Wang, R. Gao, J. Váncza, J. Krüger, X. Wang, S. Makris and G. Chryssolouris, 'Symbiotic human-robot collaborative assembly', *CIRP Annals*, vol. 68, no. 2, pp. 701–726, 2019, ISSN: 0007-8506. DOI: https://doi.org/10.1016/j.cirp.2019.05.002. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0007850619301593.

[48] M. Adams, *Yawl - user manual - version 4.3*, The YAWL Foundation, Feb. 2020. [Online]. Available: https://yawlfoundation.github.io/assets/files/YAWLUserManual4.3.pdf (visited on 8th Feb. 2022).

[49] Y. Jiang, L. Yu, H. Jia, H. Zhao and H. Xia, 'Absolute positioning accuracy improvement in an industrial robot', *Sensors*, vol. 20, no. 16, 2020, ISSN: 1424-8220. DOI: 10.3390/s20164354. [Online]. Available: https://www.mdpi.com/1424-8220/20/16/4354.

[50] T. Kopp, M. Baumgartner and S. Kinkel, 'Success factors for introducing industrial human-robot interaction in practice: An empirically driven framework', *The International Journal of Advanced Manufacturing Technology*, pp. 1–20, 2020. [Online]. Available: https://link.springer.com/article/10.1007/s00170-020-06398-0.

[51] M. Shen, Y. Wang, Y. Jiang, H. Ji, B. Wang and Z. Huang, 'A new positioning method based on multiple ultrasonic sensors for autonomous mobile robot', *Sensors*, vol. 20, no. 1, 2020, ISSN: 1424-8220. DOI: 10.3390/s20010017. [Online]. Available: https://www.mdpi.com/1424-8220/20/1/17.

[52] J. H. Teo, A. Loganathan, P. Goh and N. S. Ahmad, 'Autonomous mobile robot navigation via rfid signal strength sensing', *International Journal of Mechanical Engineering and Robotics Research*, vol. 9, no. 8, pp. 1140–1144, 2020.

[53] Z. Bi, C. Luo, Z. Miao, B. Zhang, W. Zhang and L. Wang, 'Safety assurance mechanisms of collaborative robotic systems in manufacturing', *Robotics and Computer-Integrated Manufacturing*, vol. 67, p. 102 022, 2021, ISSN: 0736-5845. DOI: https://doi.org/10.1016/j.rcim.2020.102022. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0736584520302337.

[54] A. Buerkle, W. Eaton, N. Lohse, T. Bamber and P. Ferreira, 'Eeg based arm movement intention recognition towards enhanced safety in symbiotic human-robot collaboration', *Robotics and Computer-Integrated Manufacturing*, vol. 70, p. 102 137, 2021, ISSN: 0736-5845. DOI: https://doi.org/10.1016/j.rcim.2021.102137. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0736584521000223.

[55] S. Crowe. (1st Mar. 2021). 'How actinav automated a challenging bin-picking task', [Online]. Available: https://www.cobottrends.com/actinav-automated-a-challenging-bin-picking-task/ (visited on 12th Apr. 2021).

[56] P. Danilyuk, *Black and silver telescope with stand*, [Image], 28th May 2021. [Online]. Available: https://www.pexels.com/photo/black-and-silver-telescope-with-stand-8438874/ (visited on 26th Nov. 2021).

[57] A. A. Malik and A. Brem, 'Digital twins for collaborative robots: A case study in human-robot interaction', *Robotics and Computer-Integrated Manufacturing*, vol. 68, p. 102 092, 2021, ISSN: 0736-5845. DOI: https://doi.org/10.1016/j.rcim.2020.102092. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0736584520303021.

[58] V. Román Ibáñez, F. Pujol, S. García Ortega and J. Sanz Perpiñán, 'Collaborative robotics in wire harnesses spot taping process', *Computers in Industry*, vol. 125, p. 103 370, 2021, ISSN: 0166-3615. DOI: https://doi.org/10.1016/j.compind.2020.103370. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0166361520306047.

[59] The jBPM Team, *Jbpm documentation - version 7.64.0.final*, Red Hat, Inc., Jan. 2022. [Online]. Available: https://docs.jbpm.org/7.64.0.Final/jbpm-docs/html_single/ (visited on 8th Feb. 2022).

[60] ABB. (). 'Robotstudio, the world's most used offline programming tool for robotics', [Online]. Available: https://new.abb.com/products/robotics/robotstudio (visited on 16th Nov. 2021).

[61] AristaFlow GmbH, *Aristaflow bpm documentation*, AristaFlow GmbH. [Online]. Available: https://docs.aristaflow.com/Latest/ (visited on 10th Feb. 2022).

[62] Bizagi Corp., *Bizagi user guide*, Bizagi Corp. [Online]. Available: https://help.bizagi.com/bpm-suite/en/ (visited on 10th Feb. 2022).

[63] Bonitasoft, S.A., *Bonita documentation*, Bonitasoft, S.A. [Online]. Available: https://documentation.bonitasoft.com/bonita/2021.2/ (visited on 10th Feb. 2022).

[64] camunda Services GmbH, *The camunda platform manual*, camunda Services GmbH. [Online]. Available: https://docs.camunda.org/manual/7.16/ (visited on 8th Feb. 2022).

[65] FANUC. (). 'Intelligent 3-d robot simulation', [Online]. Available: https://www.fanuc.eu/at/en/robots/accessories/roboguide (visited on 16th Nov. 2021).

[66] KUKA. (). 'Kuka.sim', [Online]. Available: https://www.kuka.com/en-at/products/robotics-systems/software/simulation-planning-optimization/kuka_sim (visited on 16th Nov. 2021).

[67] Mobile Industrial Robots A/S, *Quick facts about the mir600*. [Online]. Available: https://www.mobile-industrial-robots.com/solutions/robots/mir600/ (visited on 26th Nov. 2021).

[68] ——, *Tb spain injection achieves significant gains in production capacity with two mir robots*, https://www.mobile-industrial-robots.com/de/insights/case-studies/tb-spain-injection-achieves-significant-gains-in-production-capacity-with-two-mir-robots/, [Image]. [Online]. Available: https://www.mobile-industrial-robots.com/media/1815104/es_tbsi_vigo_mir-201014100548.jpg (visited on 6th May 2021).

[69] OPC Foundation. (). 'Classic', [Online]. Available: https://opcfoundation.org/about/opc-technologies/opc-classic/ (visited on 23rd Nov. 2021).

[70] ——, (). 'The Industrial Interoperability Standard', [Online]. Available: https://opcfoundation.org/ (visited on 23rd Nov. 2021).

[71] ——, (). 'Unified architecture', [Online]. Available: https://opcfoundation.org/about/opc-technologies/opc-ua/ (visited on 23rd Nov. 2021).

[72] ——, (). 'What is opc?', [Online]. Available: https://opcfoundation.org/about/what-is-opc/ (visited on 23rd Nov. 2021).

[73] Universal Robots A/S. (). 'Introducing actinav', [Online]. Available: https://www.universal-robots.com/products/actinav/ (visited on 12th Apr. 2021).

[74] ——, (). 'Offline simulator - e-series - ur sim for linux 5.11.5', [Online]. Available: https://www.universal-robots.com/download/software-e-series/simulator-linux/offline-simulator-e-series-ur-sim-for-linux-5115/ (visited on 16th Nov. 2021).

[75] ——, 'White paper: An introduction to common collaborative robot applications', Tech. Rep. [Online]. Available: https://info.universal-robots.com/hubfs/Enablers/White%20papers/Common%20Cobot%20Applications.pdf (visited on 15th Nov. 2021).

[76] University of Ulm - Institute of Databases and Information Systems (DBIS). (). 'Aristaflow - description', [Online]. Available: https://www.uni-ulm.de/in/iui-dbis/forschung/abgeschlossene-projekte/aristaflow/ (visited on 10th Feb. 2022).

[77] YASKAWA. (). 'Motosim eg-vrc', [Online]. Available: https://www.yaskawa.de/produkte/software/productdetail/product/motosim-eg-vrc_1686 (visited on 16th Nov. 2021).

Acronyms

AMR Autonomous mobile robot. 16

API Application Programming Interface. 46–49, 62, 63, 67–69

BPMN Business Process Model and Notation. iii, v, 46–51, 75, 78, 82, 84

CPPS Cyber-Physical Production System. 4, 49

CPS Cyber-Physical System. 4, 49

HMI Human Machine Interface. 41

HRC Human-Robot Collaboration. iii, 1–4, 12, 18, 22, 105

IE Industrial Ethernet. 37

OLP Robotic Offline Programming. 43

OPC UA Open Platform Communications Unified Architecture. iii, v, xiii, 39–45, 53, 54, 56, 58–60, 62, 63, 65, 66, 73, 75, 92, 104–106

PFL power and force limiting. 7, 9

PLC Programmable Logic Controller. xiii, 3, 37, 38, 40, 41, 49

REST Representational State Transfer. xiv, 46–48, 62–64, 67, 68, 70, 75, 79, 84, 92, 93

SCADA Supervisory Control and Data Acquisition. 41

SOAP Simple Object Access Protocol. 47, 48

SRMS safety-rated monitored stop. 7, 9

SSM speed and separation monitoring. 7, 9

VM Virtual Machine. xiii, 40, 41, 45, 53–58, 60, 62, 73, 105

WfMS Workflow Management System. iii, v, xi, xiii, 4, 37–40, 43, 45–47, 49, 53, 54, 58, 62, 63, 67, 68, 70, 72, 75, 105

YAWL Yet Another Workflow Language. 48

A. Appendix

A.1. Implemented cobot programs

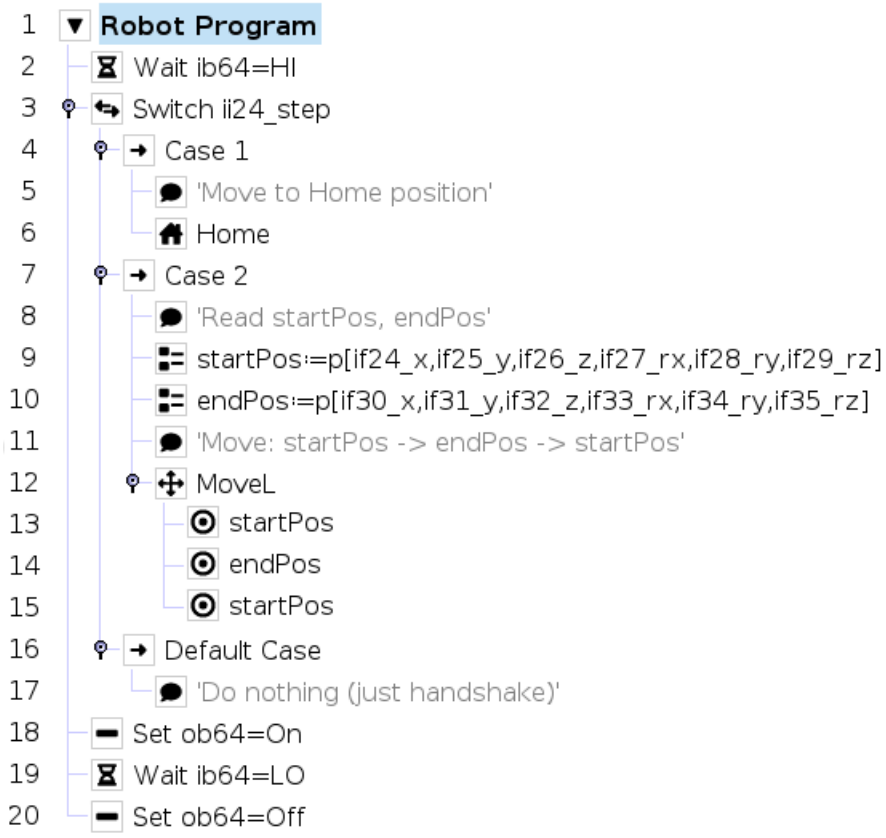


Figure A.1.: Cobot program used in the implementation of the Polishing process (screenshot).

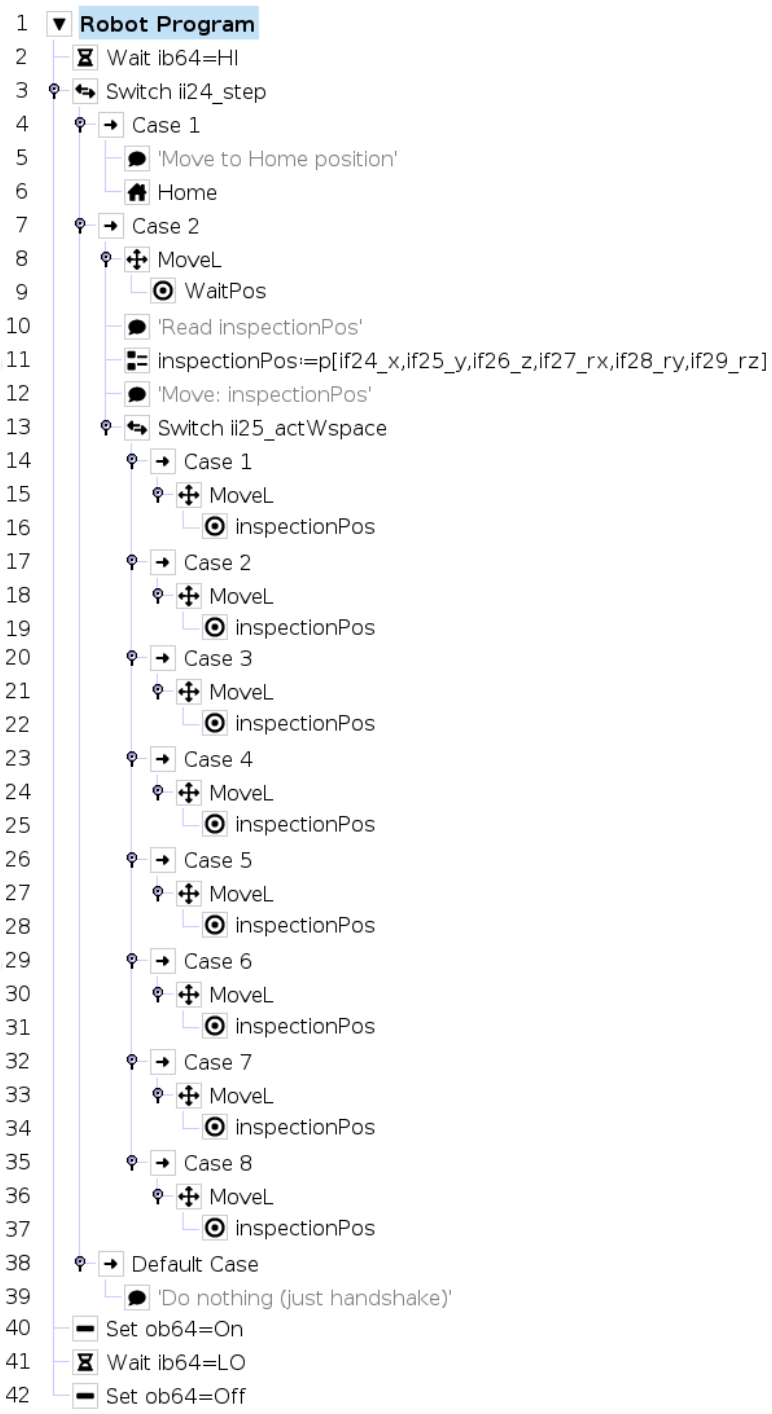


Figure A.2.: Cobot program used in the implementation of the Quality inspection process (screenshot).

Figure A.3.: Cobot program used in the implementation of the Pick & Place process (screenshot).

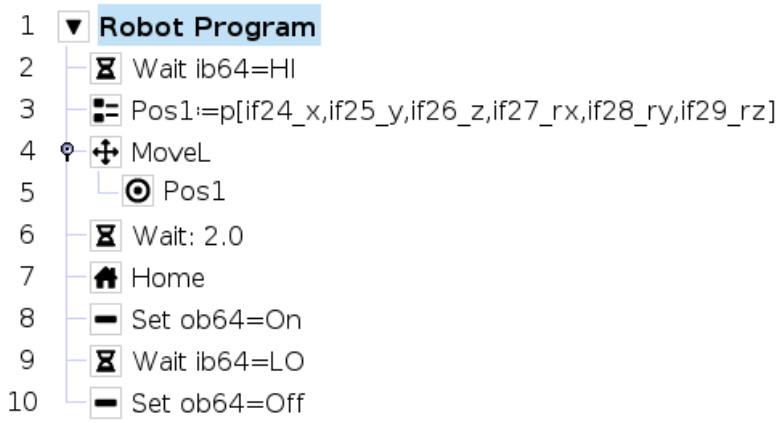


Figure A.4.: Cobot program used in the implementation of the Spot taping process (screenshot).

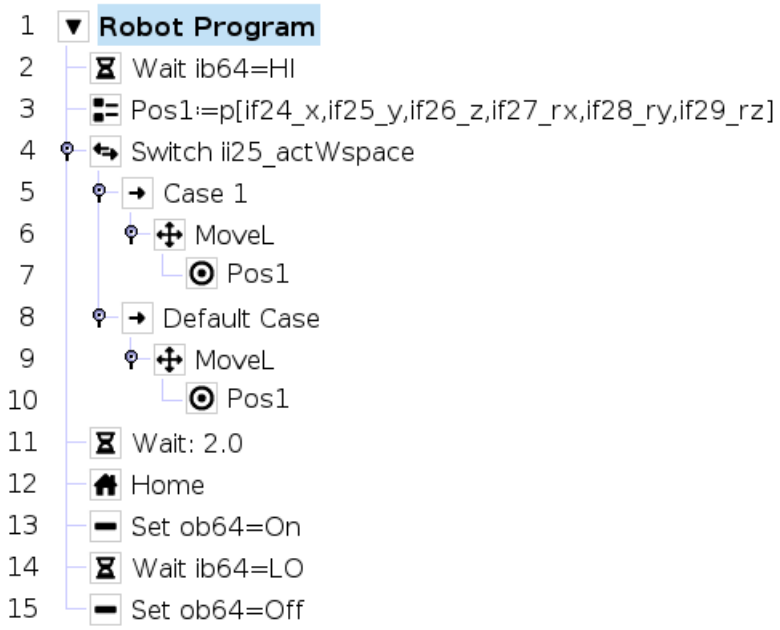


Figure A.5.: Cobot program used in the implementation of the Extended spot taping process (screenshot).

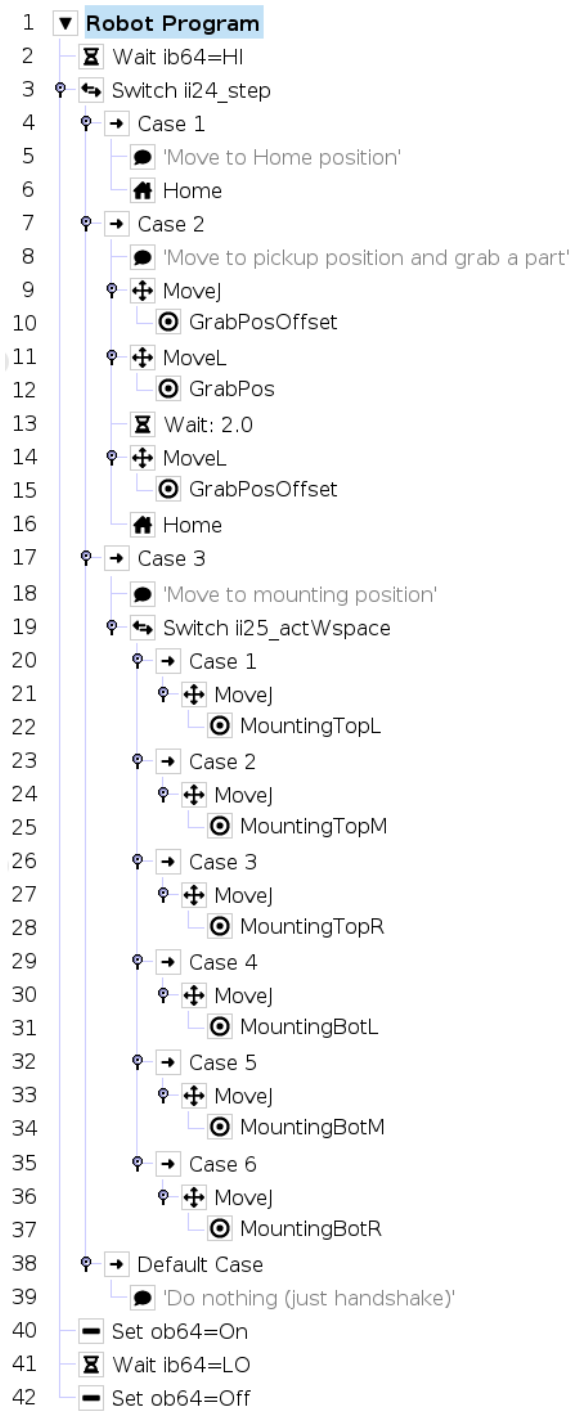


Figure A.6.: Cobot program used in the implementation of the Fixture process (screenshot).

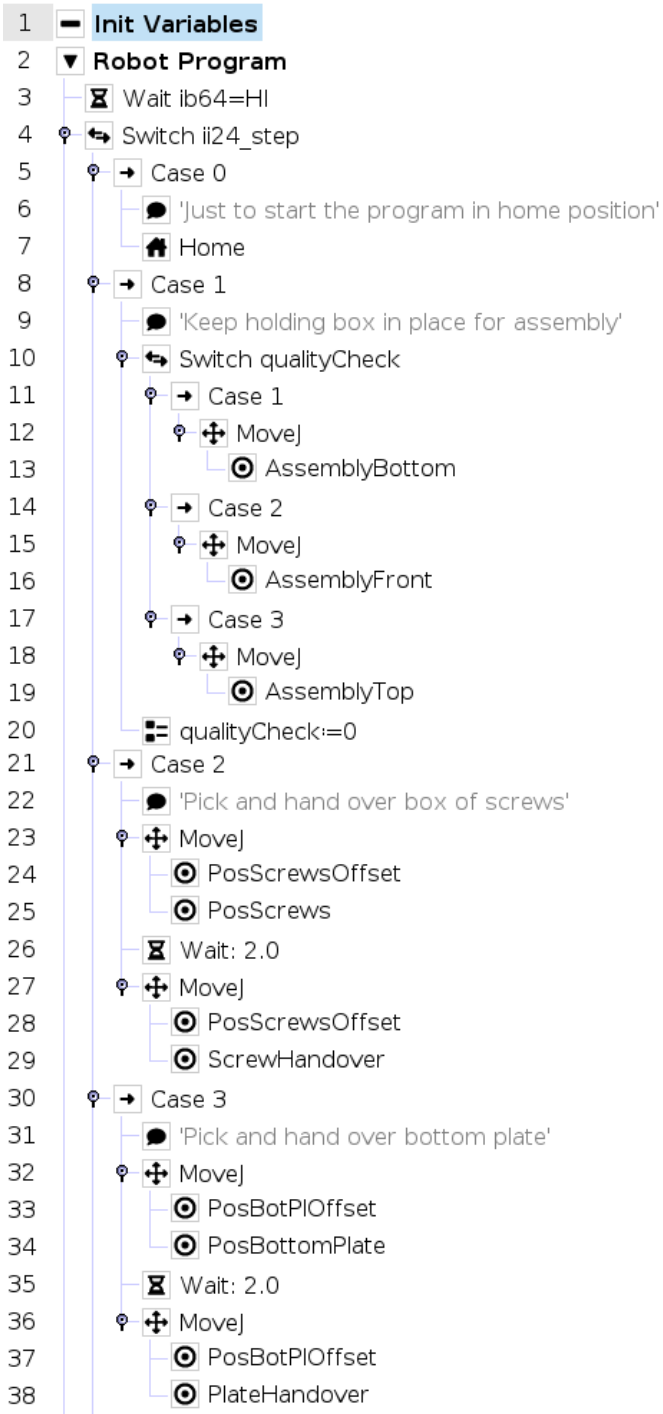


Figure A.7.: Cobot program used in the implementation of the Collaborative assembly process (screenshot part 1). *Init Variables* initializes *qualityCheck* to 0.

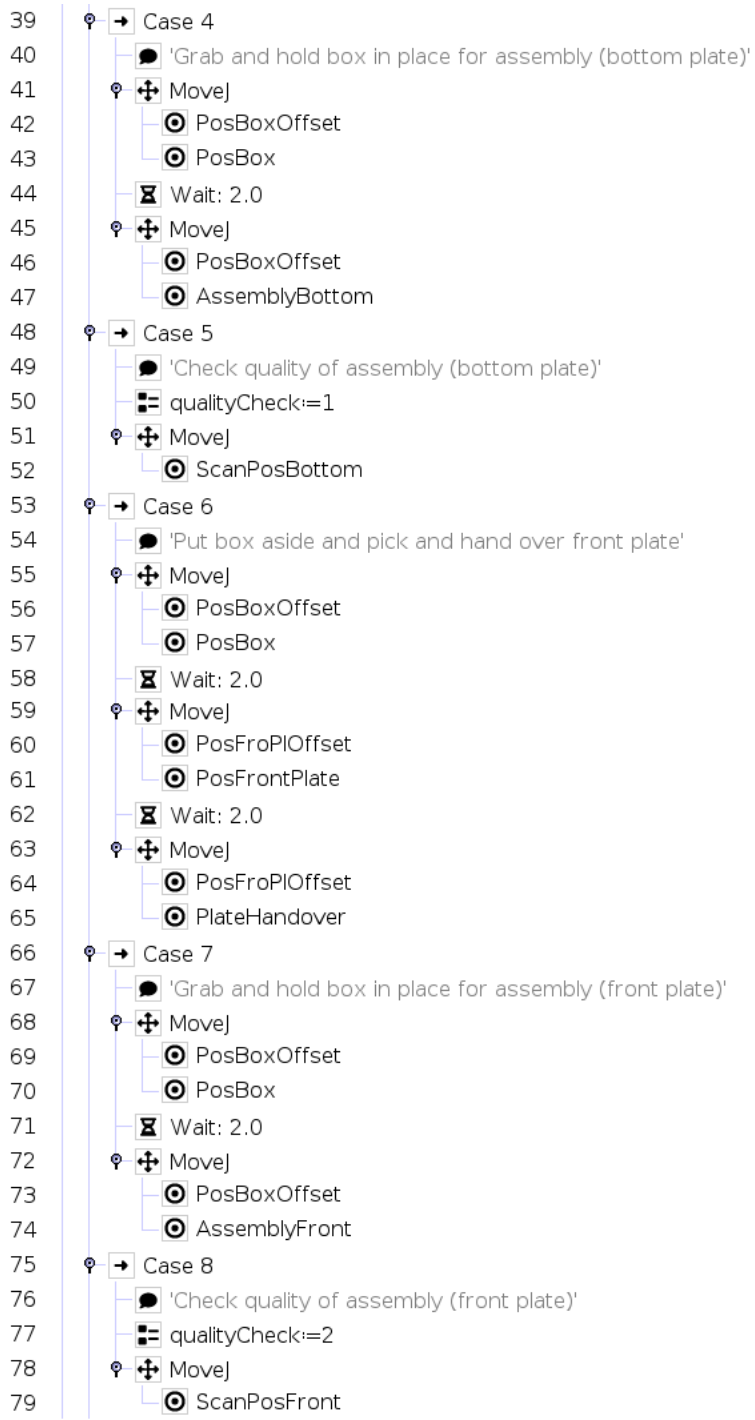


Figure A.8.: Cobot program used in the implementation of the Collaborative assembly process (screenshot part 2).

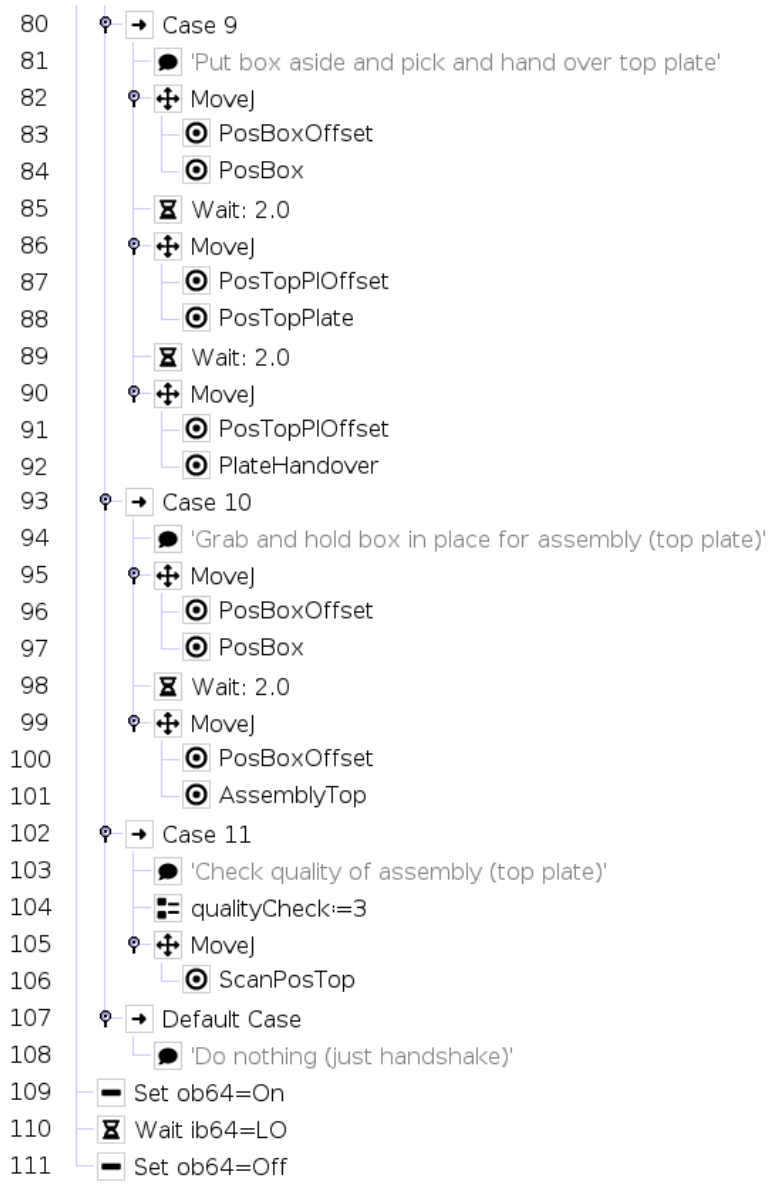


Figure A.9.: Cobot program used in the implementation of the Collaborative assembly process (screenshot part 3).