universität wien

MASTERARBEIT | MASTER'S THESIS

Titel | Title

Mining Cobot Logs: Identifying Collaborative Patterns in Manufacturing

verfasst von | submitted by

Stela Kucek BSc

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

Master of Science (MSc)

Wien | Vienna, 2024

Studienkennzahl lt. Studienblatt | Degree programme code as it appears on the student record sheet:	UA 066 921
Studienrichtung lt. Studienblatt | Degree programme as it appears on the student record sheet:	Masterstudium Informatik
Betreut von | Supervisor:	Mag. Dipl.-Ing. Dr.techn. Maria Leitner Bakk.

Acknowledgements

I would like to thank my supervisor, Univ.-Prof. Dipl.-Ing. Mag. Dr. techn. Maria Leitner, Bakk. for giving me the opportunity to compose my thesis on this interesting subject.

I would further like to thank my partner for supporting and inspiring me throughout this journey.

Above all, I would like to thank my family, for always believing in me, pushing me, and motivating me not just during my studies, but in life.

Finally, I dedicate this work to my mother who, even though she cannot physically experience this, I hope is proud of me.

Abstract

In recent years, there has been increased integration of new, smart technologies in manufacturing settings with the aim to optimize production processes and increase productivity. In particular, cyber-phyisical systems, internet of things, and robotics have become components of high importance in modern facilities.

As robust, mobile, and highly configurable technologies, collaborative robots (cobots) have been increasingly popular as replacements and aids for human workforce in the industrial domain. A special aspect of cobots compared to other industrial robots is that they are designed to work in a shared environment with human operators and other cobots in close proximity, enabling interactivity and collaboration. The collaboration can occur in different ways in terms of spatial, temporal and functional constraints. Depending on the nature of those constraints, collaborative patterns can be identified.

In this work, the goal was to develop a concept for recognizing one of four collaborative patterns (independent, sequential, simultaneous, or supportive) in cobot log data. Specifically, using available equipment with two (collaborative) robot arms, four example scenarios were implemented, reflecting the behavior as defined by the respective collaborative patterns.

The definitions and constraints of the four patterns were used as a basis for developing an algorithm for identifying the pattern occurring in the process logged in a given dataset. Due to limitations of the available equipment, some constraints had to be translated into concepts that can be written to and read from logs. The algorithm is applicable to setups with N cobots and a human operator. Future work might foresee and test different settings.

Furthermore, a software application for mining cobot logs has been developed, which demonstrates the analysis process of the log data by showcasing intermediate results of the steps performed in the background. These steps include reading and parsing the logs, cleaning and transforming the data into a format usable for process discovery, and finally discovering the process and the therein occurring collaborative pattern using the developed algorithm.

Kurzfassung

In den letzen Jahren wurden zunehmend neue, intelligente Technologien in Fertigungsumgebungen integriert mit dem Ziel, Produktionsprozesse zu optimieren und die Produktivität zu steigern. Insbesondere sind cyber-physische Systeme, das Internet der Dinge, und Robotik wichtige Komponenten in modernen Anlagen geworden.

Als robuste, mobile und zu einem hohen Grad konfigurierbare Technologien wurden kollaborative Roboter (Cobots) immer populärer als Ersatz oder Hilfe für menschliche Arbeitskräfte in der Industriedomäne. Ein spezieller Aspekt der Cobots im Vergleich zu anderen Industrieroboter ist, dass sie für das Arbeiten in einer gemeinsamen Arbeitsumgebung mit Menschen und anderen Cobots in unmittelbarer Nähe geschaffen sind, was Interaktivität und Zusammenarbeit ermöglicht. Die Zusammenarbeit kann, in Bezug auf räumliche, zeitliche und funktionale Einschränkungen, auf verschiedene Arten erfolgen. Abhängig von der Art dieser Einschränkungen können unterschiedliche Muster in der Zusammenarbeit erkannt werden.

In dieser Arbeit war das Ziel, ein Konzept zur Erkennung eines der vier kollaborativen Muster (unabhängig, sequentiell, gleichzeitig, oder unterstützend) in den Logdaten eines Cobots zu entwickeln. Konkret wurden, unter Verwendung der zur Verfügung gestellten Ausstattung mit zwei (kollaborativen) Roboterarmen, vier Beispielszenarien implementiert, die das Verhalten, wie in den entsprechenden Mustern definiert, widerspiegeln.

Die Definitionen und Einschränkungen der vier Muster wurden als Basis für die Entwicklung eines Algorithmus verwendet, der das auftretende Muster in einem Prozess identifiziert, der in den gegebenen Logdaten protokolliert wurde. Wegen der Limitierungen der verfügbaren Ausstattung mussten einzelne Muster-Einschränkungen in Konzepte übersetzt werden, die in den Logs geschrieben und aus ihnen gelesen werden können. Der Algorithmus ist für eine Aufstellung mit N Cobots und einem menschlichen Operator anwendbar. In zukünftiger Arbeit könnten auch andere Aufstellungen in Betracht gezogen und getestet werden.

Zusätzlich wurde eine Softwareanwendung für die Analyse von Cobot Logs entwickelt, die den Analyseprozess der Logdaten veranschaulicht, indem die einzelnen Zwischenresultate der Analyseschritte gezeigt werden. Zu diesen Schritten gehören das Lesen und Parsen der Logs, Säubern und Umwandeln der Daten in ein für die Prozesserkennung verwendbares Format, und schließlich Erkennung des Prozesses und des darin vorkommenden Musters mittels des entwickelten Algorithmus.

Contents

List of Tables

List of Figures

List of Algorithms

1 Introduction

We currently live in an era characterized by the ever-growing utilization of technologies such as cloud computing, cyber-physical systems (CPS), internet of things (IoT) and robotics. This era is commonly referred to as the Fourth Industrial Revolution or Industry 4.0, conceptualizing the trends of high automation and digitalization across different industries.

Robots, as important assets in modern industrial facilities, are used to perform difficult, dangerous or repetitive tasks instead of humans, and as programmed machines, they can do so more efficiently: faster and with high precision. The robotics domain has been evolving rapidly, with collaborative robots as one of the newer representatives of that evolution. Collaborative robots are a special subtype of industrial robots, designed to work alongside human operators in a shared workspace.

Considering cobots and their interaction with humans and other robots, it may be of interest to analyze the generated logs during different collaboration scenarios and inspect the presence of recurring patterns in the data. If the recognition and identification of these patterns is performed successfully, a basis for optimizing the human-robot collaboration process may be established. This work focuses on four collaborative patterns (synchronized, cooperation, collaboration, and coexistence) describing collaborative interaction scenarios between robots and human operators, and their identification within robot log data.

In order to detect and identify the mentioned collaborative patterns in the logs, process mining approaches can be used. Process mining is a collection of methods for analyzing business processes based on event logs produced by a company's asset, in this case: cobots. By discovering the process underlying the activities performed by cobots, a starting point for further analysis of the nature of their interaction is constructed. In a subsequent step, the process can be inspected based on certain criteria, which in turn may lead to new insights and conclusions.

The background information on the aforementioned three pillars of this thesis, namely (1) collaborative robots and their log data, (2) collaboration patterns, and (3) process mining, are introduced in more detail in Section 2.1, Section 2.2, and Section 2.3, respectively.

The main goal of this thesis is to utilize process mining techniques to gather, prepare, analyze, and finally evaluate log entries from a Universal Robots UR5e robot arm in order to identify collaborative patterns in the data. More concretely, the aim is to: (1) set up data-acquisition infrastructure to collect log data from robot arms, (2) parse and filter the data to transform it into standard eXtensible Event Stream (XES) [6] format, (3) implement an adequate mining technique for determining occurrences of defined collaborative patterns, and (4) evaluate the implemented approach based on real log data. In the best-case scenario, the results of this work can subsequently be used as basis for process mining and optimization of collaborative tasks performed by humans and robots. In the scope of this work, the aim is to find answers to the following research questions (RQs):

- **RQ1**: How is logging supported by a collaborative robot and how can it be transformed to

reflect the cobot's activity during program execution?

- **RQ2**: How can a (collaborative) production process be discovered based on the cobot's logs?
- **RQ3**: What are the defining characteristics, constraints and attributes of collaborative patterns that can be recognized in the logs?
- **RQ4**: What conclusions can be drawn from evaluating the implemented approach on UR5e cobot logs?

The current chapter introduces the context of the thesis, and defines the goals and research questions that the work focuses on. This thesis is further comprised as follows. Chapter 2 provides relevant background information necessary for better understanding of the concepts used in the project, and motivates the subject by reviewing existing related work. Chapter 3 describes the methodology of approach, including the different phases of the project work and the therein performed steps, as well as their respective goals. Chapter 4 sets the scene by first, explaining the basics of collaborative patterns, and then presenting the scenarios (use cases) designed for the purpose of showcasing these patterns in practice. Chapter 5 answers research question **RQ1** by introducing the data (cobot logs) used as a basis for this research, its relevant characteristics including structure, format and content, as well as the semantics behind it. Furthermore, the chapter highlights the limitations of the available as-is log data, and suggest means for overcoming these limitations and produce more informative logs. Finally, the chapter describes how the data is prepared for further processing. Chapter 6 presents the core of this work, elaborating the concept and the algorithm for discovering collaboration patterns, but also describing the details of the technical implementation: in the physical, as well as the virtual setting. Thus, this chapter provides answers to research questions **RQ2** and **RQ3**. Chapter 7 outlines a walk-through of the evaluation process from scenario implementation, to a demonstration of the developed project based on real logs from Universal Robots (UR) cobots, to a reflection on the results and challenges, answering research question **RQ4**. Chapter 8 concludes the thesis with a summary of work done and insights gained in this project.

2 Background and Related Work

2.1 Collaborative Robots

As opposed to traditional industrial robots, which are typically large and heavy machines programmed to perform (for humans) difficult, application-specific tasks and are isolated from human operators, collaborative robots (so-called cobots) are mobile, as they are lighter in weight, highly flexible, as they can perform a variety of different tasks and can thus be implemented in different industries, and finally, they are easily (re-)programmable and are designed to work safely alongside humans [15]. Introducing joint operation of human workers and robots yields many benefits in terms of performance during task execution in production processes. These benefits are enabled by combining the flexibility and cognitive abilities of humans with high precision, speed and repeatability of robots [5]. In the scope of this thesis, we base our work on the UR UR5e collaborative robot arm [20] which is depicted in Figure 2.1. The UR cobots generate log data from the moment they are powered on until they are powered off. There are different types of logs, depending on factors such as the source (component producing the log) and the nature of the log (joint calibration, movement, power, system state, software output). In the scope of this thesis, we focus the analysis on program execution logs, as we are interested in interactivity aspects of the cobot's "behavior" while a program is running. Our experimental cell consists of two UR5e robot arms, and the constructed collaborative use cases involve one human operator.

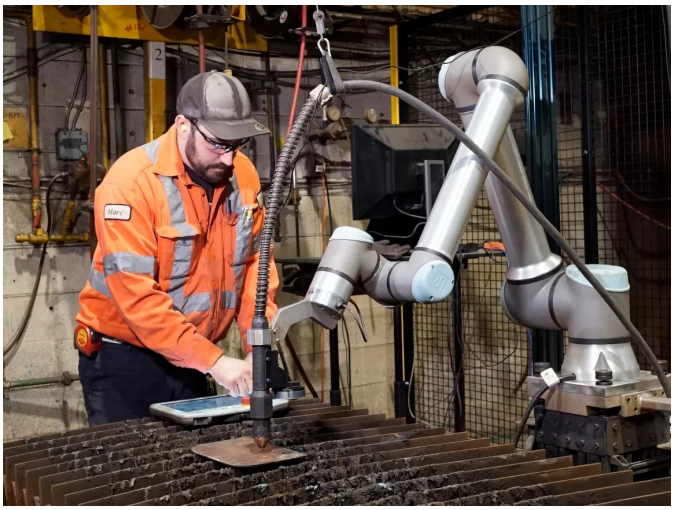


Figure 2.1: A Universal Robots collaborative robot arm in a production environment. [23]

2.2 Collaborative Patterns

In the described context of human-robot interaction, we adopt and assume the following definition of a collaborative pattern, as described by Samhaber and Leitner [14]: "A Collaborative Pattern (CP) describes a collaborative working environment where humans and cobots may interact. The main elements of a collaborative pattern are a process, actors, a workpiece type, and constraints." Furthermore, this work focuses on four collaborative patterns in particular, which are also thoroughly defined in the paper by Samhaber and Leitner and are here sorted by the level of interactivity and dependence between collaboration parties:

- **CP1: Coexistence.** This pattern specifies that individual actors work independently of each other; on different types of workpieces and execute independent processes.
- **CP2: Synchronized.** In this pattern, the actors perform their tasks on the same workpiece type, whereby the tasks are performed at different times and one of them requires the output of the other task in order to be executed.
- **CP3: Cooperation.** This pattern is similar to CP2 as the actors perform tasks on the same type of workpiece. However, the tasks are spatially constrained instead of timely. As the cooperating parties perform their tasks simultaneously, they need to be aware of spatial constraints in order to ensure safety.
- **CP4: Collaboration.** In this case, there is the highest level of interactivity among the involved actors, as they work together to achieve a common goal. Their actions are partially functionally dependent on each other which means certain tasks rely on each other in order to be executed successfully.

A depiction of the described patterns is given in Figure 2.2. Note that the definitions of the collaborative patterns will be further specified in Chapter 4 as they will be used in the design and implementation of the collaborative pattern recognition module.

In order to reliably detect or recognize the described patterns when observing the collaboration of human operators and cobots, apart from the log data produced by the cobots and formal definitions of the patterns, one more component is needed: a set of methods and mechanisms for the analysis of the log data.

2.3 Process Mining

The performance, efficiency and productivity of any utilized system can virtually always be improved and optimized. Often, to determine the weaknesses, potential issues or even just aspects which can be improved, analytic approaches such as process mining are used. The analysis of event streams is the central focus of process mining. Process mining is a relatively young research field concerned with the discovery, analysis and improvement of processes based on available event data produced by machines and human operators. This event data, so-called *event logs*, can be used to reconstruct actually occurring activities in an existing business process. Three basic types of process mining can be distinguished: (1) discovery, (2) conformance checking, and (3)

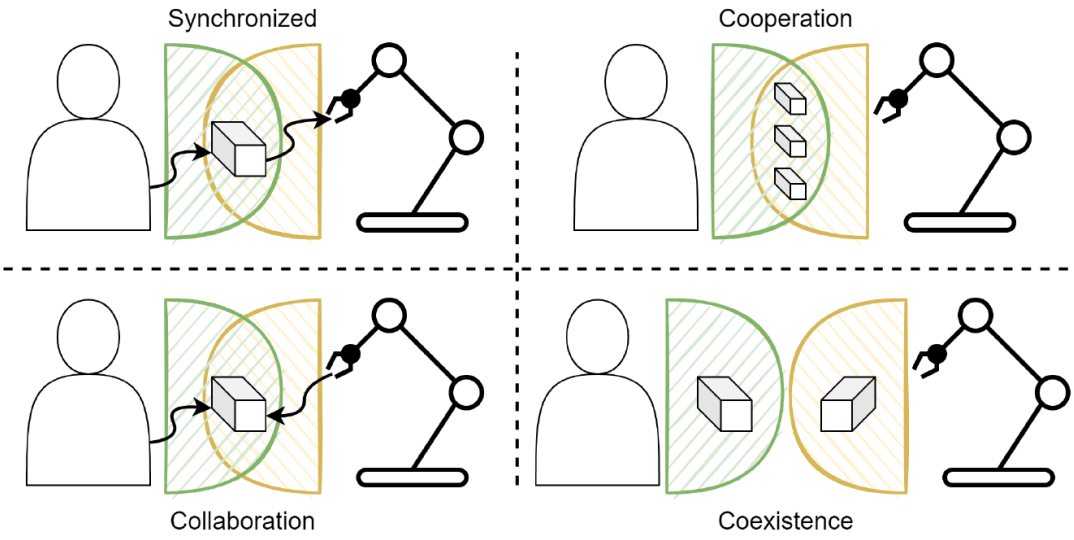


Figure 2.2: Collaborative patterns. [14]

enhancement. [25] These three types are depicted in Figure 2.3, along with the interconnections among different components which process mining is based on.

Note that this thesis only utilizes process discovery, and therefore does not cover any conformance checking or enhancement methods.

2.4 Related Work: Mining Approaches in Industry 4.0

In a case study conducted in 2017 by Oliff and Liu [9], the authors implemented a cost-effective way of mining a historical dataset (a collection of re-work records with short textual descriptions of observed faults, and the actions taken to correct these faults), with the aim to develop a basis for a decision-support system to improve material flow process quality. The authors utilized simple, yet effective tools do develop a proof-of-concept; WEKA (Waikato Environment for Knowledge Acquisition), a cost-free data-mining software with application of rule-based supervised learning algorithms. However, as the authors report, the entire process requires significant amounts of human activity, in terms of processing, as well as interpretation. The results of the study were successfully found patterns of common errors and issues with machine components and devices where they occur. In contrast, this work will focus on patterns of collaborative patterns, and the aim to minimize human effort and interference as much as feasible to create a majorly automated system.

Another related work by Atzmueller and Kloepper [1] analyzes log data by converting them to attributed interaction networks, and using descriptive community mining methods in order to discover patterns in the structure of production systems. This enables the recreation of material flow in the production solely by analyzing machine and device component logs which indicate the interaction patterns among the components in the production line, as well as the frequency of

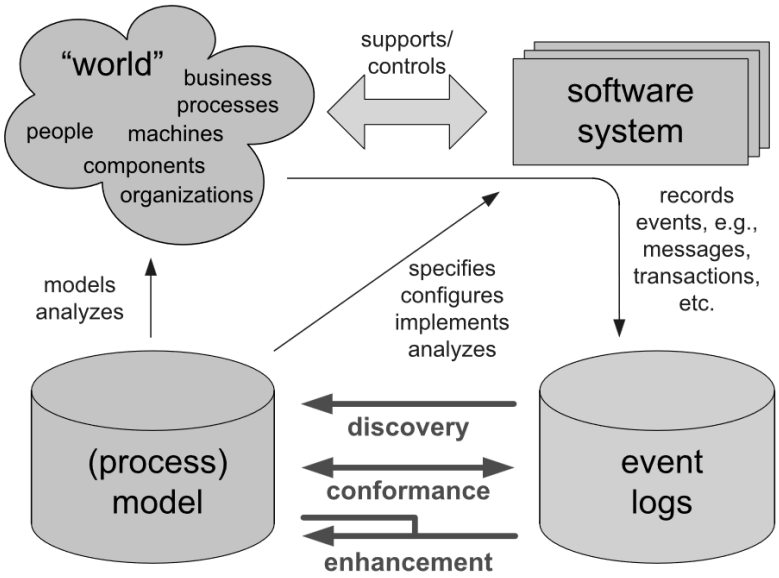


Figure 2.3: The three basic types of process mining: (a) discovery, (b) conformance, and (c) enhancement. [25]

their occurrences. As the goal in this thesis is not the reconstruction of the material flow, and we also do not possess the resources (many devices from multiple production lines) for which the interaction networks would be applicable and of interest, our approaches and aims differ from this work.

Roldán et al. [13] propose the utilization of process mining as a tool of analysis for multi-robot missions with the ultimate goal to improve the mission execution by detecting bottlenecks and inefficiencies. The base of this analysis process were event logs, generated not only by the robots themselves, but also other algorithms involved in the mission, such as those executed from the interface, or by post-mission telemetry-processing scripts. The event logs are stored into spreadsheets, in formats such as XLS and CSV. In the described work, the experiments involved a few different robots: two drones and one ground robot. This thesis differs from the paper by Roldán et al. in that it considers cobots specifically as a special type of (industrial) robot, and furthermore, does not focus on "fleets" of robots and multi-robot missions.

Another process-mining focused approach is used in a paper by Duong et al. [2], where the goals of process log analysis is more product-oriented: the paper proposes a method to compute a (product's) penalty index as a relevant indicator of product quality. Here, process-mining approaches are applied to discover and represent the underlying model, as well as to check product path conformance. For evaluation purposes, the proposed framework was applied to a real assembly process. The described work strongly focuses on product quality and extracting relevant information for other business and economics-related entities such as the after-sales service. In contrast, this thesis will make use of (process) logs for determining patterns which are relevant for the collaboration processes.

In a more recent paper by Sulhi [17], the author investigates the usage of data mining methods in the case of IoT-based intelligent manufacturing. In particular, the focus of the research lies on the utilization of frequently found data mining algorithms such as association rule mining (ARM) and apriori algortihm (AA) for processing information from a decision support system (DSS). In the paper, a theoretical framework is proposed which demonstrates the benefits of using data mining technologies for analyzing data of decision support systems for intelligent manufacturing, and, subsequently, supplying the decision makers with significant insights and information to improve the decision-making process. As opposed to the goals of Sulhi's research, which also focuses on data mining and the (intelligent) manufacturing domain but aims at improving a DSS, this thesis aims at pattern-discovery in the data which could aid in future optimization work on collaborative tasks in a human-robot shared workspace.

3 Methodology

Before taking concrete steps towards process discovery within the cobot logs, extensive research in terms of existing approaches and related work needs to be carried out first. With sufficient background knowledge on the topic, it can be proceeded with the attempt to answer the formulated research questions listed in Chapter 1. In the "Data acquisition and preparation" phase, infrastructure is set up for gathering system logs from the robot and transferring them to another endpoint (local machine, workstation), where a program for parsing, filtering, and converting the data into valid XES format is developed. The acquired dataset is then transformed into XES using the created program. Next, the resulting, transformed data is analyzed to identify attributes that can be used for process mining. Based on the defined attributes from the previous step, a process mining approach combining custom analysis functions and a process discovery algorithm to extract occurring patterns is implemented. This is the "Process discovery" phase. Finally, the implemented approach is evaluated by applying it to the real log data acquired from the robot (UR5e) in the "Evaluation" phase. An overview of the described approach including the phases, steps and corresponding goals is given in Figure 3.1.

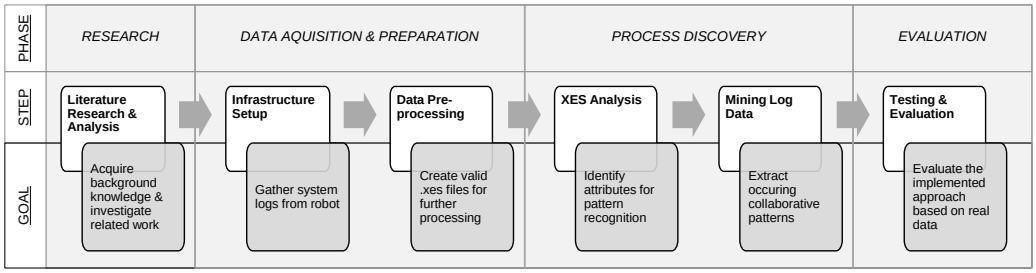


Figure 3.1: Methodology of approach

3.1 Research Phase

First, the UR cobots were thoroughly researched, especially with regards to the logging mechanisms, structure and organization of the logged data, as well as logging during robot program execution. In this phase, the main goal was to investigate if and how a cobot's activities performed during program execution are logged. This phase involved online research including available vendor documentation, forum discussions, as well as specific UR software documentation such as the documentation of the UR Log Viewer [24], where certain aspects of the log file (also called *support file*) are described and written in a human-readable and more comprehensive format

(e.g., names of certain components instead of their alphanumeric codes). Additionally, manual inspection of the raw log files and their representation in the mentioned UR Log Viewer was conducted. The gained insights were then used in the subsequent phase, where the logs were acquired and prepared for further processing.

3.2 Data Acquisition and Preparation Phase

In this phase, methods for reading and parsing, as well as transforming the cobot logs were implemented, based on the known structure and semantics of the available log data. In particular, a software module for reading the UR support file was developed, which loads the available log data into a table-like format with named columns according to the gained knowledge from the research phase. The read logs are then cleaned and transformed into a format with attributes adhering to the XES Standard, to enable further processing and analysis of the log data in the process discovery phase.

3.3 Process Discovery Phase

This phase represents the core of this project, encompassing (1) process discovery based on the transformed log format, (2) definition and extraction of relevant attributes for determining the occurrence of a certain collaboration pattern, and (3) the development of a pattern mining module. The pattern mining module is built based on the process discovery results produced by the inductive miner algorithm, as well as the determined attributes characterizing the respective collaboration patterns. Specifically, in order for a certain pattern to be detected, the corresponding defined constraints have to be fulfilled in the relevant attributes of the log data.

3.4 Evaluation Phase

Finally, in the evaluation phase, the developed pattern mining algorithm is tested and evaluated using real cobot log data. Concretely, four collaboration scenarios were designed and implemented on two UR5e robot arms. Apart from the two cobots, the scenarios involve a human operator with a supporting and managing role. Each of the four developed scenarios consists of program executions on both cobots. The scenarios require the operator's intervention and signaling before each iteration, and multiple iterations are needed to produce sufficient log entries for analysis. Due to limited resources, the scenarios were transferred and adapted to a simulated environment of the UR cobots (URSim [21]). In total, fifty iterations of each scenario were performed. The logs generated by the two cobots were then read, transformed and processed to discover the occurred collaboration pattern based on their log entries during program execution.

4 Collaboration Scenarios

4.1 Background: Collaborative Patterns

The work in this thesis builds upon the concept of *collaborative patterns (CP)*, as defined in the work of Samhaber and Leitner [14]. In particular, the authors define a collaborative pattern as a collaborative working environment where humans and cobots may interact. A CP has four main elements: (1) a process, (2) actors, (3) a workpiece type, and (4) constraints. According to the authors, these elements can be roughly defined as follows. A **process** is a collection of tasks, where each task (simple action) is performed by one actor. An **actor** is an individual executing tasks that, in the context of collaboration scenarios, can either be a human operator or a cobot. The tasks are performed on objects called **workpieces**: defined as inputs or outputs of processes or tasks. To enable safe and efficient collaborative activities, certain constraints are introduced. A **constraint** describes a dependency between tasks within a process and falls into one of three categories, depending on its type:

- **Time-based**, where the output of one task is needed for the execution of another, and thereby the execution order of the two interdependent tasks is clearly defined.
- **Spatial**, where the actors need to access the same working area (space) to perform a task.
- **Functional**, where tasks across actors rely on each other for successful execution and the actors work interactively together to reach a common goal.

Based on differences in terms of the described elements, four collaboration patterns can be distinguished. However, not all elements are relevant for differentiating between patterns. In every pattern, at least one human operator and one cobot represent the actors. Moreover, the tasks within processes are always carried out by a single actor, which also holds for all patterns. Thus, the relevant elements for identifying a pattern are the workpieces and the constraints between tasks of different actors. As mentioned in Section 2.2, the patterns are namely

1. **synchronized**, where the execution of one actor's task depends on the output of another actor's task, making the tasks ordered sequentially one after another (time-based constraint) and they are performed on the same workpiece,
2. **cooperation**, where the tasks of different actors are not performed in sequence (i.e., no time-based constraint), but rather in parallel, performed on the same workpiece, which requires a spatial constraint to be met in order to ensure safety,
3. **collaboration**, where the tasks of different actors depend on each other in order to be successfully executed (functional constraint), working on the same workpiece to achieve a common goal, i.e. perform a joint process, and

4. **coexistence**, where there are no dependencies (no constraints) between the tasks of different actors and the actors execute their tasks on different workpieces.

In order to generate cobot log data containing relevant information for identifying a collaborative pattern, certain use cases or scenarios need to be developed where these patterns occur in the interactive behavior during program execution. To provide example use cases for each of the four described collaborative patterns, four scenarios involving a human operator and two cobots have been designed. Details on the scenario design and reasoning behind it are given in the next section.

4.2 Scenario Design

As mentioned in Section 4.1, four scenarios corresponding to the four collaborative patterns have been developed to enable demonstrating the patterns' occurrence, as well as their detection in the subsequent analysis of the log data. The first scenario demonstrates the synchronized pattern, where the cobots operate in succession on a single workpiece. In the second scenario, the cooperation cooperation pattern is demonstrated, where the cobots simultaneously perform their tasks on a workpiece. The third scenario demonstrates the highest level of interactivity and collaboration between the cobots, where one cobot is assisting the other in performing its task. Finally, the fourth scenario represents the case where the cobots operate independently on different workpieces and thus demonstrates the coexistence pattern. Each of these scenarios is further elaborated in the following subsections.

4.2.1 Assumptions

Before moving onto detailed descriptions of the scenarios, a few assumptions should be mentioned that describe the commonalities of the four scenarios. In the designed scenarios[1] the actors are a single human operator having a primarily supportive role, and two cobots demonstrating different levels of collaboration when carrying out their tasks on given workpieces. The focus lies on the interactions between the cobots, while the operator enables seamless operation of the cobots by setting up the working environment and supervising the process. Another assumption is that the scenarios are designed for two cobots (named Robot A and Robot B for simplicity), one of which has a vacuum tool attached (see Figure 4.1 left), while the other has a two-finger gripper tool (see Figure 4.1 right). Neither of the two cobots have movement sensors or cameras mounted. Thus, the interaction among the three actors (i.e., human operator and two cobots) is realized through digital signal exchange. A signal is also what initiates the start of each iteration of a scenario execution. There is no predefined number of iterations. Instead, the human operator decides at the end of each iteration if another should be initiated. This is because the operator's role includes "setting the stage" before each iteration, such as preparation of workpieces and manual safety checks. Another assumption regarding the mechanics of the setup is that there is a single point of access from the operator to the robots, i.e., there is only one control board with signal pins, which is connected to Robot A. This is why Robot A needs to "propagate" the start signal

[1] Note that the scenario workflows are visualized using Business Process Model and Notation (BPMN) 2.0 [10] models.

of the operator in scenarios where the cobots' operations should be executed simultaneously or Robot B needs to execute a task first. Moreover, for consistency and safety reasons, the scenarios are designed such that at the end of each iteration the cobots return to their respective "Home" positions.

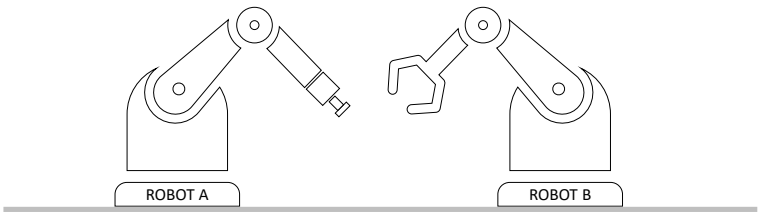


Figure 4.1: Illustration of the two collaborative robot arms for which the scenarios were designed.

4.2.2 Scenario 1: Successive Pick-and-Place

In this scenario, the cobots perform *pick and place* operations with a box as a workpiece in sequence. This means that before Robot B can execute its operation, Robot A must have finished its last task. Here, the order of the tasks is clearly defined, as Robot B needs the output of the operation of Robot A in order to proceed. Figure 4.2 illustrates this scenario: at a point in time $t = 1$ Robot A is working on workpiece $Box\ M$, and at a subsequent point in time $t = 2$ Robot B is performing its operation on the same workpiece.

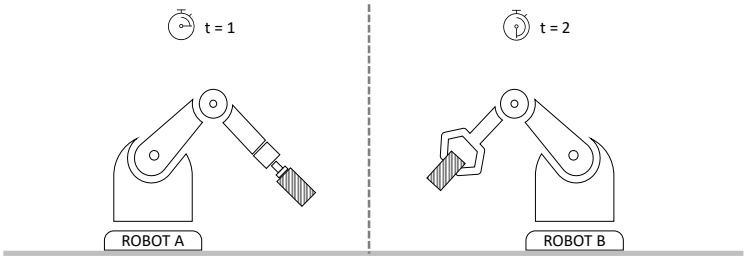


Figure 4.2: Illustration of scenario 1: Successive Pick-and-Place

A graphical representation of the described workflow can be seen in Figure 4.3. The process model shows that the activities of the three actors (Human Operator, Robot A and Robot B) are carried out in sequence. Moreover, it can be seen that the "transition" from the last activity of Robot A to the first activity of Robot B is enabled by a "GO" signal. This ensures that only after one cobot is done with its tasks, the other can start with its operation. The model also illustrates that one such sequence of activities comprises one iteration of the scenario execution, and the human operator decides whether another iteration should be run, or the process terminated.

4.2.3 Scenario 2: Pack-and-Paint

In this scenario, the two cobots perform different processes on the same workpiece at the same time. Concretely, Robot A packs an item into $Box\ L$, while Robot A paints its lid (see Figure 4.4). This scenario represents the cooperation pattern, as the cobots work simultaneously on the same workpiece, and their processes are functionally and temporally independent.

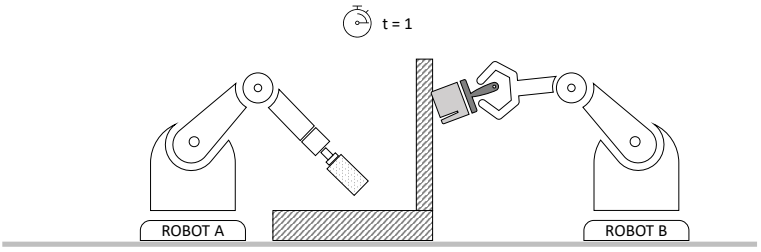


Figure 4.4: Illustration of scenario 2: Pack-and-Paint

A BPMN diagram representing this workflow is shown in Figure 4.5. Notice how the parallel execution of the cobots' independent tasks is enabled by signal propagation from Robot A to Robot B upon iteration start. An iteration is complete when both cobots complete their tasks on the same workpiece.

4.2.4 Scenario 3: Assisted Packing

This scenario demonstrates the highest level of interdependence and collaboration out of the four use cases, as there is a functional constraint between the cobots' tasks. In particular, Robot B is holding $Box\ S$ at a reachable and easily accessible position for Robot A, while Robot A is placing business cards into the box. As can be seen in the simplified illustration of the use case in Figure 4.6, the cobots collaborate in order to achieve a common goal: pack the business cards. After Robot A has finished placing the business cards into $Box\ S$, it sends a signal to Robot B, which in turn places the box at a designated spot.

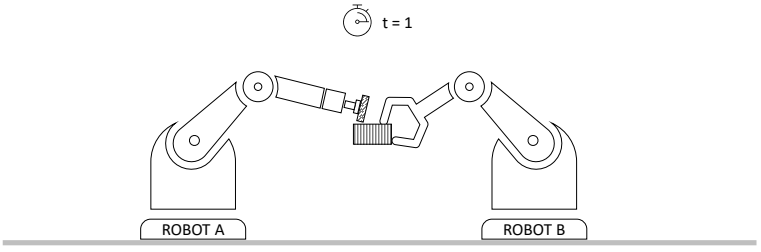


Figure 4.6: Illustration of scenario 3: Assisted Packing

Figure 4.7 visualizes the described workflow. Here, there are more exchanged signals compared to the other scenarios, which reflects the higher level of interaction necessary for coordinating the

tasks of the cobots. First, Robot A propagates the start signal to Robot B which initiates its first task: to move into position for packing. After Robot B has reached the target position holding the box, it sends a signal to Robot A which can then start packing the cards into the box. When Robot A has completed its packing tasks, it informs Robot B about the completion by sending a "FIN" signal. This signifies Robot B it can move the packed box to the designated location.

4.2.5 Scenario 4: Independent Pick-and-Place

This scenario is similar to Scenario 1 (Section 4.2.2) in that the cobots perform the same activities. However, their processes are independent in every sense (temporally, functionally and spatially) and performed on different workpieces. This scenario is an example of the case when the behavioral pattern of the cobots during execution does not match any of the other patterns in terms of collaborative activities. In short, if no interdependence between the cobots and their tasks is found, the occurring pattern is identified as *coexistence*.

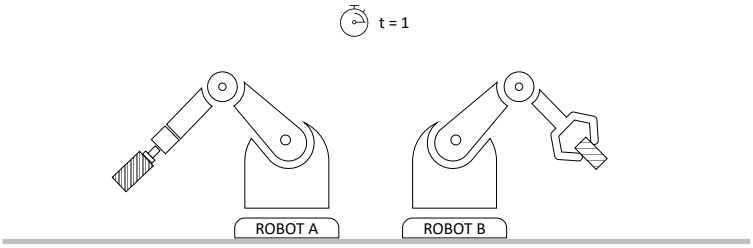


Figure 4.8: Illustration of scenario 4: Independent Pick-and-Place

The BPMN model representing this scenario is depicted in Figure 4.9. As this is the case with the highest level of independence and consequently lowest level of interaction, the operations of the two cobots are shown as separate processes. However, because of the mechanical restrictions mentioned in Section 4.2.1 (Assumptions), Robot A still needs to trigger the start of Robot B's execution by sending the "GO" signal when it is not in a position where a collision risk exists (hence the signal is sent only after Robot A has returned to "Home" position).

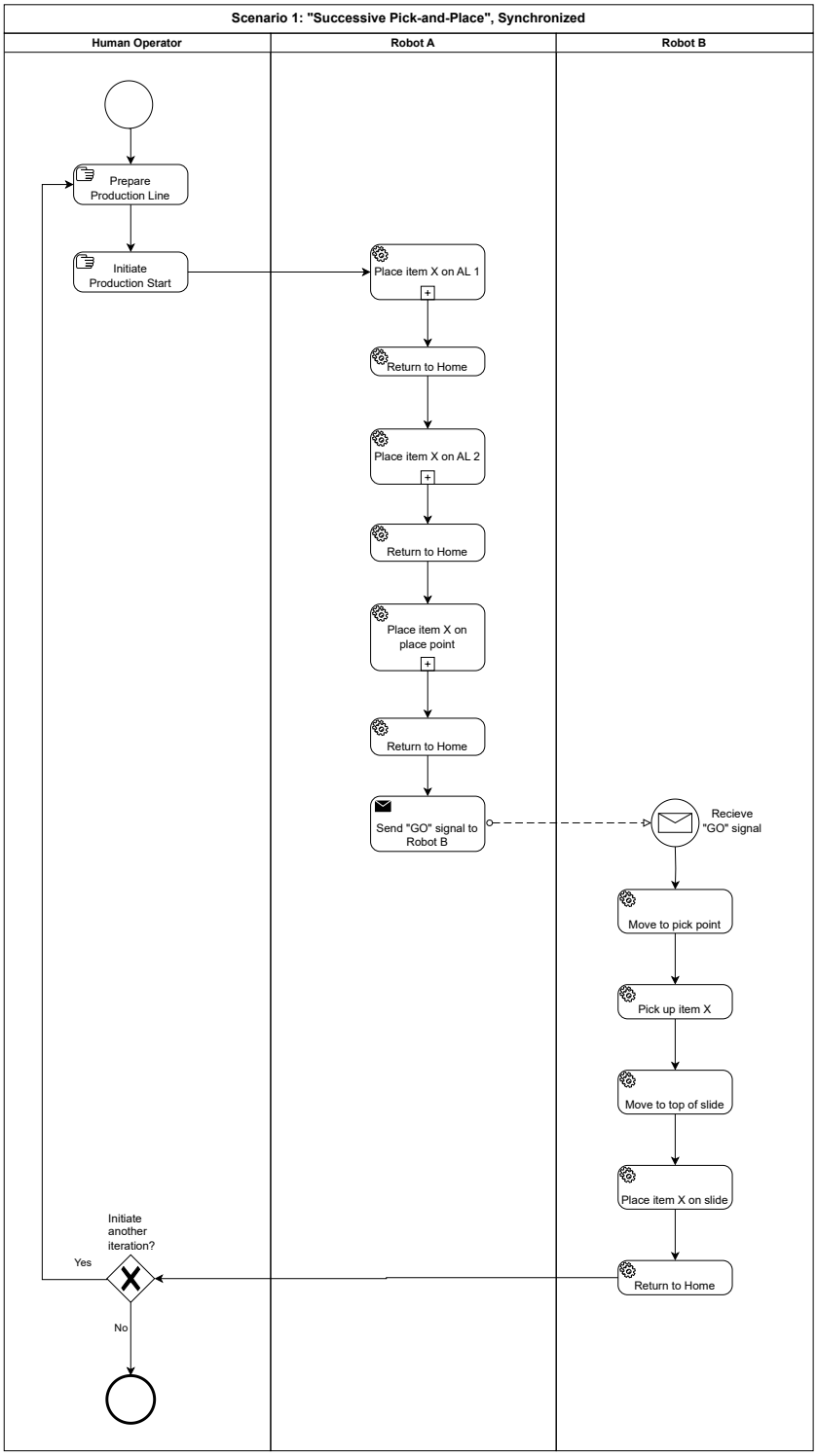


Figure 4.3: Scenario 1

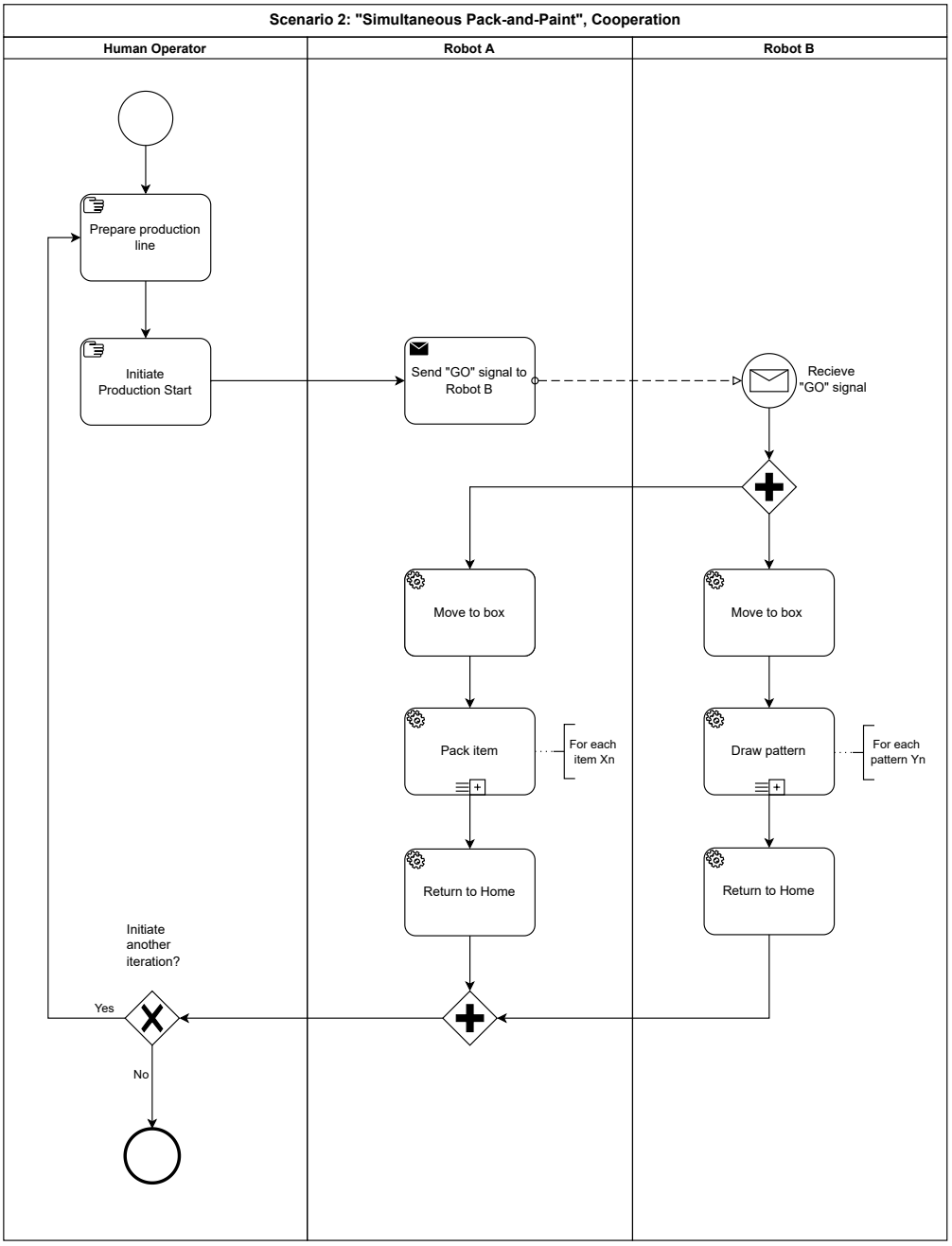


Figure 4.5: Scenario 2

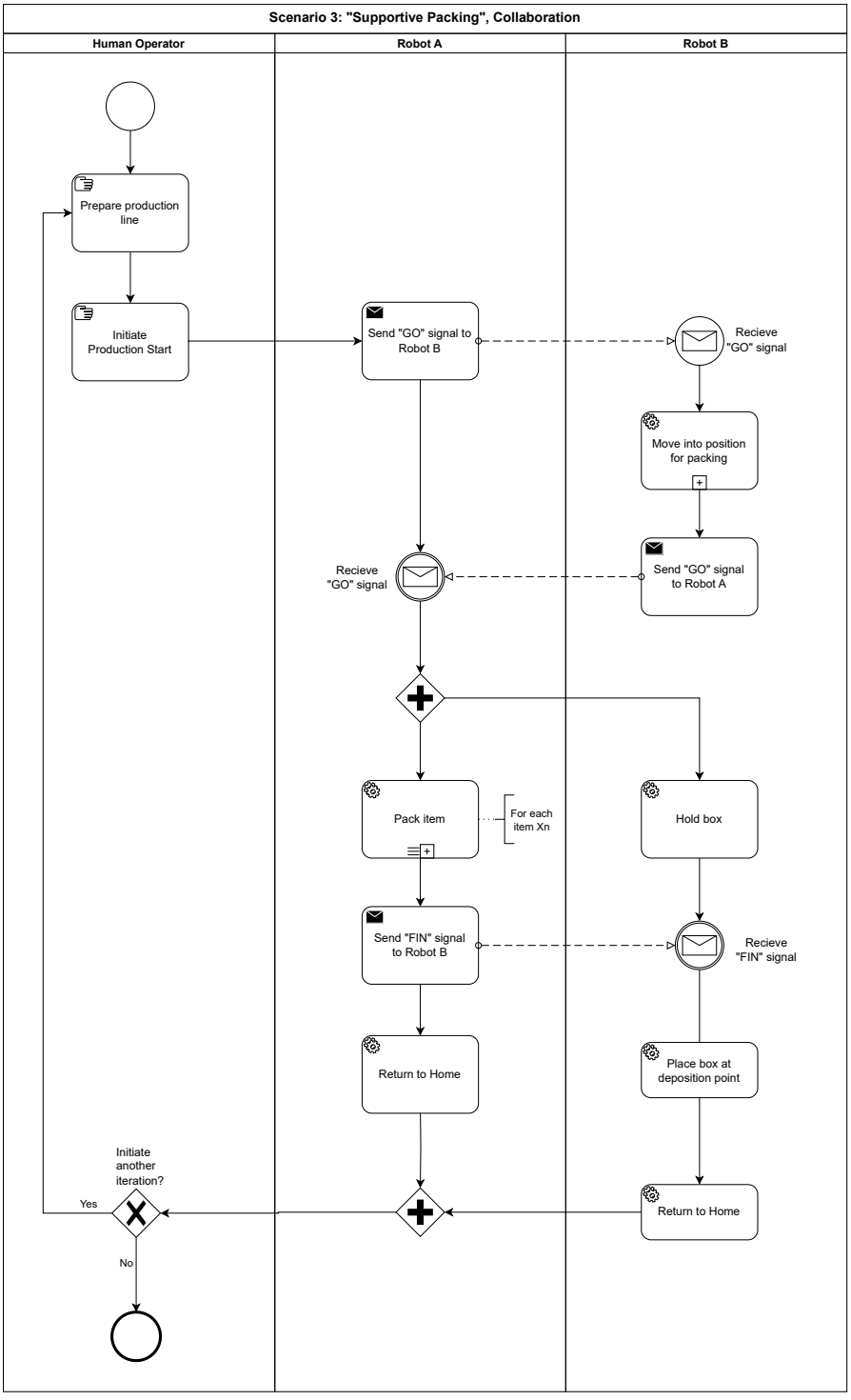


Figure 4.7: Scenario 3

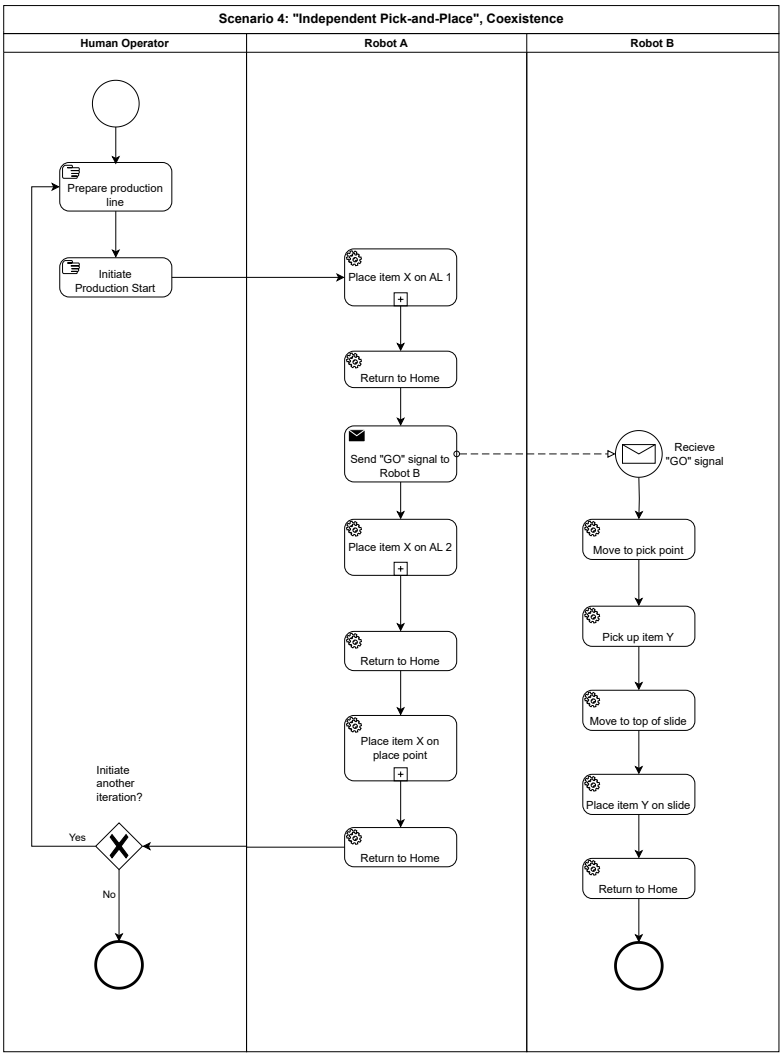


Figure 4.9: Scenario 4

5 Data Acquisition and Preparation

5.1 Cobot Logs: Analysis based on Universal Robots UR5

This work is based on the analysis of logs produced by collaborative robot arms. Prior to discussing the analysis process, it is necessary to introduce some background information on the log data itself: its structure and contained information regarding cobot operation.

As mentioned before, logs are automatically generated datasets carrying information on events happening in a system. A row in such a dataset is a log entry. Each log entry is created at a certain point in time, which is represented by a corresponding timestamp. Using timestamps, the events in the system described by log entries can be ordered and visualized, producing workflows. Since this work is based on log data from Universal Robots cobot UR5, the scope of the cobot log analysis is limited to their specific format and content. Thus, the log data from other collaborative robots may differ from the here described logs.

In the following sections different aspects of UR5 logs are described. First, data acquisition through supported logging and export mechanisms of the cobots is explained. Then, an analysis of the acquired logs' structure and content is provided, including relevance and limitations of the contained information. Last but not least, a solution for countering the limitations of existing log data using custom log outputs is presented.

5.1.1 Supported Logging and Export Mechanisms

UR5 cobots generate a variety of data files logging different aspects of their operation. Some of the files created by a cobot include, for instance: calibration information (joint positioning), power and voltage measurements, safety settings and system configuration. In the scope of this thesis, we focus our analysis on event-based log files which include timestamps, sources of produced log entries, as well as error codes and program execution outputs. On an UR5 cobot, a file containing the described event information is stored in a file called $log\_history.txt$. This file, along with the other support files covering the aforementioned aspects of the cobot's state and operation, can be exported in a compressed $.ZIP$ bundle on demand via the cobot's interface (see Figure 5.1).

On a real, physical robot, it is recommended to store the exported bundle on an external storage device (such as a USB drive). This export mechanism is easy, intuitive and can be used by users with all levels of experience with UR cobots. Advanced users familiar with the UR cobots' file system may transfer the $log\_history.txt$ file directly to another machine via, e.g., a Secure Shell Protocol (SSH) connection. When working with a simulated setup (URSim), the support-file bundle can be exported to a folder shared between the host and guest operating system. Other, advanced methods include an SSH transfer or copying the $log\_history.txt$ file from the original folder on the guest file system to the shared folder.

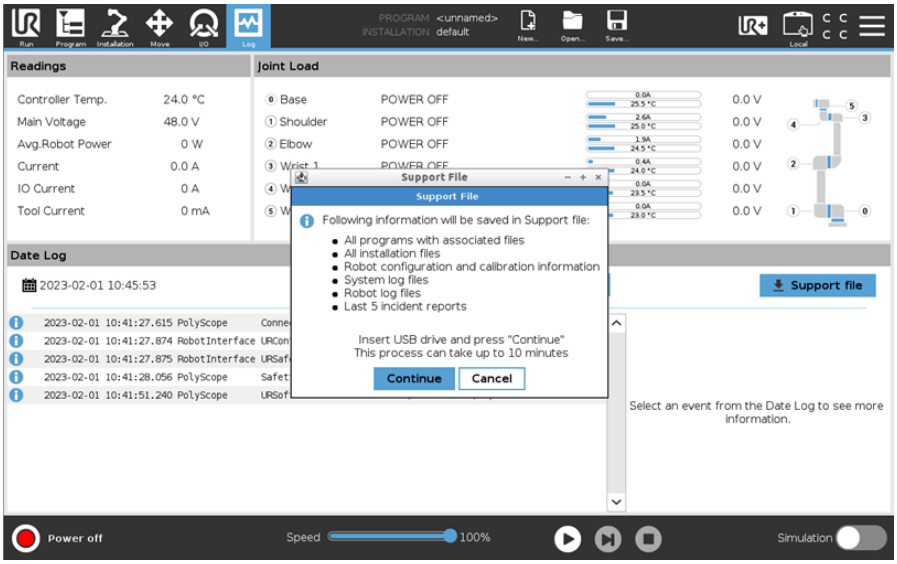


Figure 5.1: Support bundle export via the UR cobot interface

Essentially, the exported folder has the following structure:

- ur_<PolyScope serial number>_<YYYY-MM-DD>_<hh-mm-ss>
  - recording<YYYYMMDD>_<hh_mm_ss>
  - ...
  - robot_configuration<YYYYMMDD>_<hh_mm_ss>
    - files
      - interpreter
      - programs
      - robot_metrics
      - urcontrol
      - <installation>.variables
      - kern.log
      - log_history.txt
      - polyscope.threads.txt
    - summary.log

The underlined elements, i.e., the **programs** folder, and the **log_history.txt** file are the most relevant in the scope of this work and will be referred to in the remainder of the thesis.

5.1.2 Log Structure and Content Analysis

In this section, the analysis of the structure, as well as the content of the **log_history.txt** file is presented. Every time the cobot's controller is powered on, a new log section is started. Each of these log sections starts with a header string containing the current date and time:

`******** Log start (YYYY-MM-DD hh:mm:ss) ********`

In the "body" of a log section, the actual log entries are listed. Each log entry contains 10 columns, and the column separator is " :: ". Out of the 10 columns, which contain different UR-specific codes and numbers, the most relevant in the context of event log data are the following:

- **Timestamp.** The moment in time when the event was logged. It is given in the format `YYYY-MM-DD hh:mm:ss.fff`.
- **Source.** This is a UR-specific number indicating the component of the cobot which produced the log. The three most relevant log sources are:
  - *RobotInterface.* This is the interface between the physical robot and the controller. It is responsible for logs reflecting e.g., the robot's power mode (confirm safety, booting, on, off, idle, ok) and safety mode (normal, safeguard stop, emergency stop).
  - *PolyScope.* This is the GUI responsible for logging certain controller connectivity and execution information, such as the software version upon booting, the connection status of the controller, safety checksums, and program start with information on when the program was last saved.
  - *RTMachine.* This is the **R**un**t**imeMachine, an execution engine for the URScript programming language. It produces log outputs related to program execution (started, stopped, paused), as well as the used URcaps which are essentially the drivers for attached end-effectors or tools such as grippers. As this component is responsible for outputs from the robot program execution, it also generates the custom log entries, which are explained in detail in the next section.
- **Event code.** The event code is also a UR-specific string and represents the complete event description including the source, the nature of the log (whether it is a warning, an error or just an info message), and its description.
- **Program output.** For the majority of the log entries, this column is either NULL or empty. However, every output produced by an executed cobot program is visible in this column, which is of high importance for custom log outputs (described in Section 5.1.3).

Usually, each log section ends with a special string, analogously to the start:

`******** Log end (YYYY-MM-DD hh:mm:ss) ********`

However, manual log inspections revealed that there are log entries between the end indicator of one section and the start indicator of the next. These "unclaimed" log entries logically belong to the preceding log section.

In the process of cleaning and filtering the log data, the start and end indicators, as well as any empty lines, are removed from the logs, i.e., they are not used further in the analysis.

Limitations of Program-Execution Logs

As mentioned in previous sections, the most important aspect of the cobots' operation for analysis is the program execution, i.e., what is happening while a certain cobot program is running and

the cobot is performing different activities within the collaborative workspace. This is crucial not only for recognizing collaborative patterns, but also discovering alternative paths or event flows in the process such as erroneous iterations. Unfortunately, the log file is limited in this regard: there do not seem to be any "built-in" mechanisms for logging the executed steps *during* program execution. The logs only contain information on program state, i.e., "started", "stopped" and "paused". For discovering the patterns occurring in a cobot's activities, it is necessary to log these activities. This means that outputs such as "Move to X", "Pick up B", and "Send Signal S" are needed in order to analyze the processes. To tackle this incompleteness in the UR logs, custom log outputs were designed which are written to the `log_history.txt` during program execution. These custom logs are described in the following section.

5.1.3 Custom Log Outputs

Background: XES Event Logs

The main guideline for designing custom log outputs is the standard XES format and the definition of the therein comprised concepts. Therefore, it is necessary to define the different concepts before diving into the design choices.

A **log** contains all event information related to a specific process. A **trace** represents the execution of a specific process instance, or case, of the logged process. An **event** is an atomic entity of activity observed during the execution of a process (trace). Lastly, **attributes** are elements which store all the information on logs, traces and events: they describe their parent elements. A simplified example of a XES log is given in Figure 5.2.

```
<?xml version='1.0' encoding='UTF-8'?>
<log xes.version="1849-2016">
  <string key="origin" value="csv"/>
  <extension name="Concept" prefix="concept" uri="http://www.xes-standard.org/concept.xesext"/>
  <extension name="Organizational" prefix="org" uri="http://www.xes-standard.org/org.xesext"/>
  <extension name="Cost" prefix="cost" uri="http://www.xes-standard.org/cost.xesext"/>
  <extension name="Time" prefix="time" uri="http://www.xes-standard.org/time.xesext"/>
  <trace>
    <string key="concept:name" value="1"/>
    <event>
      <string key="concept:name" value="register request"/>
      <date key="time:timestamp" value="2010-12-30T11:02:00.000+01:00"/>
      <int key="cost:total" value="50"/>
      <string key="org:resource" value="Pete"/>
      <int key="@@index" value="14"/>
    </event>
    <event>
      <string key="concept:name" value="examine thoroughly"/>
      <date key="time:timestamp" value="2010-12-31T10:06:00.000+01:00"/>
      <int key="cost:total" value="400"/>
      <string key="org:resource" value="Sue"/>
      <int key="@@index" value="15"/>
    </event>
  </trace>
</log>
```

Figure 5.2: Example XES (Adapted from "Running Example" by Fraunhofer Institute for Applied Information Technology [3])

As the example shows, an XES file is an XML-based file, where the logs, traces and events are structured in a tree-like manner: a log (root element) contains one or more traces, while a trace contains one or more events. Furthermore, it can be seen that each of these elements contain attribute elements describing them. The attributes of events are especially important for process analysis, because they contain valuable information of the events' occurrence. For instance, the first event in the first trace represents a "register request" activity performed by Pete on December 30^{th} 2010, and the cost of this activity equals 50. The next day, Sue carries out the "examine thoroughly" action which costs 400. The trace ends with the second event. Attributes generate added value through the contained information on the events, as they can provide insights into the order of tasks, their duration, the associated costs, responsible actors, and so on.

Since attributes are the carriers of all event-related information, they are the central focus of the design process for custom log outputs. Apart from the "mandatory" attributes defining a specific event (case ID, event ID, timestamp and activity name), for detecting the collaborative pattern some additional considerations need to be taken into account. A detailed description of the attributes design is given in the next section.

Attributes Design for Cobot Activities

The designed attributes for cobot activities executed during a running program are listed in Table 5.1. As mentioned before, some attributes need to be defined for the log to adhere to the XES standard and to be used by a discovery algorithm later (top four attributes listed in the table). The other attributes in the table represent additional information on the cobot activities that is relevant for describing the activities and ultimately discovering the occurring collaboration pattern.

Table 5.1: Specification of the different attributes required for both XES format conformance and collaboration pattern mining.

Attribute	Format/Type	Description	Value Range
CaseID	Positive integer	Incremental case identifier within a log	$\mathbb{N}$
EventID	Non-negative integer	Incremental event identifier within a case	$\mathbb{N}_0$
Timestamp	Datetime, YYYY-MM-DD hh:mm:ss	Logged time of execution	[677-09-21 00:12:43, 2262-04-11 23:47:16]
Activity	String	Name of an event	{Draw, Move, Pick, Place, Signal}
Lifecycle	String	Specifier for lifecycle state of an event	{start, end}
ActorID	String	Identifier for the actor performing the activity	{Operator, Robot-A, Robot-B}
Target	String	Specifier for the object involved in the activity	see Targets section
LogID	String	Constant identifier for custom log entries	SK

Targets This paragraph contains descriptions of possible values for the "Target" attribute in the custom logs. As stated in Table 5.1 under "Description" of the "Target" attribute, a target is an object involved in the cobot's activity. This "object" of an action can either be (1) a point in the workspace, (2) a signal sent from one actor to another, (3) a workpiece that a cobot works on, or (4) a position that the cobot moves into. Table 5.2 lists the different targets.

Table 5.2: Different target types used in the collaboration scenarios.

Points	Signals	Workpiece objects	Positions
AL1-front	START	Box-S	Home
AL1-back	GO	Box-M	Pack-position
AL2-left	FIN	Box-L	
AL2-right	READY		
Slide-top			
Slide-btm			
Table-edge			
Chessboard			
Depo-point			
Pattern			

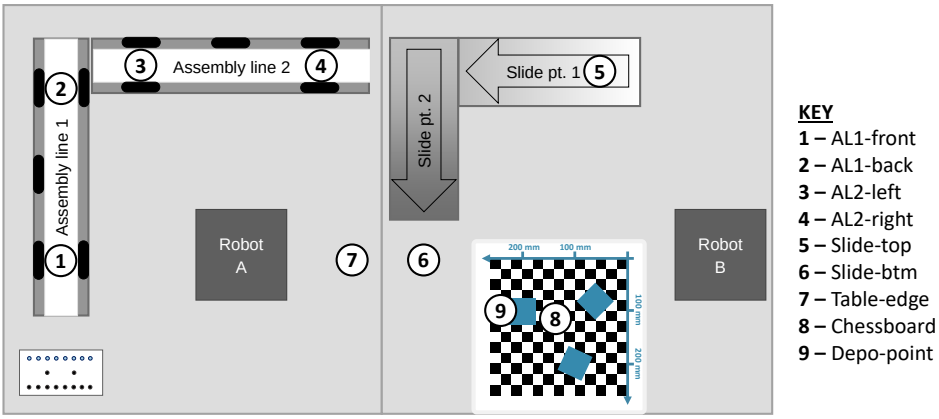


Figure 5.3: Sketch of the cobot cell with mapped points as target types.

Figure 5.3 depicts the setting of the two-cobot cell which the design of the target points was based on. Each X-marked spot on the sketch represents a target point: Assembly Line 1 (AL1) front and back, Assembly Line 2 (AL2) left and right, Slide top and bottom, Table edge, Chessboard, and Deposition point. Additionally, there is a set of points which comprise a "Pattern" (envisioned for Scenario 2, where Robot B paints the box lid). This is a special point-target that is not visible on the cobot-cell sketch because it represents points of movement on a particular workpiece.

Another target that may not be as obvious is a signal. A signal is a digital form of communication that can be exchanged among all three actors. Each signal bears a meaning, or a message intended for the receiving actor. For instance, the "START" signal is sent by the operator to Robot A to initiate

an iteration of a scenario, while the "GO" signal is sent from Robot A to Robot B indicating that Robot B may start its first task.

The real, physical targets that the cobots operate on are the **workpieces**. A predefined set of workpiece types enables easier understanding of the processes and the occurring interactions with respect to the collaborative scenarios. Please note, however, that the workpieces need to be explicitly annotated with "wp:<*workpiece name*>". The prefix "wp:" enables workpieces to be distinguished from other target objects.

Lastly, there is a point defined as "Home", representing the safe position of each cobot in which they start and end their operation, and there is a position (for holding a box by one cobot) designed specifically for packing (performed by the other cobot): this is the "Pack-position".

With all the necessary attributes defined, what is left is outputting them into the *log_history.txt* file. Details on how this is accomplished are given in the following section.

Writing Custom Outputs to UR Log File

The relevant attributes for process discovery and pattern recognition can be written to the log file in form of a string using the URScript programming language for Universal Robots cobots. In particular, the `textmsg` function (see Figure 5.4) can be utilized to produce a custom log entry within a cobot program and write it to the *log_history.txt* file.

textmsg(*s1, s2=''*)

Send text message to log

Send message with s1 and s2 concatenated to be shown on the GUI log-tab

Parameters

`s1`: message string, variables of other types (int, bool poses etc.) can also be sent

`s2`: message string, variables of other types (int, bool poses etc.) can also be sent

Example command: `textmsg("value=", 3)`

- Example Parameters:
  - s1 set first part of message to "value="
  - s2 set second part of message to 3
    - message in the log is "value=3"

Figure 5.4: Documentation of the URScript `textmsg` function

As the output string has to contain all attributes defined in the previous section, it needs to be structured such that the individual attributes can be easily extracted. Thus, the format of the output string is `"LogID,ActorID,Activity,Target,EventID,Lifecycle,CaseID"`[1]. The comma separator allows for easy string-splitting and extracting of the attributes in the cleaning

[1] Note that the timestamps are generated automatically when the runtime execution component of the cobot writes the log entry, so it does not need to be contained in the custom output string.

and preprocessing phase. Please refer to Item 4 in Section 7.2.3 for concrete usage of the `textmsg` function in the implementation of the scenarios.

5.2 Preparing the Data

As mentioned before, the original log file of a cobot is a complex, 10-column dataframe with different pieces of information, most of which are logged in numeric format. To make sense of the numbers and utilize the contained information, as well as extract relevant data, certain cleaning and preprocessing steps need to be carried out. These steps are outlined in Figure 5.5 and described in detail in the following sections.

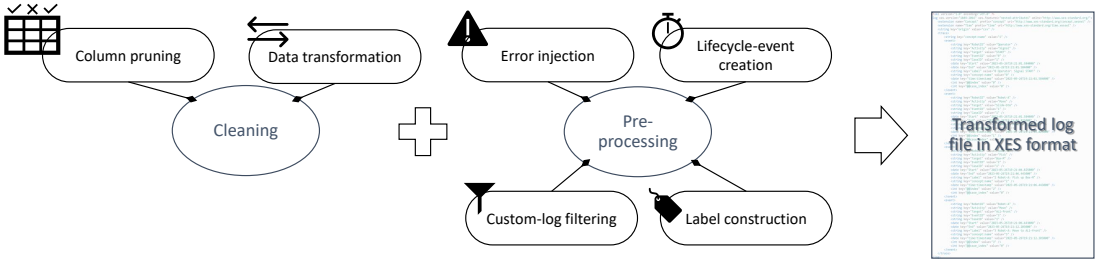


Figure 5.5: Visual summary of the data preparation process.

5.2.1 Cleaning

Cleaning encompasses (1) selection of relevant columns from the original log dataframe (column pruning) and (2) transformation of the contents of the filtered columns to make it more comprehensible and manageable. Out of a total of 10 columns, 7 were chosen as relevant. The 7 relevant columns are the timestamp, the log source, the error code, the additional info code, the log category, and two columns with textual descriptions. The transformation of the data from the selected columns included replacing designated codes of the log sources with their names (e.g., -3 is replaced by "RTMachine"), mapping category codes of the logs to their proper annotations (e.g., 2 means "Warning"), and merging the contents of the two textual columns, so that relevant information is preserved (e.g., "Program started", and "Program X started" are merged into "Program started", and "null" and "<custom output>" are merged into "<custom output>").

5.2.2 Preprocessing

Upon cleaning the original dataframe, further refinements are made on the content, including (1) adding an error case (for demonstrating an alternative event-flow in a scenario), (2) extracting and expanding custom log outputs such that the attributes defined in the string become separate columns in the dataframe, and (3) creating events with a lifecycle using the "start" and "end" timestamps of each event.

The injection of an error case into the logs consists of choosing at random one of three common errors that can occur on a UR cobot and placing it at some point in the process' logs to demonstrate an alternative trace of events. The three common errors are namely:

- **Emergency Stop**. An Emergency Stop occurs when the *e-stop* button on the control unit of the robot is pressed.
- **Safeguard Stop**. The safeguard stop is triggered by a safety hardware set to a configurable digital input.
- **Protective Stop**. A protective stop is caused by the robot software when a certain hardware limit is reached (e.g., joint angles or speed) or when a collision is detected.

* Note that, in this project, regardless of the error type, a generic label "ERROR" is shown in the resulting process models as its purpose is to demonstrate an alternative flow of events. In particular, in case of an error, the affected trace/iteration is aborted and no activities are carried out after the error.

This step is done "artificially" so that any real harm to the cobots or the working environment (from e.g., collisions) is prevented. Then, the custom log outputs are filtered in the data using the LogID described before (string constant "SK"), and expanded into columns, where the column names become the attribute names defined in Table 5.1 and the contents become the values separated by commas in the custom output string. Then, the log entries with the same EventID and CaseID are merged into one so that from the entry where the Lifecycle attribute has a value "start" the Timestamp becomes the value of the "Start" attribute, and the other entry, where the Lifecycle attribute has a value "end" the Timestamp becomes the value of the "End" attribute of the merged log entry. Lastly, the labels (unique full activity annotations) are constructed by combining the EventID, ActorID, and Activity attributes.

5.2.3 Formatting the Dataframe

The described cleaning and preprocessing steps result in a dataframe with all necessary information for further processing. Before moving onto mining collaborative patterns, a final step is formatting the dataframe into an XES-compliant format so that it can be used in the process discovery step later on. This is easily done using the `format_dataframe` function from the pm4py [4] library, which "transforms the event data table into a format that can be used by any process mining algorithm in pm4py. That is, the format_dataframe()-function creates a copy of the input event log, and renames the assigned columns to standardized column names used in pm4py.".

6 Mining Collaborative Patterns

Once the data has been acquired and prepared, further analysis on the cobot logs can be performed. In this chapter, the concept behind finding collaborative patterns in the logged behavior of cobots during program execution is described. Next, the algorithm enabling the identification of the occurring pattern is presented and explained. Lastly, the technical implementation of the introduced concept is elaborated.

6.1 Concept Description

Figure 6.1 presents the ecosystem in which the cobot logs are generated, gathered, transformed and analyzed with the aim to discover behavioral patterns in the cobots' (inter)actions in the shared work environment. The initiator and an important actor in the functioning of the ecosystem is the

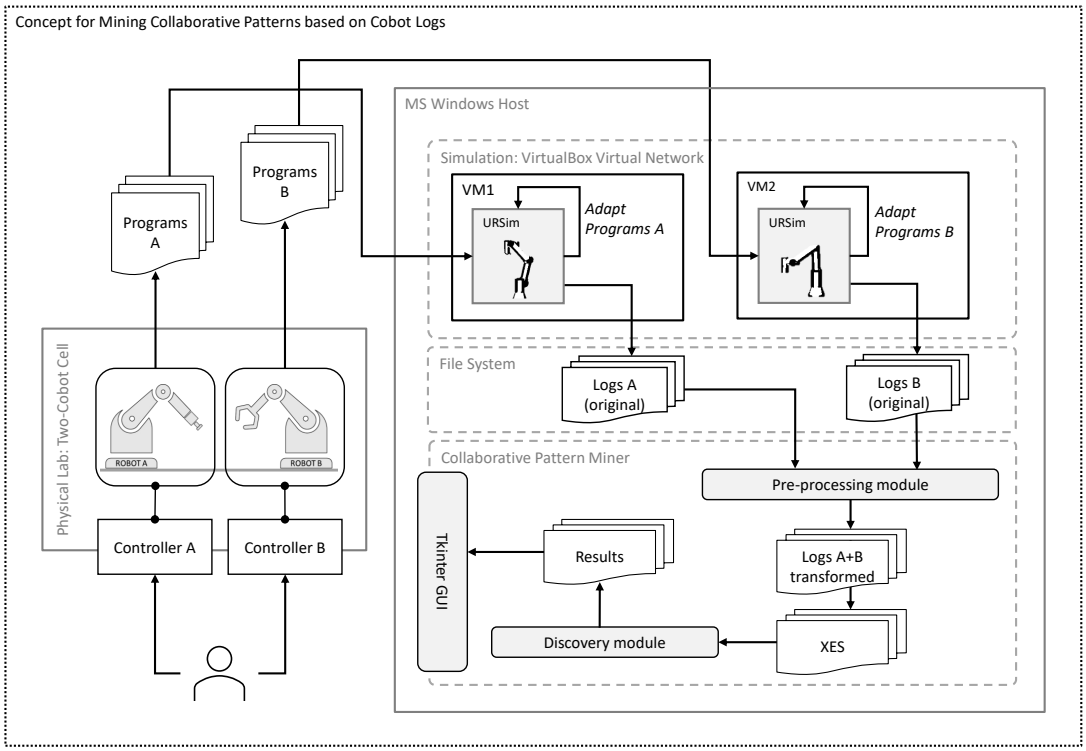


Figure 6.1: Concept for mining collaborative patterns based on cobot logs.

human operator. The ecosystem model shows that an operator uses the control units (*controllers*)

of the cobots as access points: using the controllers, cobot programs can be developed on-site. In contrast to remote program development, this approach enables real-life testing and troubleshooting of the programs, as well as a real "look and feel" of the developed scenarios in the working environment, where other equipment is usually available and its positioning affects the cobots' range of motion. What is more, the positioning of other cobots in the environment can be adequately considered when programming is done on-site. Thus, despite the fact that the remainder of the analysis process is done using a simulated environment, the programs are accurately tailored for the actual, physical work environment.

Once the programs are created and stored locally, they can be exported through the network or via external storage device and transferred to the target system: in this case, a Microsoft Windows computer. On this machine, further called the *host*, digital replicas of the physical robots are created by setting up virtual machines with the Universal Robots' operating system. These virtual machines are URSim [21] instances. On the URSim machines, there is a program providing a graphical interface identical to that of a real cobot controller, called *Polyscope* [22]. Using Polyscope, the programs acquired from the physical cobots can be loaded, edited and executed, just as using a real controller. Polyscope on URSim also provides setup options relevant for executing the scenarios in a simulated environment: establishing network connections. This is a crucial aspect because is enables communication between the cobots using signal exchange over a common network. Specifically, a virtual network is created and both URSim machines are attached to it. Then, inside Polyscope, a MODBUS TCP/IP [7] connection between the cobots is set up. This enables a client-server communication architecture, but with both cobots acting as a server and a client, allowing for a two-sided signal exchange.

What follows is the execution of the programmed scenarios. Each scenario consists of running programs on both robots. This means that to execute a certain scenario, two cobot programs (developed for that scenario) have to be running simultaneously. However, the starting of the two programs needs to only be done once per scenario. An arbitrary number of iterations can be performed while the programs are running, since the iterations are executed upon receiving the operator's start signal. Another option (especially useful in the simulated environment with no physical equipment) is to utilize a programming loop, which will be executed a defined number of times. Nevertheless, in a real environment this would only be efficient if there is another, automated entity setting up the workpieces and preparing the working environment in such a way that there is no "setup-time" necessary between two iterations. Upon scenario execution, the support files (logs) from both cobots are exported and stored at an accessible location for the Collaborative Pattern Miner application.

The Collaborative Pattern Miner is a software solution comprised of several modules that can be roughly divided into 2 parts: data processing and presentation. The data processing segment encompasses a preprocessing module, an analysis module and an output module, whereas the presentation segment contains modules defining views and components which shape the Graphical User Interface (GUI). The preprocessing module encompasses a set of scripts used for (1) reading and parsing the log files into usable data tables, (2) merging logs from different cobots into a single data table, (3) performing a set of cleaning and transformation functions to produce a complete, XES-compliant dataframe which can be processed by the analysis module. The analysis module, then, uses the preprocessed dataframe as a basis for further process mining operations.

In particular, it produces twofold results. One is the discovered process model representing the unaltered preprocessed dataframe. The second is a collaboration pattern detected by the developed pattern recognition algorithm. The algorithm in question is presented and explained in depth in the next section. The two mentioned results of the analysis module, as well as the intermediate results of the preprocessing module, are presented using a GUI in a step-by-step manner. For more details please refer to the technical implementation description in Section 6.3 as well as the demonstration of the application in Section 7.4.

6.2 Algorithm for Identifying Collaborative Patterns

As mentioned in the previous section, the Collaborative Pattern Miner application includes a discovery module which runs an algorithm for determining the collaborative pattern occurring in the logs of a loaded scenario. The pseudocode of the algorithm is shown in Algorithm 1. Please note that the details on preparing the relevant data (grouping, counting and extracting specific values) is omitted in the listing to reduce complexity and outline the essence of the algorithm. The following three steps are carried out either before or during the analysis but are not explicitly shown in the algorithm for the sake of simplicity.

First, as there is always one logged activity of the human operator per case, namely the start signal, these entries are filtered out for the per-case analysis in the algorithm. Second, the case data is further grouped by the cobot ID (ActorID) so that for each cobot, there is exactly one entry per case, and the following columns (attributes) are created:

- **RobotID**. This corresponds to the ActorID in the original dataframe.
- **StartOfFirstTask**. This is the minimum value of the "Start" column for the cobot.
- **EndOfLastTask**. This is the maximum value of the "End" column for the cobot.
- **Workpiece**. This is the "Target" value of the cobot's log entries that is a workpiece (see Table 5.2).
- **ExchangedSignals**. This is the total number of signals exchanged in the current case.

Third and final, erroneous cases are skipped, as such cases do not contain any activities after an error has occurred and thus cannot be used to determine the occurring pattern.

The steps of the algorithm can be summarized as follows. The input parameter is the preprocessed dataframe df as described in Section 5.2. N is the number of cobots involved (determined beforehand via number of unique cobot IDs). First, the dataframe is grouped by CaseID and stored into the $cases$ object, and an array to store the detected patterns (per case), $patterns$, is initialized. Then, for each $case$ in the $cases$ groups, the following procedure is executed. The total number of exchanged signals in the case is calculated by adding 1 (operator's signal) to the sum of occurring cobot signals (line 6). Then, as mentioned before, the case dataframe is grouped by the ActorID column which is actually the cobot ID since we filtered out the operator-related log entry beforehand. Now, the $case\_by\_robot$ object contains groups which are dataframes per cobot, and each one is aggregated into a single entry holding the cobot's ID, the start timestamp

of its first task, and the end timestamp of its last task. Then, the "Workpiece" column is added to each of the cobot rows. The workpiece value is determined in a separate function, by searching through the "Target" column per cobot ID and finding a workpiece as defined in Table 5.2. The $case\_by\_robot$ dataframe is then sorted by the "StartOfFirstTask" column. Before moving onto checking conditions related to patterns, one more variable is defined: $same\_workpiece$.This boolean variable contains the truth value of the expression

$$Bot_1.Workpiece == Bot_2.Workpiece == Bot_3.Workpiece == \cdots == Bot_N.Workpiece \quad ,$$

that is, whether all cobots operated on the same workpiece. If the number of exchanged signals is

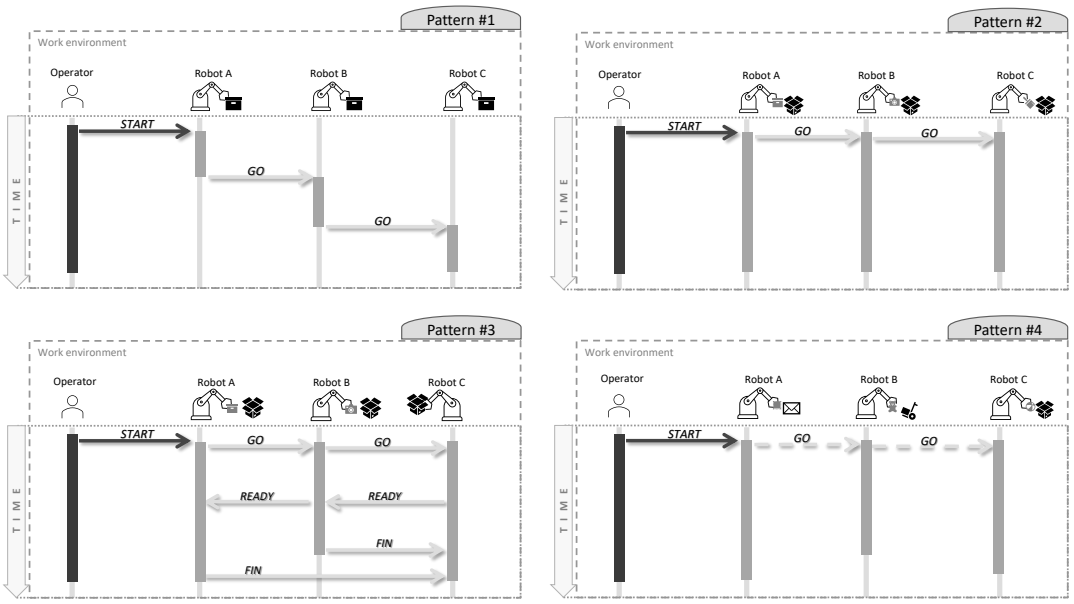


Figure 6.2: Patterns

equal to the number of cobots involved *and* all operated on the same workpiece, the algorithm's next step is to determine whether the activities of the cobots are sequentially ordered. In other words, the goal is to check if for each pair of subsequent cobot entries, the last task of the first cobot ends before the first task of the second cobot has started. The truth value of this expression is stored in the $pairwise\_synced$ array for each cobot pair. After this (line 35), we use the $reduce$ function again to evaluate the following expression into a truth value and store it to the all_synced variable:

$$pairwise\_synced[0] \ \& \ pairwise\_synced[1] \ \& \ pairwise\_synced[2] \ \& \ \cdots \ \& \ pairwise\_synced[N-1]$$

If this expression evaluates to `True`, the *synchronized* pattern is detected. Otherwise (meaning the activities of different cobots are not in successive order), the *cooperation* pattern is recognized.

Moving onto the **else if** block in line 41: if there are more than N exchanged signals in the case *and* the cobots operate on the same workpiece, the *collaboration* pattern is discovered. Every other case that does not fall into one of the first two conditional branches, represents the *coexistence* pattern.

The patterns are illustrated in Figure 6.2. The black objects next to the robots represent target

workpieces, and the arrows represent signals: solid arrows are logged signals, whereas dashed ones (such as in the bottom right example, the coexistence pattern) are not logged. The signals that are not logged are exchanged because of the architecture of the scenarios: as mentioned in the scenario descriptions in Section 4.2, the operator has only one access point to the cobot cell. This means that the start signal needs to be propagated through the network for each iteration. Even in case of the coexistence pattern where the cobots work independently, the goal is to execute as many tasks in parallel as possible to increase productivity. Note that even if these signals were logged, the algorithm would still detect the coexistence pattern due to the fact that every cobot operates on a different workpiece.

Algorithm 1 Pattern mining for the general case with N robots

```
function DISCOVER_PATTERN(df)                          ▷ Discovers a collaborative pattern based on dataframe df
    N...number of robots
    cases := df.groupby("CaseID")                                                          ▷ Group by case
    patterns := []
    for case in cases do
        exchanged_signals_count := case["Activity"].value_counts()["Signal"] + 1
        case_by_robot := case.groupby("ActorID").agg(                                      ▷ Group by cobot ID
                RobotID = ("ActorID", "first"),
                StartOfFirstTask = ("Start", min),
                EndOfLastTask = ("End", max)
        )
        case_by_robot["Workpiece"] := case.groupby("ActorID")["Target"].apply(
                get_workpiece
        )
        case_by_robot := case_by_robot.sort(by = "StartOfFirstTask")                       ▷ Sort the dataframe
        same_workpiece := reduce(                          ▷ Check if workpieces of all cobots are equal
                operator.eq,
                case_by_robot["Workpiece"]
        )
        if exchanged_signals_count == N  &  same_workpiece then
            pairwise_synced := []
            previousBot_EndOfLastTask := None
            currentBot_StartOfFirstTask := None
            for bot in case_by_robot do                                  ▷ Iterate through cobot entries
                currentBot_StartOfFirstTask := bot.StartOfFirstTask
                if previousBot_EndOfLastTask then
                    if currentBot_StartOfFirstTask > previousBot_EndOfLastTask then
                        pairwise_synced.append(True)
                    else
                        pairwise_synced.append(False)
                    end if
                end if
                previousBot_EndOfLastTask := bot.EndOfLastTask
            end for
            all_synced := reduce(operator.and, pairwise_synced)
            if all_synced then
                patterns.append(1)                          ▷ Synchronization pattern discovered
            else
                patterns.append(2)                          ▷ Cooperation pattern discovered
            end if
        else if exchanged_signals_count > N  &  same_workpiece then
            patterns.append(3)                              ▷ Collaboration pattern discovered
        else
            patterns.append(4)                              ▷ Coexistence pattern discovered
        end if
    end for
    return most_frequent(patterns)          ▷ The result pattern is the one that was found in most cases
end function
```

6.3 Technical Implementation

In this section, the technical implementation of the physical and simulated setup is presented, as well as the implementation of the Collaborative Pattern Miner (further referred to as CPM) application.

6.3.1 Physical Cobot Cell Setup (Lab)

The physical cobot cell encompasses all equipment relevant for the realization of collaboration scenarios. The cobot cell is set up in the research laboratory of the university's *Workflow Systems and Technologies* research group. An image of the cobot cell with annotated key components is given in Figure 6.3 and is further described in the sections below.

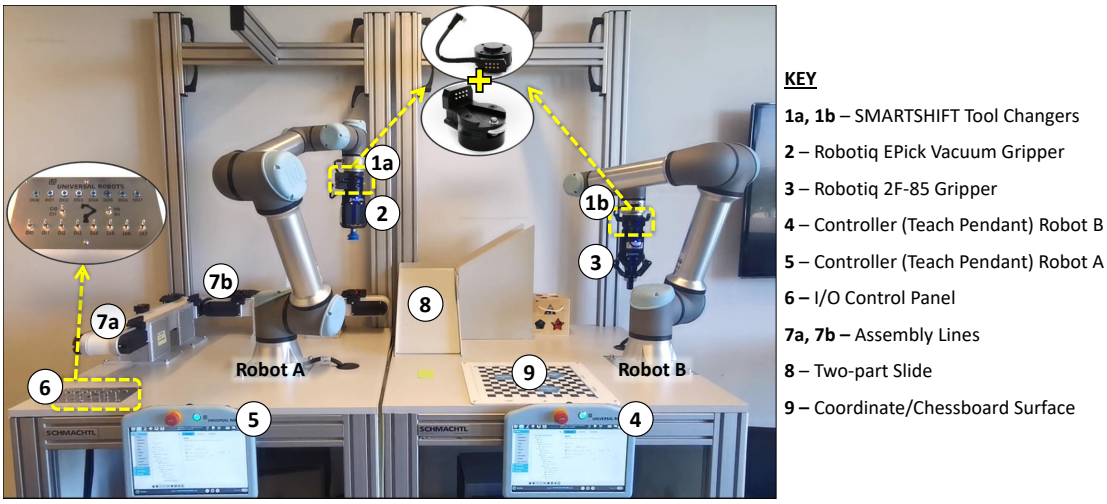


Figure 6.3: Cobot cell setup in the research laboratory.

Equipment

As mentioned throughout this thesis, the cobot that we use is the Universal Robots UR5e model. There are two such cobots installed in the lab. Each of them is mounted on a separate, robust table which is the base operating surface that the other equipment items and workpieces are placed on.

As can be observed in Figure 6.3, the robots have different end-effectors (tools) attached (label numbers 2 and 3). The available end-effectors are so-called grippers, intended for use as carriers to move objects of different sizes to specific points within the reachable range. The grippers installed on our two cobots are namely the *2F-85 (2-Finger Adaptive Gripper)* on Robot A, and the *Epick Vacuum Gripper* on Robot B, both provided by *Robotiq*[1]. These grippers are mounted using *SMARTSHIFT Tool Changers*[2] (labels number 1a and 1b) which enable easier and faster tool

[1] https://robotiq.com/products
[2] https://smartshift-robotics.com/tool-changer/

changing on the robot arms. Each cobot has its so-called *Teach Pendant* (label numbers 4 and 5), which is a touch-screen user interface used to control and interact with the cobots.

On the left-hand side of the image, labeled with number 6 is the *IO Control Panel*, which contains digital input switches and digital output led indicators. This panel is used for sending the "start" signal to the cobots to initiate the beginning of a scenario iteration during a running program. It is important to note that the panel is connected to Robot A's control box, and can thus only send signals to Robot A. Next, labeled with numbers 7a and 7b are two *Assembly Lines*, placed orthogonally to each other. The assembly lines are connected to Robot A's control box and can therefore be controlled via Robot A's Teach Pendant.

On the right-hand side of the image, labeled with number 8 is a rigid, metal *Slide* consisting of two slopes with a flat surface between them. Near the slide, a *Coordinate/Chessboard surface* is mounted (label number 9). Using the black and white squares with sides 10mm long, workpieces that the cobots operate on can be precisely placed (which is necessary since the cobots move with high precision to the programmed points: recall the top-view sketch in Figure 5.3).

Control Software

To be able to use the tools mounted on the UR5e cobots, it is necessary to install the corresponding control software on the Teach Pendants. To do this, we followed the instructions given in the user manuals delivered with the cobot tools. The software program is the same for both of our gripper tools. From the Robotiq support website[3] the latest control software can be downloaded. At the time of installation, the latest version was 1.8.9 (the software file is *Robotiq_Grippers-1.8.9.14887.urcap*) which is the version installed on our robots A and B.

Upon local download, the named .urcap file is transferred to a USB stick that is delivered with the UR robots. This USB stick is then used to install the software on the teach pendant of each robot. To complete this installation, again, the provided instructions were followed, and the following steps were carried out:

1. Plug USB stick into teach pendant
2. Start pendant
3. Go to Settings (right-most icon in the header)
4. Select *System* > *URCaps*
5. Select the "+" button on the bottom
6. Choose the *.urcap* file from USB stick via the file system
7. Select "Open"
8. Select "Restart" to restart the pendant for the changes to take effect

Under the Installation tab (third icon in the header), the respective gripper can be found via the "Gripper" or "Vacuum" menu item under "URCaps". The grippers can also be accessed directly via the URCaps header icon as is shown in Figure 6.4.

[3] https://robotiq.com/support

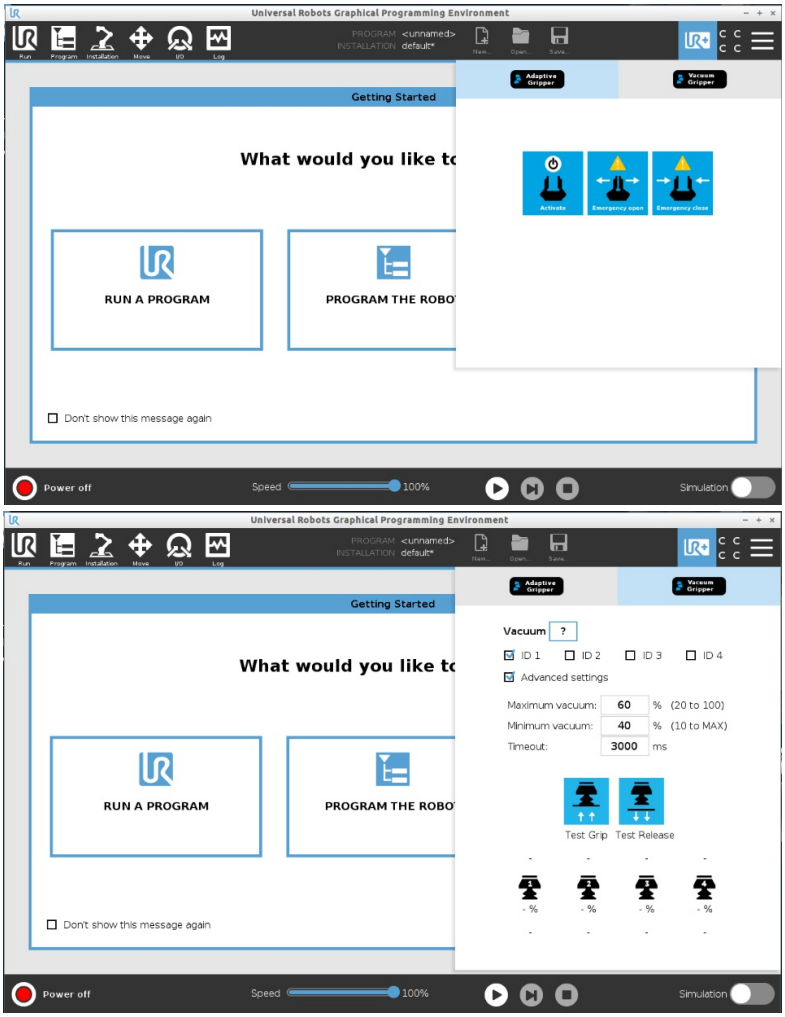


Figure 6.4: Accessing grippers via header icon.

Installation Settings Configuration

To support smooth and safe operation, certain specifications such as the right payload and Tool Center Point (TCP) need to be properly set. Additionally, a MODBUS connection needs to be established between the cobots to enable signal exchanges. The complete description of the installation settings for the given setup is provided in the following.

Payload and Tool Center Point (TCP) Setup For safe operation of the robots, the payload and TCP settings must be defined properly. These settings enable the robot to apply adequate forces in static states, as well as during movement. This is especially important for cases where heavy items are carried. Note that the robot arms have tools installed (grippers), which affect both the payload and the TCP settings.

As we are using tools from Robotiq, we can refer to their documentation [12] for more precise information regarding the relevant measurements. In the referenced file which contains "TCP and Center of Mass of Robotiq Products" data, we can find our gripper models, namely "2F-85" and "EPick (with one suction cup)".

Table 6.1: Relevant specifications of the Robotiq grippers (extracted from documentation [12]).

Gripper model	TCP [mm]			Center of mass [mm]			Mass [g]
	X	Y	Z	X	Y	Z	
2F-85	0.0	0.0	174.0	0.0	0.0	60.0	921
EPick	0.0	0.0	145.0	0.0	0.0	55.2	741

In addition to the Robotiq tools, we use SMARTSHIFT tool changer add-ons which also contribute to the total weight of the payload, as well as the TCP coordinates. The "Technical Instructions Smartshift Tool Change System" report [16] contains the information we need to calculate the total mass, as well as tool and payload centers. As can be seen in Figure 6.5, the main parts of a SMARTSHIFT clutch are the Robot Master (RM) and the Tool Holder (TH). Our tool changers have electrical connectors, so for our case, we refer to components shown in Figure 6.5 D and E.

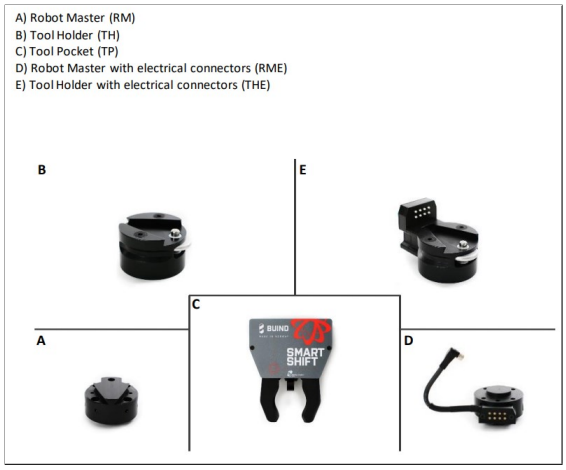


Figure 6.5: Different components of a SMARTSHIFT clutch [16]

To determine the final correct settings for the robots' TCPs and payloads, we need the height and weight of these items. The parts have the properties listed in Table 6.2.

Table 6.2: Specifications of the SMARTSHIFT electrical tool changer components (extracted from user manual [16], p. 24).

Component	Height [mm]	Mass [g]
Robot Master with electrical connectors (RME)	65.0	180
Tool Holder with electrical connectors (THE)		310

Putting it all together, the final TCP and payload settings should be defined as follows:

Robot A, Epick gripper

- TCP:
  Position (X, Y, Z) in mm: (0.0, 0.0, 210.0)
- Payload:
  Center of Mass/Gravity (X, Y, Z) in mm: (0.0, 0.0, 120.2)
  Mass: 1.231 kg

Robot B, 2F-85 gripper

- TCP:
  Position (X, Y, Z) in mm: (0.0, 0.0, 239.0)
- Payload:
  Center of Mass/Gravity (X, Y, Z) in mm: (0.0, 0.0, 125.0)
  Mass: 1.411 kg

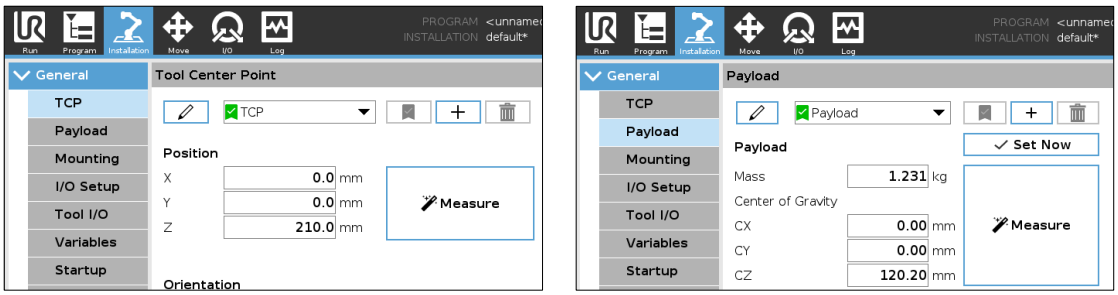


Figure 6.6: TCP (left) and Payload (right) settings of Robot A. The values are analogously set in the *Installation* settings of Robot B.

MODBUS Connection In order for the two cobots to communicate, i.e., exchange signals, they need to be connected over the common network. In this thesis, we use the MODBUS[4] protocol to establish this connection. The MODBUS communication is based on a client-server architecture.

On one cobot's teach pendant, we navigate to the *Installation* tab via the respective header icon. On the left-hand side, under the *Fieldbus* element, we open the *MODBUS* setup and click on the *Add MODBUS Unit* button. The other cobot's IP address (can be viewed in *Settings* > *System* > *Network*, see Figure 6.7) is inserted into the respective field in the MODBUS setup screen. Then, the input/output signals are defined which allow the MODBUS parties to exchange information and signal events (e.g., when it is safe to move).

[4] https://modbus.org/specs.php

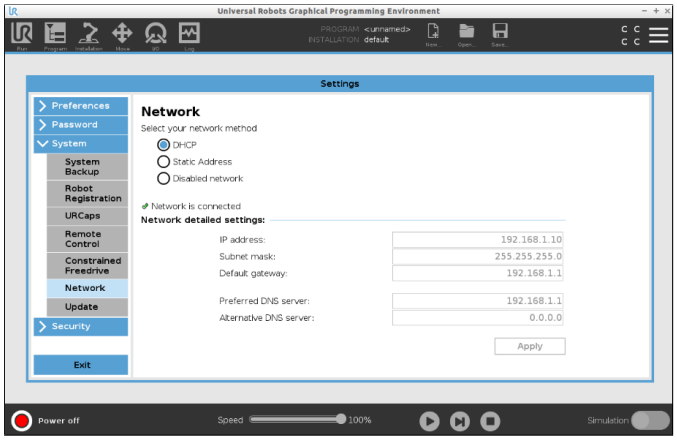


Figure 6.7: Network settings of Robot A

Via one established connection, one party writes to a signal, and the other reads from it. In order to establish a full bidirectional communication, in our case the client-server communication is set up both ways (see Figure 6.8).

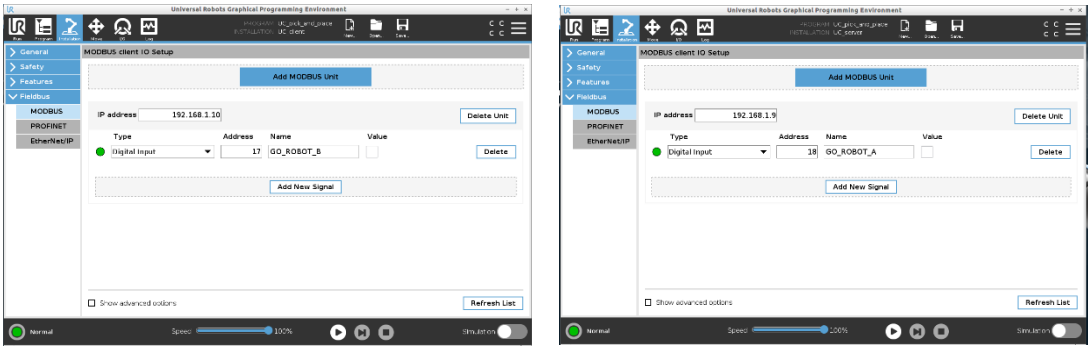


Figure 6.8: MODBUS client setup on both cobot teach pendants

6.3.2 Simulated Cobot Cell Setup (VirtualBox)

The simulated cobot cell is essentially a digital replica of the physical cell in the research lab. This means that the simulated cell also contains two UR5e cobots and that they are connected through a MODBUS connection. In the virtual world, this is realized through virtual machines attached to the same virtual network. To create this virtual environment, the *VirtualBox*[5] virtualization software was used. VirtualBox is available free of charge as open-source software under the GNU General Public License (GPL).

Universal Robots provide a download of the Offline Simulator virtual machine (URSim) which contains a component resembling the physical controller of UR cobots. In particular, it includes

[5] https://www.virtualbox.org/

the control interface which can be run on the virtual machine as a program. The installation of the URSim Virtual Machine (VM) was carried out by following the VirtualBox-specific instructions provided on the download page[6]. Please refer to Table 6.3 for details regarding used software versions.

Table 6.3: Host system and relevant software versions

Component	Specification	Version
Host OS	Microsoft Windows 11	22H2 Build 22621
Virtualization Software	VritualBox	6.1.42
Simulation Software (VM)	Offline Simulator E-Series URSim for Non-Linux	5.11.5
UR Simulator Program	UR5e	5.11.5

After the URSim VM was installed in VirtualBox, some additional configuration was necessary to **(1) optimize display quality**, **(2) enable shared-folder access**, and **(3) set up networking**. As these adjustments are necessary for the virtual machines of both cobots, one virtual machine is set up completely first, and the second is then created as its clone. The mentioned additional configuration steps are thoroughly described in the following.

Display Configuration

After running the VM with the suggested VirtualBox-configuration (concretely, the "VBOXVGA" or "VBOXSVGA" selected as the graphics controller), the robot visualization in the URSim application was flickering, which is very irritating for the eyes after some time working with the simulator. This issue was solved by enabling 3D acceleration in the display/screen tab of the VM settings. When this choice (3D acceleration) is confirmed in the settings, the graphics controller will automatically be changed to "VMSVGA" by the VirtualBox Manager. These changes solved the flickering problem. However, the VMSVGA controller does not seem to play well with the VirtualBox Guest Additions, and as a result, the screen resolution is reduced to 800x600 which cuts off almost 60% of the screen as it is not scaled. There was no possibility to manually change/adapt the screen size in any of the VirtualBox or the VM's interface menus. The solution to this issue was to manually adapt the screen resolution from within the running VM via terminal, using the inbuilt ubuntu *cvt*[7] and *xrandr*[8] utilities. The cvt tool is used to determine a "mode-line" adhering to the CVT standard based on a given h x v resolution, while xrandr is used to set the screen resolution. The exact steps and commands are listed in the following.

1. Run the URSim VM
2. Open a terminal window via *Start > System Tools > UXTerm*
3. Run the following command to see the name of your monitor:

[6] https://www.universal-robots.com/download/software-e-series/simulator-non-linux/offline-simulator-e-series-ursim-for-non-linux-5115/
[7] http://manpages.ubuntu.com/manpages/bionic/man1/cvt.1.html
[8] https://www.x.org/releases/X11R7.5/doc/man/man1/xrandr.1.html

```
$ xrandr
```

The output should look something like this:

In this case, the name is "Virtual1". The name is needed for setting the resolution.

4. Run the cvt command with desired resolution (in this case, 1920x975 seemed to be optimal[9] for the used host):

```
$ cvt 1920 975
```

5. Create and open a new text file (this made copy/pasting easier)

6. Copy the output of the command from step 3 after the **Modeline** keyword, by selecting the text, and then pasting it into the text file by pressing the middle, scroll button on the mouse. This entry should look as follows:

```
"1920x975\_60.00" 155.25 1920 2040 2240 2560 975 978 988 1012 -hsync +vsync
```

7. Edit the text in the file to contain the following:

```
sudo xrandr --newmode "1920x975\_60.00" 155.25 1920 2040 2240 2560 975 978
    988 1012 -hsync +vsync
sudo xrandr --addmode \textit{<monitor\_name>} "1920x975\_60.00"
xrandr --size "1920x975\_60.00"
xrandr --dpi 96
```

8. Save the file for future reference

9. Return to terminal and run the command[10]:

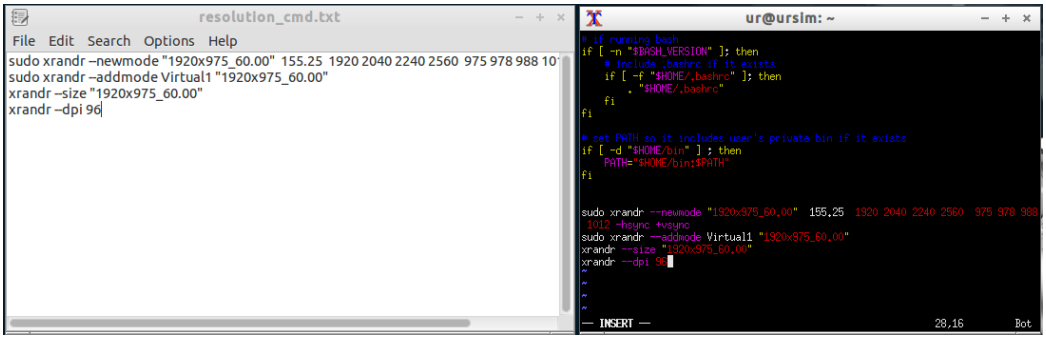
```
$ sudo vim \~/.profile
```

10. Navigate to the end of the file and press "I" on the keyboard to edit the file

11. Copy/paste the four lines of the text file into the *.profile* file. The mentioned files should look as follows:

[9] To determine the optimal resolution for the VM screen, run the VM without the 3D acceleration enabled (and with the VBOXVGA/VBOXSVGA graphics controller set), and run the "xrandr" command without any arguments in the XTerm. The output should be something like "Screen 0: minimum 64 x 64, current 1920 x 975, maximum 32766 x 32766". The optimal resolution is printed after the "current" keyword.

[10] If *vim* is not installed, run **sudo apt-get install vim** first



12. Press “Esc” and type “:wq” to exit editing, save the changes and close the file

13. Finally, run the following to restart the VM:

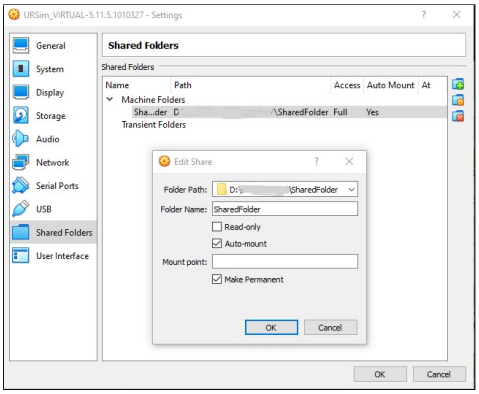
```
$ sudo reboot
```

Shared Folder Setup

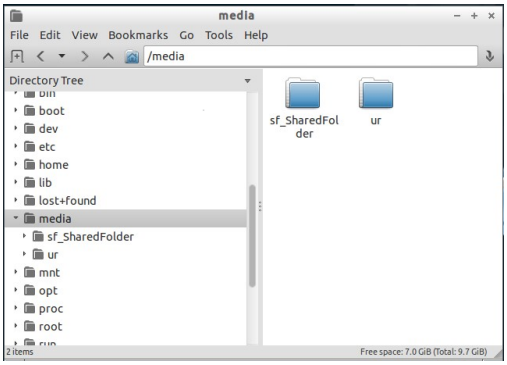
Since the VirtualBox Guest Additions are already installed in the URSim VM, we have access to features such as shared clipboard between the host and guest systems. However, clipboard sharing does not enable copying of entire files (for instance, if we want to back up the robot programs on the host machine). To make file sharing between the two systems possible, we set up a shared folder using the VM settings in VirtualBox as shown in Figure 6.9 (a).

The “Folder Path” is the path to the folder which you want to share on the *host* machine. The name will be derived automatically. It is important to select the “Auto-mount” and “Make Permanent” options. Note that this setup can also be done from the running VM itself by selecting *Devices* > *Shared Folders* > *Shared Folder Settings* in the menu bar.

The configured shared folder will be located in the */media/* directory (see Figure 6.9 (b)). To access the file manager in the URSim VM, navigate to *Start* > *Accessories* > *File Manager PCManFM.*



(a) Shared folder setup in VM’s settings



(b) Shared folder accessed within the VM via file manager

Figure 6.9: Shared folder setup

Note that access will be denied by default to any user other than *root*. Thus, the folder can be accessed either via terminal when switching to the root user, or via file manager after making the following adaptations. In the properties of the *sf_SharedFolder* (see Figure 6.10) there are some access control options, but as *ur* user, we cannot change them.

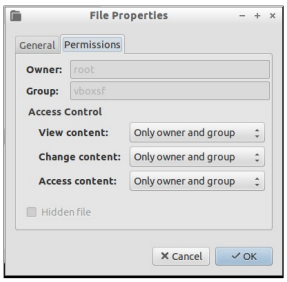


Figure 6.10: Shared folder properties

It is also evident that apart from the owner (*root*), the *vboxsf* user group has all access rights to the folder. Therefore, the easiest modification is to add the *ur* user to the *vboxsf* group via command line:

```
$ sudo adduser ur vboxsf
```

After this, a restart of the machine is needed for the changes to take effect and then the folder and its contents will be accessible. This is a shared folder, so both the host and the guest machine can read, edit, add, or remove files within this folder.

Network Setup

A final configuration step that was necessary for the realization of this use case was the network setup. To be able to establish a closed network between the two VMs (and the host), the "Bridged" network type was chosen, as can be seen in Figure 6.11.

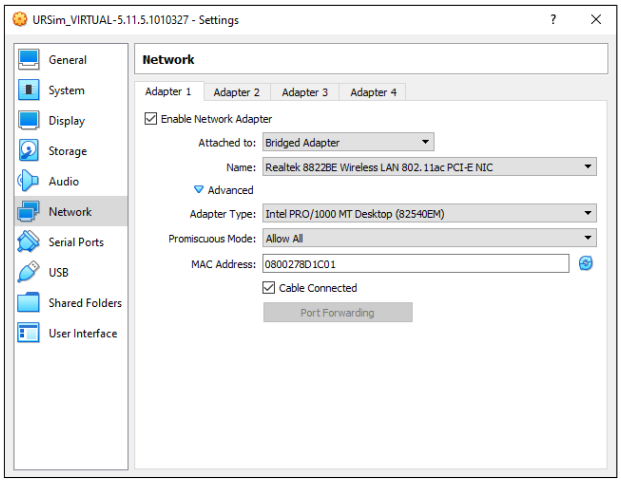


Figure 6.11: Network setup for URSim VM

Creating the Second URSim VM After all the above steps have been performed, everything is set for creating the second VM. The second VM was created by *cloning* the first, fully configured VM. This is done with only a few clicks in VirtualBox; after the target VM is selected, a clone can be created using the *Clone* button in the menu options on the right-hand side of the interface. After entering a name and desired VM folder destination, the clone type needs to be chosen. There are two clone types: a *full* clone, and a *linked* clone. Since we want our machines to be completely independent as are real robot arms, we created a full clone.

The clone will have all the features and configurations that the first VM has, except the different name that was assigned to it, and the MAC address which needs to be newly generated in the network settings tab.

To test the VMs' connectivity to each other via the network, both VMs must be run at the same time. The IP addresses can be discovered via terminal using the *ifconfig* utility, and then pinged to check accessibility.

Establishing a MODBUS Client-Server Connection The goal is to simulate a real MODBUS TCP/IP connection using our bridged network setup and the URSim software. As the control software on URSim is identical to that on a physical teach pendant, setting up the MODBUS connection is done in the same way: please refer to Section 6.3.1 for setup instructions.

6.3.3 Collaborative Pattern Miner (CPM)

The CPM is a simple Python desktop application built using the *tkinter* [11] library to create and manage visual components of the GUI, while the "backend" mainly uses *pandas* [8] and *pm4py* [4] for data processing. A high-level overview of the application architecture is given in Figure 6.12. As mentioned in the concept description in Section 6.1, the CPM has a simple structure with clear separation between the frontend (presentation segment, top half of the illustration) and the backend (data processing segment, bottom half of the illustration). The frontend is a GUI with a single application window showing different views of the log analysis process. The logic in the backend can be roughly divided into 3 distinct modules: (1) preprocessing module, (2) analysis module and (3) output module. In essence, the different modules of the data processing segment are utilized by the presentation segment to perform analysis functions and produce appropriate outputs, depending on the view.

The source code of this project is published on GitLab[11] under the GPLv3 license[12].

GUI: Presentation Layer

The application is built so that the results of major processing steps are presented in the GUI in form of views. There are five different views, whereas the first is the start or main view, where an example use case/scenario can be chosen from four given options. The second view is the *Original Logs View* which showcases the read and parsed unaltered logs of the cobots. The logs are read from text files residing in the *Data* directory, and the reading and parsing functionality is provided

[11] https://gitlab.com/stelakucek/collab-pattern-miner
[12] https://www.gnu.org/licenses/gpl-3.0.en.html

by the preprocessing module. The next view in the GUI, the *Transformed Logs View*, shows the result of the next processing step: the cleaned and transformed cobot logs. Behind the scenes in this step, the logs of all cobots are merged into a single dataframe, relevant data is filtered and the values are transformed into a simpler and more comprehensive format. In this view, the different elements defining process instances such as CaseIDs and EventIDs can be recognized and the details of the activities can be observed in a table. The subsequent view, the *XES View*, shows the transformed logs in XES standard format. The conversion into this format is provided by the analysis module which utilizes the *pm4py* library for formatting the dataframe. When the dataframe is formatted according to the XES standard, it is written to a file inside the *Results* folder. The final view is the *Process View*, which shows both the discovered (BPMN) process model, and the detected collaboration pattern as a final result of processing the cobot logs. As is the case with the XES file, the BPMN process model is also stored into the *Results* folder in both Scalable Vector Graphics (SVG) and Portable Network Graphics (PNG) image formats[13].

Preprocessing Module

This module is responsible for reading, parsing, cleaning and transforming the log data. Naturally, this module is used by the *Original Logs View*, as it loads the original log data from files and preprocesses it into readable data tables, as well as by the *Transformed Logs View*, as it transforms the logs so that they are XES-compliant but also so that they only contain relevant attributes for further analysis. The transformation includes (1) merging the logs from different cobots into a single dataframe, (2) filtering the data so that only the relevant columns (timestamp, source, error code, info code, category, text descriptors) are kept, (3) transforming the content so that it is more comprehensive, e.g., vague alphanumerical codes are translated into nominal values (strings), (4) inserting an error log entry at a random point within the logs, (5) filtering and expanding the custom log entries into attributes (new column values) that are relevant for the subsequent analysis and discovery, (6) adding unique event identifiers, (7) using two entries corresponding to the beginning and end of a single event to construct a single entry with "Start" and "End" timestamp columns, and (8) adding a label column which contains the string to be used as activity annotation in the resulting process models.

In the following listings, and excerpt of the Transformed Logs View class is shown using the preprocessing module to perform the described operations on the log data, as well as an excerpt of the actual preprocessor code containing calls to the respective preprocessing functions.

```
# transformed_logs.py

class TransformedLogsView(BaseView):

    def __init__(self, df_dict: dict):
        super().__init__()
        # merge logs from robots
        dfs = list(df_dict.values())
        self.transformed_logs = preprocessor.merge_dfs(dfs)

        # preprocess merged df
```

[13] For readability and better quality of the process image, the process model is first stored in SVG format. However, due to limitations when displaying images in a *tkinter* window, it was necessary to convert it into PNG.

```
12          self.transformed_logs = preprocessor.preprocess_df(self.transformed_logs)
```

```
# preprocessor.py

    def preprocess_df(df: pd.DataFrame) -> pd.DataFrame:
    """
    Executes functions for preprocessing the dataframe including cleaning,
    transformation,
    injecting error cases, and filtering.

    Args:
        df (pd.DataFrame): Dataframe to preprocess

    Returns:
        pd.DataFrame: Preprocessed dataframe
    """
    # column pruning
    new_df = df[relevant_columns].copy()

    # data transformation to make column content readable & comprehensive
    new_df = transform_df(new_df)

    # inject error cases
    new_df = inject_error_cases(new_df)

    # keep only relevant (custom) logs in df & expand them to attributes (cols)
    new_df = keep_custom_logs(new_df)
    new_df = expand_custom_logs_to_attributes(new_df)

    # generate unique ids for events (activities) based on their order
    add_unique_event_ids(new_df)

    # create lifecycle events from their start and end log entries
    new_df = create_lifecycle_events(new_df)

    # finally, set activity labels of the events
    set_activity_labels(new_df)

    return new_df
```

Analysis Module

The functionality of the analysis module is used by the *XES View* and the *Process View*. For the XES View, this module performs a simple formatting operation to make the transformed log dataframe XES-compliant. In the listings below both the excerpt of the XES View and the called function for retrieving the formatted dataframe are shown. As can be seen in line 21 in the listing of the "discovery.py" script, the **fomat_dataframe** function from the pm4py library is used to make the conversion.

```
# xes_view.py

    class XESView(BaseView):

    def __init__(self, df: pd.DataFrame):
        super().__init__()
        self.log_formatted = get_xes(df)
```

```python
# discovery.py

def get_xes(
    df: pd.DataFrame,
    activity_key: str = XESCols.EVENT_ID.value,
    case_id: str = XESCols.CASE_ID.value,
    timestamp_key: str = TIMESTAMP_END,
) -> pd.DataFrame:
    """
    Formats the given dataframe into a XES standard format using the parameters.

    Args:
        df (pd.DataFrame): Dataframe to format
        activity_key (str): Column name to be used as the activity key (event ID)
        case_id (str): Column name to be used as case/trace ID
        timestamp_key (str): Column name to be used as the relevant timestamp

    Returns:
        pd.DataFrame: The formatted XES dataframe
    """
    xes_formatted = pm4py.format_dataframe(
        df=df, activity_key=activity_key, case_id=case_id, timestamp_key=
    timestamp_key
    )

    return xes_formatted
```

For the Process View, the Analysis Module yields the results in terms of the discovered collaborative pattern based on the XES-compliant dataframe. The following listings show an excerpt of the Process View class calling the **discover_pattern** function, as well as an excerpt from the "pattern_miner.py" script containing the function. The **discover_pattern** function implements the algorithm for recognizing the collaborative pattern as described in Section 6.2.

```python
# process_view.py

    class ProcessView(BaseView):
    def __init__(self, df_xes: pd.DataFrame):
        super().__init__()
        self.result_pattern, self.example_case_df = pattern_miner.discover_pattern
    (
            df_xes
        )
```

```python
# pattern_miner.py

    def discover_pattern(df: pd.DataFrame) -> tuple[str, pd.DataFrame]:
    """
    Discovers the collaborative pattern in a process represented by the given
    dataframe.

    Args:
        df (pd.DataFrame): Dataframe in which the pattern is to be identified

    Returns:
        tuple[str, pd.DataFrame]:
            (‘<the identified collaborative pattern>,
             <an example case with relevant attributes>‘)
    """
    clean_df = remove_operator_event(df)
    number_of_robots = robot_count(clean_df)
```

```
17
18      df_by_cases = clean_df.groupby("case:concept:name")
19      detected_patterns = []
20      case_by_robot_example = None
21      for case_id, case_data in df_by_cases:
22          exchanged_signals_count = 0
23          try:
24              exchanged_signals_count = case_data[XESCols.ACTIVITY.value].
    value_counts()["Signal"]+1
25          except KeyError:
26              pass
27
28          is_err = is_error_case(case_data)
29          if is_err:
30              # if it is an error case, it means the iteration is not complete
31              # therefore we skip it
32              continue
33
34          case_by_robot = case_data.groupby("ActorID").agg(
35              RobotID=("ActorID", "first"),
36              StartOfFirstTask=("Start", min),
37              EndOfLastTask=("End", max)
38          )
39          case_by_robot["Workpiece"] = case_data.groupby("ActorID")["Target"].apply(
40              get_workpiece
41          )
42          case_by_robot["Workpiece"] = case_by_robot["Workpiece"].astype(str)
43          case_by_robot["ExchangedSignals"] = exchanged_signals_count
44          case_by_robot = case_by_robot.sort_values(by="StartOfFirstTask")
45
46          if case_by_robot_example is None:
47              case_by_robot_example = case_by_robot
48          workpieces = case_by_robot["Workpiece"]
49          same_workpiece = all(x == workpieces[0] for x in workpieces)
50
51          if exchanged_signals_count == number_of_robots and same_workpiece:
52              pairwise_synced = []
53              end_prev_last = None
54
55              for botdata in case_by_robot.itertuples():
56                  start_current_first = botdata.StartOfFirstTask
57                  if end_prev_last:
58                      if start_current_first > end_prev_last:
59                          pairwise_synced.append(True)
60                      else:
61                          pairwise_synced.append(False)
62                  end_prev_last = botdata.EndOfLastTask
63
64              all_synced = reduce(operator.and_, pairwise_synced)
65              if all_synced:
66                  detected_patterns.append(1)
67              else:
68                  detected_patterns.append(2)
69          elif exchanged_signals_count > number_of_robots and same_workpiece:
70              detected_patterns.append(3)
71          else:
72              detected_patterns.append(4)
73
74      detected_patterns = [*map(PATTERNS.get, detected_patterns)]
75      result_pattern = most_frequent_element(detected_patterns)
76
77      return result_pattern, case_by_robot_example
```

Additionally, the Analysis Module provides a function for discovering the process model in BPMN format (using the respective pm4py function).

Output Module

This module provides functionality related to different outputs. This includes process model (BPMN) outputs and formatting, textual pattern descriptions and other constants, as well as XES outputs. On one hand, it provides the objects for output to the View components, and on the other it stores the resulting outputs (XES and process image files) to the file system.

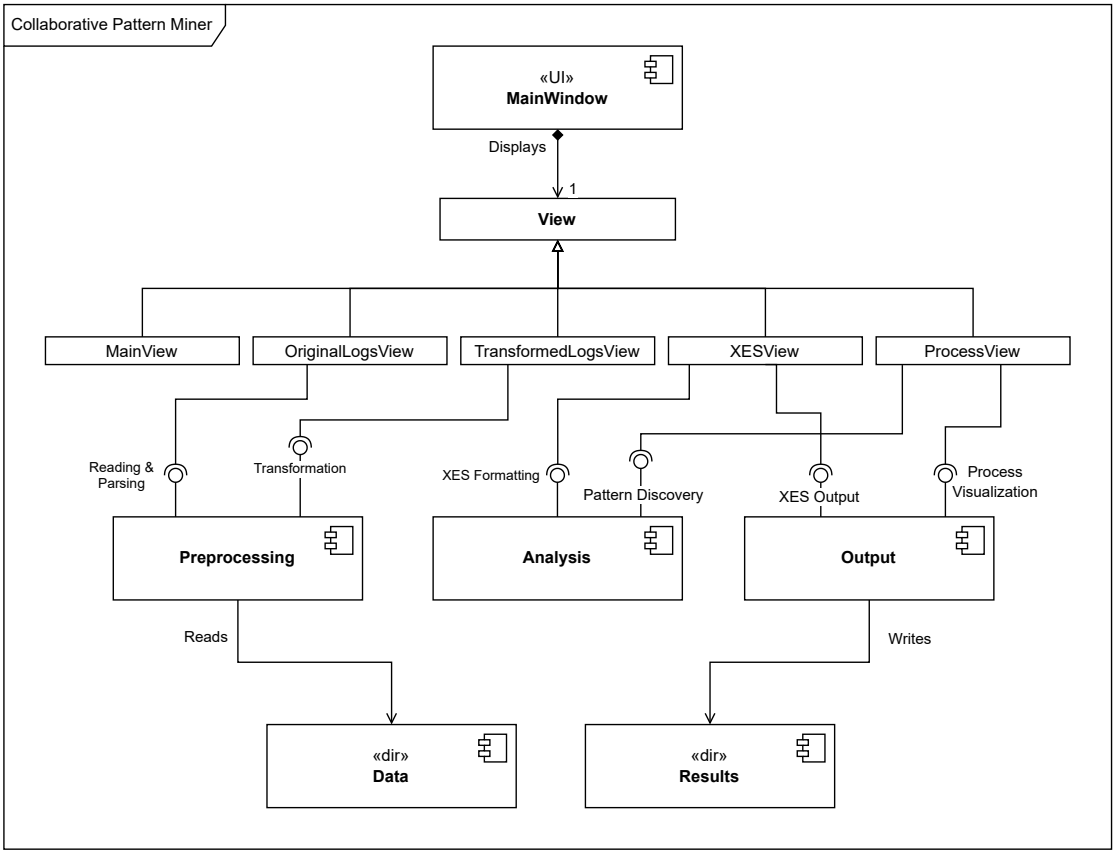


Figure 6.12: Overview of the application structure of the Collaborative Pattern Miner

Additional Features: Log Generator and Log Upload

For the purpose of better evaluation of the implemented collaborative pattern mining approach, two additional aiding features have been developed: (1) a simple CLI (Command Line Interface) Python script for generating logs that comply with the format used in the CPM application and exporting them to a CSV file, and (2) the option to upload such a CSV-based log file (this feature is accessible directly from the Main View of the CPM GUI).

Custom Log File Requirements The requirements for the custom CSV files with the process data are as follows.

- The custom log file is a set of rows with comma-separated values.
- The first row contains the column (attribute) names, which exactly match the following strings, in the herein defined order:
  1. ActorID
  2. Activity
  3. Target
  4. EventID
  5. CaseID
  6. Start
  7. End
- For the semantics and format of the given columns please refer to the specifications given in Table 5.1 on page 25. Note that the "Start" and "End" columns are in fact "Timestamp" attributes and simply denote the beginning and end of an event.
- The remaining rows represent separate events.
- A target which is a workpiece is denoted with a "wp:" prefix.
- If the actor is a cobot, it is denoted with a "Robot-" prefix.
- An example custom CSV log file with 2 traces would look as follows:

ActorID	Activity	Target	EventID	CaseID	Start	End
Operator	Signal	START	0	1	2023-12-21 18:25:35.897	2023-12-21 18:25:35.897
Robot-John	Place	wp:Box	1	1	2023-12-21 18:25:38.081	2023-12-21 18:25:43.901
Robot-John	Signal	GO	2	1	2023-12-21 18:25:49.611	2023-12-21 18:25:53.907
Robot-Steve	Pick	wp:Box	1	1	2023-12-21 18:25:58.131	2023-12-21 18:26:02.947
Robot-Steve	Signal	GO	2	1	2023-12-21 18:26:07.747	2023-12-21 18:26:14.409
Robot-Bob	Check	wp:Box	1	1	2023-12-21 18:26:21.857	2023-12-21 18:26:24.199
Robot-John	Paint	wp:Box	3	1	2023-12-21 18:26:21.857	2023-12-21 18:26:24.199
Operator	Signal	START	0	2	2023-12-21 18:26:38.861	2023-12-21 18:26:38.861
Robot-John	Place	wp:Box	1	2	2023-12-21 18:26:47.985	2023-12-21 18:26:53.289
Robot-John	Signal	GO	2	2	2023-12-21 18:26:56.727	2023-12-21 18:26:59.439
Robot-Steve	Pick	wp:Box	1	2	2023-12-21 18:27:02.673	2023-12-21 18:27:06.539
Robot-Steve	Signal	GO	2	2	2023-12-21 18:27:12.137	2023-12-21 18:27:20.819
Robot-John	Paint	wp:Box	3	2	2023-12-21 18:27:30.677	2023-12-21 18:27:35.317
Robot-Bob	Check	wp:Box	1	2	2023-12-21 18:27:30.677	2023-12-21 18:27:35.317

- Parallel activities are recognized under the following conditions:
  1. If the key timestamps of the activities are equal, the ordering in the log file is taken into account: if the activities appear in different order in different iterations, the activities are considered parallel. Otherwise, the activities will appear in sequential order (the order in which they appear in the log file) in the resulting process model.

2. If the key timestamps of the activities are different, but their chronological ordering differs in different iterations, the activities are considered parallel. Otherwise, the activities will appear in sequential order (the chronological order of their respective key timestamps) in the resulting process model.

Notice how, in the above listing, in the second iteration the last two events (those of robots John and Bob, lines 14 and 15) are in reversed order compared to the first iteration (lines 7 and 8), while their key timestamps ("End") are the same. This will result in these two activities to be recognized as parallel, and will thus appear as such in the resulting process model.

- Error events have the "ERROR" keyword in the Activity column, a single space string " " in the Target column, and an "E<X>" string in the EventID column, where X is the numeric event ID of the event that *should have* occurred at this point in the process. As an error is an instant event, the start and end timestamps should be equal. However, since in the process discovery algorithm only one timestamp is taken into account as the key, this is not strictly required.

  For instance, considering the above example log file, the event in row 13 could be transformed into an error event as demonstrated in the following.

  Before:

13	Robot-Steve	Signal	GO	2	2	2023-12-21 18:27:12.137	2023-12-21 18:27:20.819

  After:

13	Robot-Steve	ERROR		E2	2	2023-12-21 18:27:12.137	2023-12-21 18:27:12.137

Log Generator The log generator is a simple Python command-line tool which generates a CSV file conforming to the requirements described above based on the provided user input.

The user is prompted to enter the events of a target process in the desired order and connect them using an arrow symbol `->` if the activities are sequential, or placing them inside square brackets separated by a comma if the activities are executed in parallel `[Activity_1,Activity_2]`. Each event is to be entered in the format `Bot_name Action Target`. If the target is a workpiece, a prefix of the form `wp:` is expected. The prompt also describes these rules and shows a valid input example.

Upon entering the process, the user is prompted to enter the desired number of traces (iterations) of the process that are to be output to the CSV file. A minimum number of 10 traces is recommended. However, depending on the number of the participating actors, this number may be lower/higher.

After the required input is entered, the script checks the input using a regular expression (RegEx) match function, and if no mismatches arise, generates the CSV file and outputs the destination file name. If there is a mismatch when checking against the required RegEx, the program outputs an error message and prompts the user to provide the input anew.

In the next chapter, the presented concept for mining collaborative patterns in cobot logs is evaluated based on implementations of the scenarios described in Chapter 4.

7 Evaluation

In this chapter, the evaluation of the CPM is carried out based on the collaboration scenarios described in Section 4.2. The evaluation procedure is described in the following section.

7.1 Evaluation Methodology

The concept for identifying collaborative patterns in cobot logs was evaluated by implementing the designed scenarios on the physical, and the virtual cobots, exporting and preparing the log data for analysis, running the developed CPM application with that data, and finally, inspecting the results. A detailed list of the evaluation steps performed is given in the following.

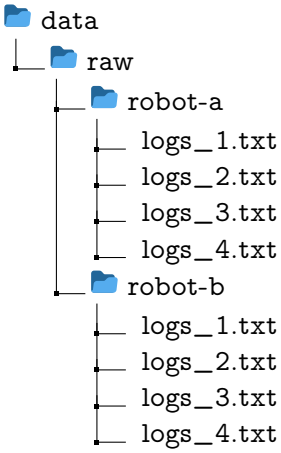
1. Implement the designed scenarios on the physical cobots.
2. Transfer the implemented robot programs from the physical to the simulated cobot cell.
3. Adapt the programs on both virtual cobots so that (1) they can be executed in the simulated environment, (2) multiple iterations can be executed automatically without requiring manual input, and (3) the exported logs will contain custom log entries reflecting the activities of cobots during program execution.
4. Execute the programs of all four scenarios on both cobots, one by one, whereby the programs for the same scenario have to be executed on both cobots at the same time.
5. For each of the scenarios, let the programs execute all programmed iterations to completion before stopping the programs.
6. Export the support files from both cobots and save it to the default folder on URSim.
7. Move the support files on both cobots from the default folder to the respective shared folder.
8. Access the files locally (on the host machine) in the shared folders and copy them to a desired target folder.
9. Open/extract the archived files.
10. Find the *log_history.txt* file inside *ur.../robot_configuration.../files/* and copy it to the *data/raw/robot-X* folder of the CPM project, where X is the cobot identifier ("a" or "b").
11. For each cobot log file, open the *log_history.txt* in a text editor and carry out the following.

a) For each scenario X, where X is a scenario identifier from 1 to 4, find the logs belonging to the execution time frame of the scenario by locating the timestamps of the program start and the program end logs.

b) Select all log entries from (including) the program start log and (including) the program end log and copy them.

c) Create a new text file named *logs_X.txt* and paste the copied logs of scenario X.

12. The resulting file structure inside the *data* folder:



13. Run the CPM application.

14. Choose each of the four scenarios from the main page and follow the instructions on the GUI until the last view containing the analysis results.

15. Inspect the results and compare resulting processes to their respective designs.

16. Reason about the findings and draw conclusions.

In the following sections, the central activities of the evaluation process are presented, namely the implementation of the designed scenarios in form of cobot programs (Section 7.2), scenario log export and preparation for analysis in the CPM (Section 7.3), running the CPM with the exported log data (Section 7.4), subsequently inspecting the obtained results (Section 7.5), and finally, summarizing the findings and insights gained as the result of the evaluation process (Section 7.6).

7.2 Scenario Implementation

In this section, the first three steps of the evaluation procedure are elaborated in detail. In particular, the implementation of the four scenarios designed in Section 4.2 is described.

With the scenario implementation, the intent was to resemble the designed processes as closely as possible using robot programs, and to reflect the behavior (activity) of the cobots during program execution in the produced logs using custom log outputs in those programs. Before going into implementation details, please refer to the following section describing the basics of robot programming using the UR interface: Polyscope.

7.2.1 Background: Robot Programming in Polyscope

As mentioned in Section 6.1, Polyscope is the GUI of the UR control software for cobots. Figure 7.1 shows the layout of the interface. It provides different utilities which include, but are not limited to: state and movement control of the cobot, configuration settings, robot programming, input and output signal dashboard, and log viewing. All of the mentioned utilities can be accessed via the interface header section.

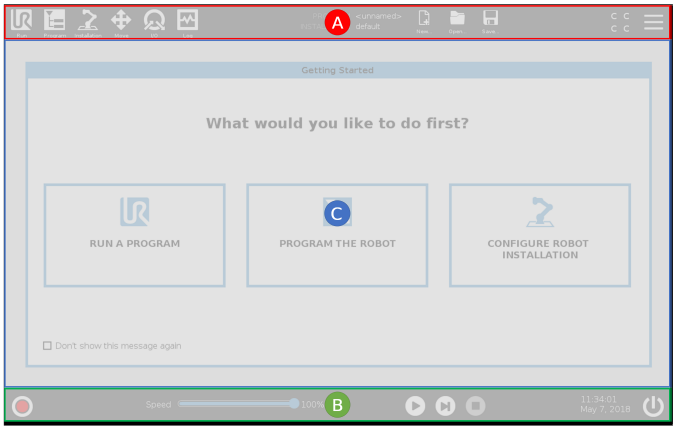


Figure 7.1: PolyScope interface with annotated sections (taken from the UR5e User Manual [19]):
A: Header with icons/tabs that route to different interactive screens
B: Footer with buttons for controlling loaded programs
C: Screen with options to manage and monitor robot actions

For describing the scenario implementation, the most relevant of the mentioned utilities are robot programming and log viewing. These two functionalities are accessible via their annotated header icons, "Program" and "Log", respectively. The *Program* screen is used to build program structures for robot operation control (see Figure 7.2).

Each robot program is basically a sequence of (basic or advanced) commands and is called a Program Tree. The commands used in the program tree are referred to as nodes and are accessible from the Node List in the left-most section of the screen. The Program Tree is shown in the middle section of the Program screen. When a node in the active Program Tree is selected (shaded blue),

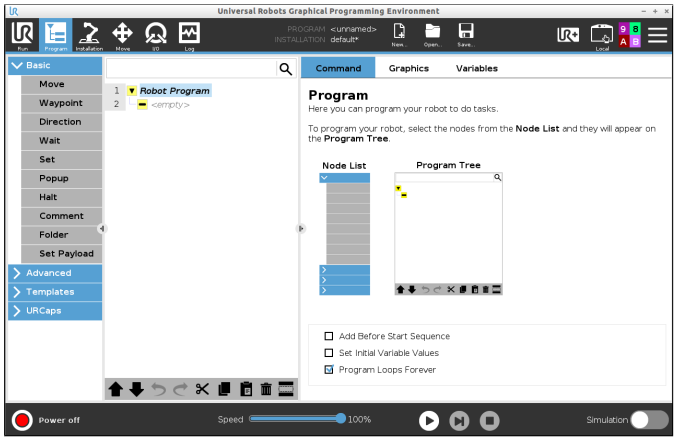


Figure 7.2: *Program* screen in Polyscope

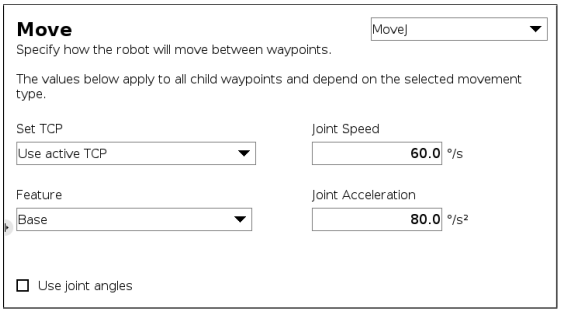
its details can be inspected and adapted (via the Command and Variables tabs) or visualized (via the Graphics tab) in the right-most section of the screen.

The program nodes are basically bundles of commands for the robot to perform a specific action. There are many nodes available to choose from on the Program screen. However, in this project, the most used were the **Move**, **Waypoint**, **Wait** and **Set** nodes from the "Basic" category, and the **Loop**, **Assignment**, **If** and **Script** nodes from the "Advanced" category. Additionally, there are special program nodes for controlling the tools/end effectors mounted on the cobots, which can be found under the "URCaps" category. The relevant URCaps nodes are **Gripper Activate**, **Gripper**, and **Vacuum**. All the mentioned nodes are briefly described in the following.

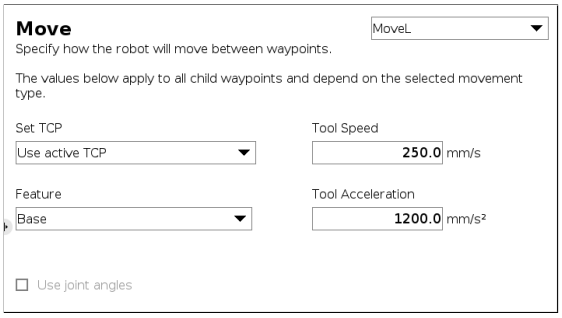
1. Basic program nodes

    1.1. The **Move** node specifies how the robot will move between waypoints, and there are multiple movement types to choose from:

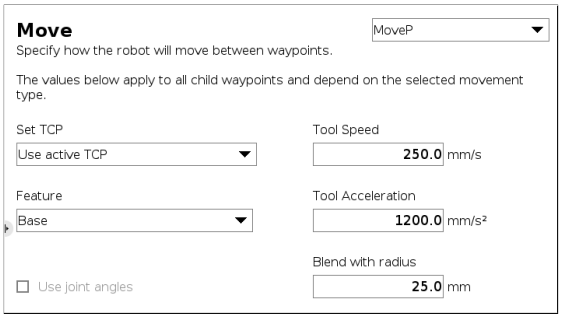
        1.1.1. **MoveJ**, where movements are calculated in the robot arm joint space and joints are controlled to finish their movements simultaneously.



        1.1.2. **MoveL**, where the TCP moves linearly between waypoints and thus each joint performs a more complicated motion to keep the tool on a straight-line path.

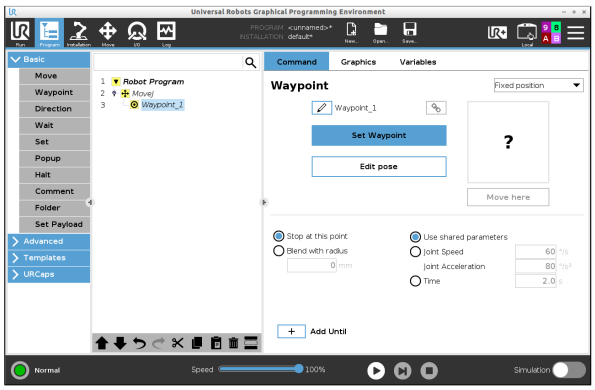


1.1.3. **MoveP**, where the tool is moved linearly with constant speed with circular blends (intended for process operations like gluing or dispensing).

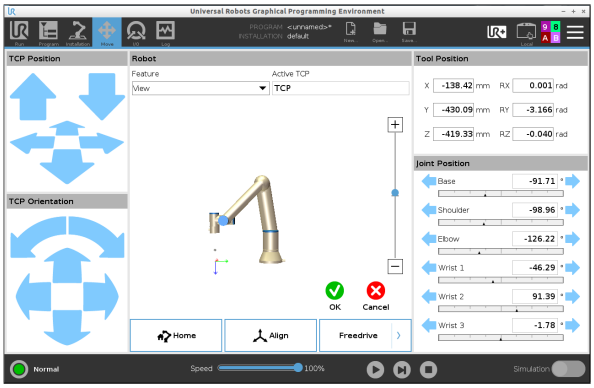


Note that in our scenario implementations, **MoveJ** and **MoveL** movement types were used.

1.2. A **Waypoint** is a point on the robot path; by inserting a Waypoint node into the program, a Move action is automatically inserted:

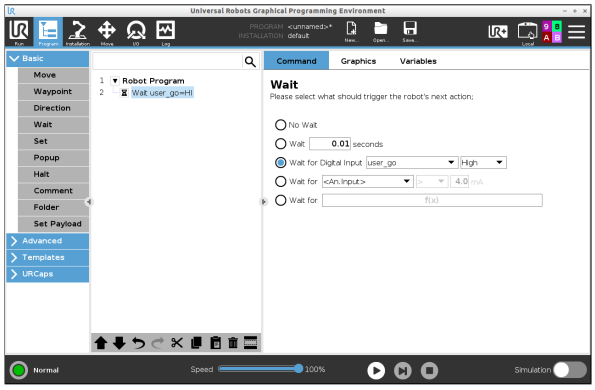


In order to define a waypoint, the "Set Waypoint" button needs to be pressed. This action opens the *Move* screen, where the desired robot pose can be defined:

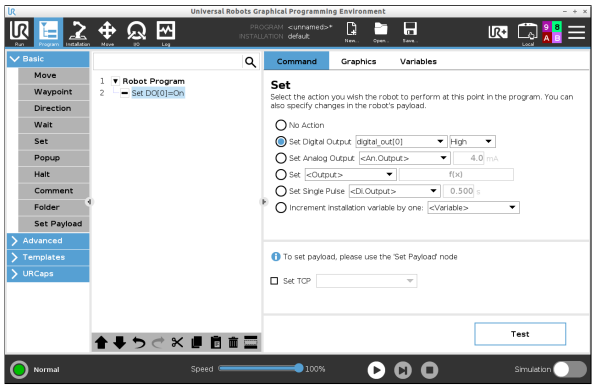


When the desired pose is set, this waypoint can be confirmed with the "OK" button beneath the cobot simulation.

1.3. **Wait** is a node that enables the cobot to wait for a certain amount of time, a specific input value or an event to occur:

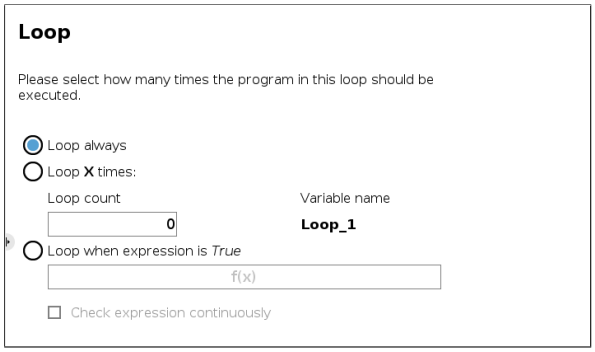


1.4. **Set** is a node enabling different values of outputs (digital as well as analog) to be set, and installation variables to be incremented by one:

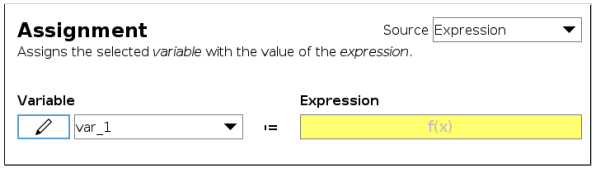


2. Advanced program nodes

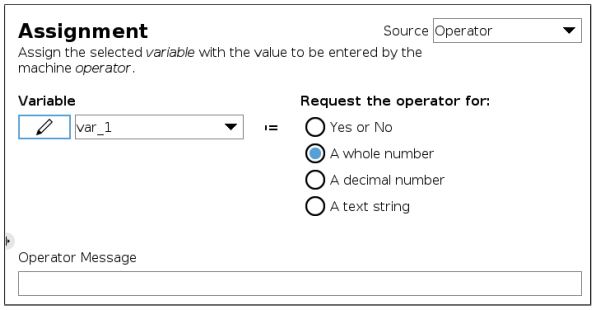
2.1. A **Loop** enables a certain set of program nodes to be executed (1) indefinitely, (2) X times, or (3) when a defined expression evaluates to *True*:



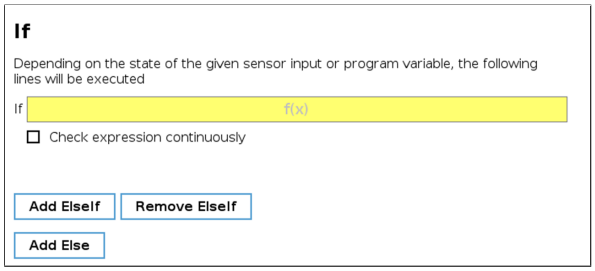
2.2. The **Assignment** node allows for value assignment to a selected or new variable. This value is either (1) the result of the expression defined in the corresponding input field:



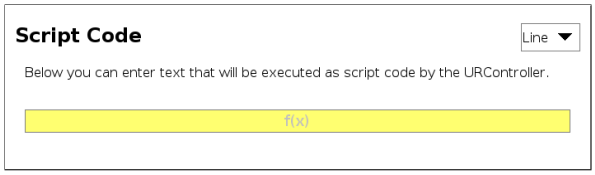
or (2) the input provided by the operator:



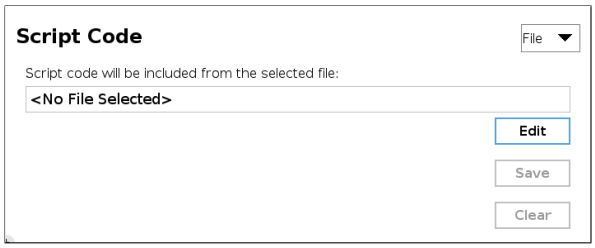
2.3. The **If** node enables conditional execution of nested program nodes depending on the truth value of the evaluated expression defined for the condition:



2.4. The **Script** node provides an entry point for executing URScript code. URScript is a programming language used to control UR robots [18]. Polyscope allows execution of a single line (expression):



as well as execution of whole script files:

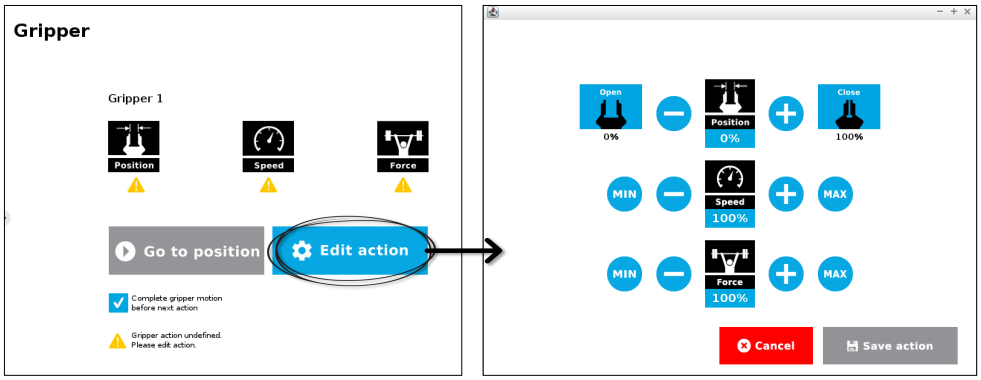


The script files can also be edited in a built-in file editor and saved to the file system.

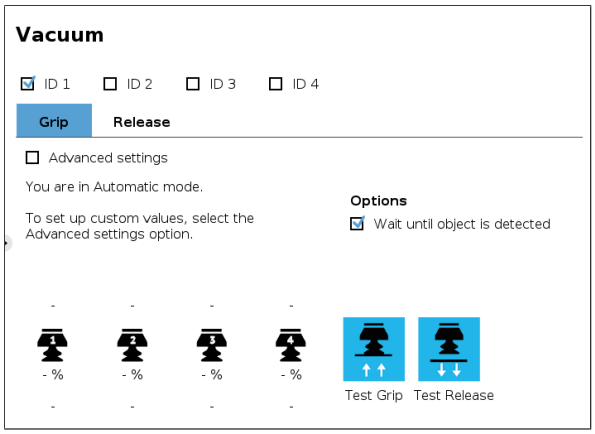
3. URCaps program nodes

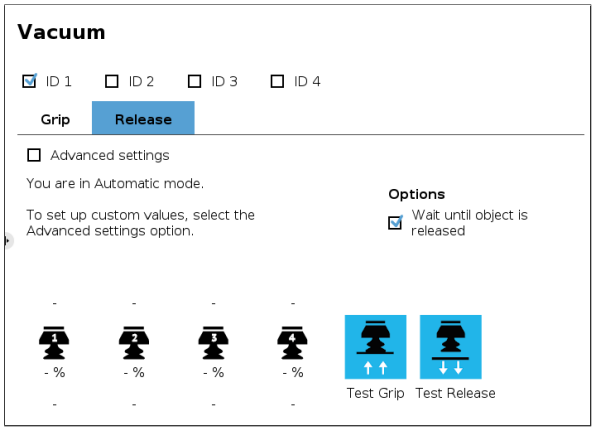
3.1. **Gripper Activate** activates the connected (2F-85) gripper. This node should be executed when first starting a program that uses the gripper after turning on the cobot:

3.2. The **Gripper** node enables controlling the movement of the (2F-85) gripper in terms of (1) percentage of the grip opening (where the two extremes of the action are "Open", 0% and "Close", 100%), (2) the speed and (3) the force of the movement. These controls are accessible by selecting the "Edit action" button:

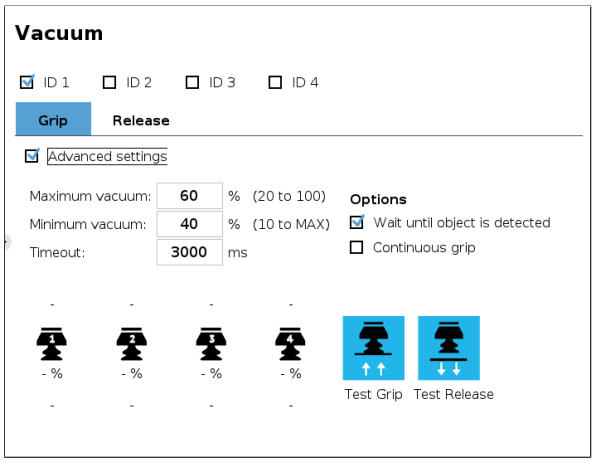


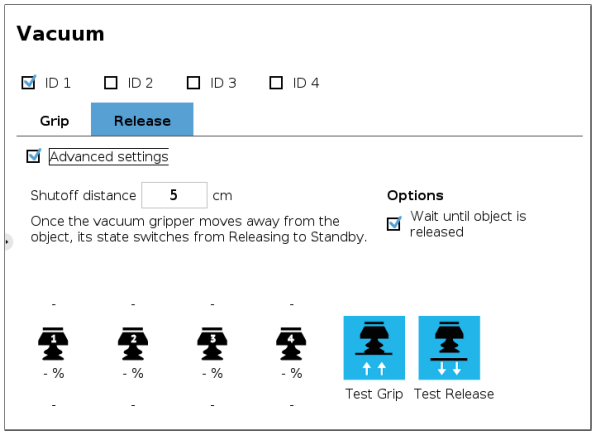
3.3. The **Vacuum** node enables controlling the (EPick) gripper. Both gripping and releasing can be performed either automatically:





or with user-defined settings:

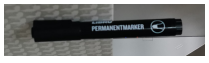



Vacuum
ID 1
ID 2
ID 3
ID 4
Grip
Release
Advanced settings
Shutoff distance
5
cm
Once the vacuum gripper moves away from the object, its state switches from Releasing to Standby.
Options
Wait until object is released
- %
- %
- %
- %
Test Grip
Test Release

7.2.2 Implementation in the Physical Cobot Cell

As listed in the evaluation steps at the beginning of this chapter, the scenarios where first implemented on the physical cobots by creating corresponding robot programs via the control software on the teach pendant (Polyscope). The approach to implementing the scenarios was to use the designed process models and step-wise "translate" the respective activities into UR program nodes. The sections that follow describe this translation for each scenario in more detail. The inventory of used objects, their size, utilization in scenarios, and (if applicable) their workpiece annotation as it appears in the logs, is given in Table 7.1.

Table 7.1: Inventory of the objects used in the collaboration scenarios.

Object	Image	Dimensions W x H x D [mm]	Used in scenario(s)	Workpiece annotation
White box		80 x 115 x 44	1, 2, 4	Box-M
Yellow box		Base 63 x 92 x 25 Lid 66 x 100 x 13	3, 4	Box-S
Box with (blue) business cards		Box 60 x 95 x 35 Cards 52 x 89	3	-
Grey Lego wall		240 x 30 x 125	1, 4	-
IKEA box		330 x 230 x 150	2	Box-L
Black marker		140 x 15 x 15	2	-

Commonalities in the Scenarios

Recalling the scenario designs in Section 4.2, note that there are some activities that occur in every scenario. In particular, the manual activities of the Human Operator, "Prepare production line" and "Initiate production start" are carried out at the beginning of every scenario. Naturally, preparing the production line or, in other words, "setting the scene" for each scenario is a physical activity which does not leave a digital footprint that would be visible in the logs. On the other hand, initiating the production start is an action which involves sending a signal to a cobot (Robot A): this is the "START" signal, as defined in Section 5.1.3. This action is presentable from within the robot programs, and thus in the generated cobot logs as well. Initiating the start of a new scenario iteration is carried out by a digital signal sent by the operator to Robot A. To enable this, a digital input signal needs to be accessed (read) from within the cobot program, and when the state of the signal is high (true), the program can proceed with execution of the following actions. To make the access easier, a certain digital input can be renamed via the *Installation* screen (see Figure 7.3), and then read within the robot program inside a *Wait* program node as shown in Figure 7.4. As described in Section 7.2.1, the *Wait* node is a condition guard which allows for the subsequent nodes to be executed only if the defined condition is met. In case of a digital or analog input, the corresponding values or states are monitored continuously until the condition is met.

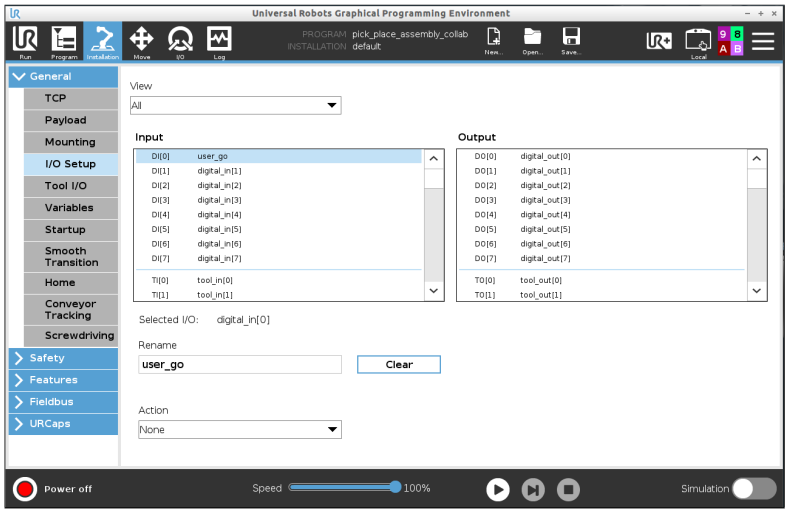


Figure 7.3: I/O Setup on the *Installation* screen.

Moreover, note that in order for only the user signal to be the trigger for the next iteration, the program itself needs to run "endlessly", regardless of reaching the last program node in the program tree. In programming terms, the program should essentially run in an endless loop (until manually stopped), with the *Wait* statement ensuring the conditional execution of the programmed steps that follow. This endless looping of the program is set via the "Command" tab on the right-hand side upon selecting the root node in the program tree called "Robot Program". When in this view, selecting the "Program Loops Forever" option (see Figure 7.5) enables the desired behavior.

One of the central aspects in the collaboration scenarios presented here is the communication between the cobots, implemented using signal exchange via a MODBUS connection (set up as

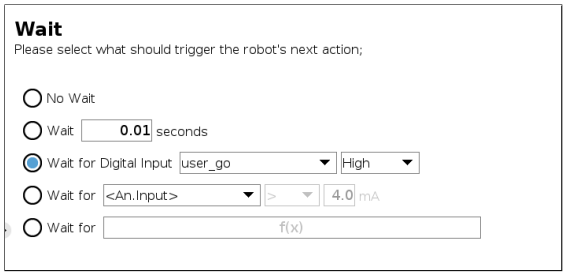


Figure 7.4: Definition of the *Wait* program node for checking the value of the "user_go" digital input.

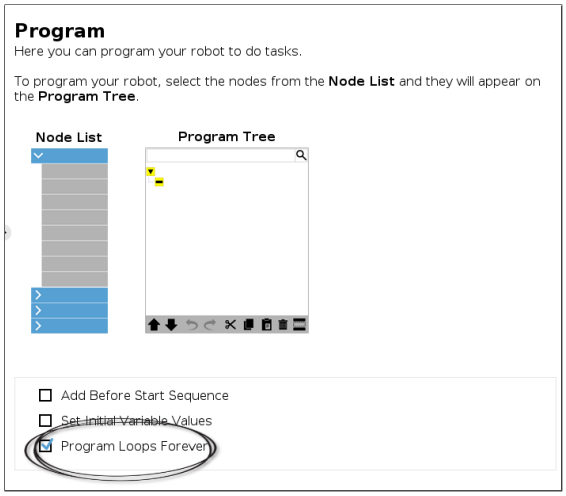


Figure 7.5: Setting the endless program looping in Polyscope.

described in Section 6.3.1). This connection is utilized in the robot programs by writing to and reading from the signal memory bit defined in the respective MODBUS client settings (cf. Figure 6.8). Reading/accessing the signal bit can be done using a program node according to the goal of the action: in the four collaboration scenarios presented here, the most commonly used program node is *Wait*. Writing to the MODBUS destination signal bit is carried out using a URScript function, namely the `write_port_bit(<address>, <value>)` function, where the `<address>` is the address of the signal bit, and the `<value>` is the content to be written to that address. In the cases here, the addresses are 16 and 17 as defined for Robot A and Robot B, respectively (cf. Figure 6.8), while the value is binary ([0, 1], [False, True], [Low, High]).

Last but not least, note that the cobots always start and end their operation in the same position: their respective safe *Home* positions (cf. Figure 6.3).

Other activities, as well as their order differ from scenario to scenario and will be described next.

Successive Pick-and-Place

This scenario, as thoroughly described in Section 4.2.2, represents a cobot collaboration where the cobots perform their activities on a single workpiece in succession. Concretely, Robot A first carries the workpiece to the first assembly line, and places it onto the chessboard after the last sensor on the second assembly line has been activated. Robot B picks up the the workpiece from the designated point on the chessboard and carries it to the top of the slide. This temporal dependency between the execution of the tasks of the two cobots defines the synchronized collaboration pattern. The initial setup and placement of workpiece is shown in Figure 7.6. As can be seen in the figure, the objects used in this scenario are the white box and the grey Lego wall. The white box represents the workpiece that the robots operate on, and its annotation (that will be used in the generated logs) is *Box-M*.

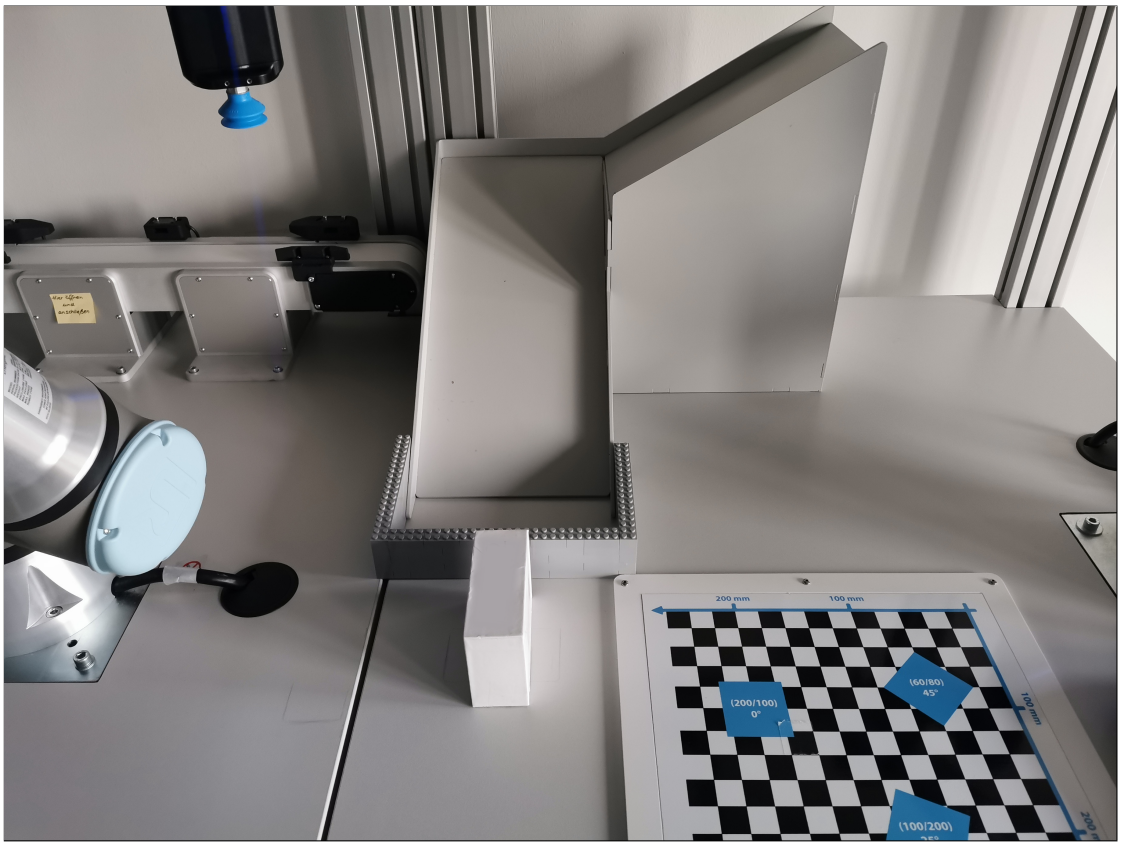


Figure 7.6: Physical cobot cell setup for Scenario 1: Successive Pick-and-Place

As mentioned in the previous section, there are two activities at the beginning of each of the four scenarios, performed by the operator: "Prepare production line" and "Initiate production start". The second one of these activities is the one that can be logged from within the robot program, as the program instructs the cobot to "wait" for the signal of the user, named "user_go", to be activated (set to true) before proceeding (see Figure 7.7). As the excerpt shows, all further

program nodes come after the condition "user_go=HI" (which means the signal is in a **logical high** state) is met.

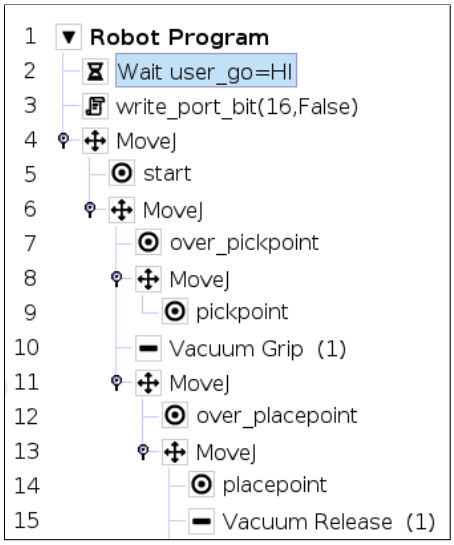


Figure 7.7: Excerpt of the program for Scenario 1 on Robot A.

This scenario involves the usage of the two assembly lines (cf. Section 4.2.2), and the steps of picking and placing the object by Robot A highly depend on the sensors on the assembly lines (see Figure 7.8 left). Note that the naming of the assembly lines here corresponds to their positioning in the cobot cell while front-facing the cell from the point of view shown in Figure 7.6: Assembly Line 1 (AL1) is the *left* AL, while Assembly Line 2 (AL2) is the *right* AL.

The assembly lines are physically connected to the control box of Robot A, and can therefore be controlled via its teach pendant. In particular, the sensors are connected to the "Configurable Input" digital signals, while the lines are controlled via "Configurable Output" digital signals (direction) and "Analog Output" voltage values (speed). These signal values can be inspected and/or set in the *I/O* screen in Polyscope shown in Figure 7.8 right. The figure shows the setup of the configurable input signals corresponding to the six sensors (three on each assembly line). The first two configurable outputs on the top left each represent a switch for the direction of movement of an assembly line. The bottom part of the screen shows controls for the speed of the assembly lines, which can be set with one of two quantities: voltage (unit: V) or current (unit: mA).

In the robot program, the signal **input** values can only be **read**, while **output** values can also be **set**. Concretely, this means that sensor values can be read, while the direction and speed of the assembly lines can also be set. As mentioned before, the sensors are relevant with regard to the movement of Robot A. In particular, they can indicate that an object has been successfully placed on the assembly line or that an object has reached the end of the assembly line and can be picked up for further transfer. The robot program nodes utilizing these two cases are shown in an excerpt of the program for Scenario 1 on Robot A Figure 7.9.

The *If* statement checks whether the front sensor of AL1 has been triggered (which indicates whether an object is detected at its position), and if yes, the AL is started by setting the voltage to

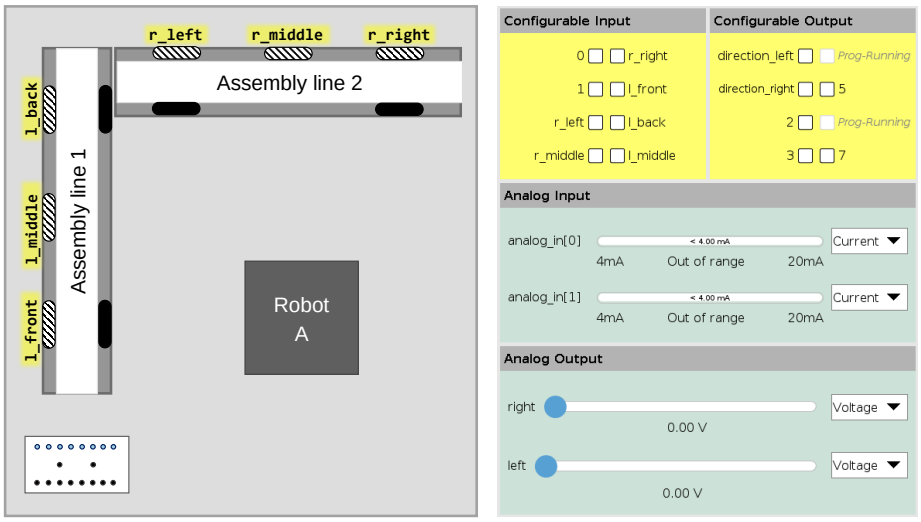


Figure 7.8: Partial sketch of the physical cobot cell layout with assembly line sensor positions highlighted (left), and part of the I/O screen with corresponding configurable digital and analog signal I/Os (right).

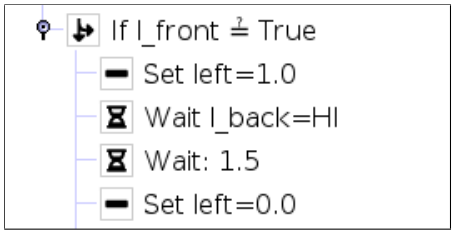


Figure 7.9: Excerpt of the program for Scenario 1 on Robot A showing reading the sensor values of AL1 and controlling its speed accordingly.

1V. Next, the cobot waits for the last sensor on AL1 to be triggered (which means that the object placed at the front has now almost reached the end of the AL), then waits for another 1.5 seconds for the middle of the object to be in the right position for pickup, and finally stops the AL. Note that this sequence is repeated for the second assembly line.

The movements of Robot A between picking and placing the workpiece are implemented using the *MoveJ* program node, and the waypoints are defined by carefully changing the cobot's position in "freedrive" mode, and then performing smaller adjustments using the controls available in the *Move* screen (cf. Item 1.2.) when necessary.

In this scenario, the cobots perform their tasks successively. This means that Robot B starts its first task after Robot A has completed its last task. This is where a signal exchange (as described in Section 7.2.2) is necessary for Robot A to communicate that Robot B can begin its operation. The implementation of this exchange is shown in Figure 7.10. Note that the program on Robot B needs to be running and the MODBUS connection set for the signal exchange to take place. Notice that Robot A essentially sends a "pulse" signal by writing "True" and 3 seconds later "False"

to the signal. This ensures that the signal's value stays "True" long enough for Robot B to read that value (even if there is a short disruption in the connection), but is then reset to "False" to ensure that Robot B does not start another iteration that was not initiated by the user. Note that this signal reading and writing mechanism is carried out in the described way for every signal exchange occurring in any of the scenarios.

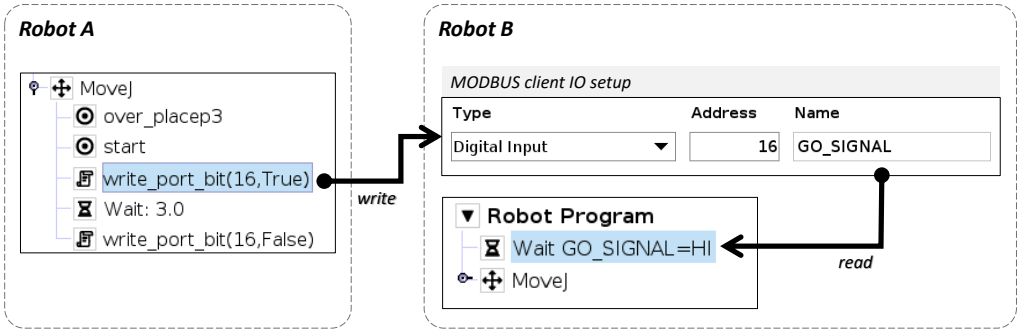


Figure 7.10: Writing to (Robot A) and reading from (Robot B) the *GO_SIGNAL* at address 16 in Scenario 1.

Pack-and-Paint

In this scenario, the cobots cooperate by each performing its own tasks on the same workpiece simultaneously. In particular, Robot A packs an item (in this case, the white box) into *Box-L*, while Robot B draws a pattern on its lid with a marker. Note that in this use case, the workpiece is *Box-L*, while the white box (*Box-M*) is considered a utility item for demonstrating the use case, but could be any other item of approximately the same size and shape that should be "packed" into the bigger box. In the same way Robot B uses the black marker as a utility item to perform its task: "paint" the lid of *Box-L*. The setup of the physical cobot cell for this scenario is shown in Figure 7.11.

To enable operation that is close to parallel in execution, the cobots need to communicate at the beginning of each iteration to start their tasks at (almost) the same time. This is done, again, as described in the previous section, by Robot A writing to the port bit set up in the MODBUS client settings, and Robot B reading it. Again, on Robot B's side, a *Wait* program node is used to ensure that Robot B only starts with its tasks after receiving the "GO" signal from Robot A, which in turn starts its tasks after sending the signal to Robot B. This results in simultaneous execution of the cobot tasks.

Furthermore, the drawing of a pattern on the lid of *Box-L* is implemented using the *MoveL* type of movement, which represents linear movement of the cobot's tool (cf. Item 1.1.2.). The program excerpt, as well as the waypoints of this linear movement pattern annotated in the cobot simulation are depicted in Figure 7.12.

Figure 7.11: Physical cobot cell setup for Scenario 2: Pack-and-Paint

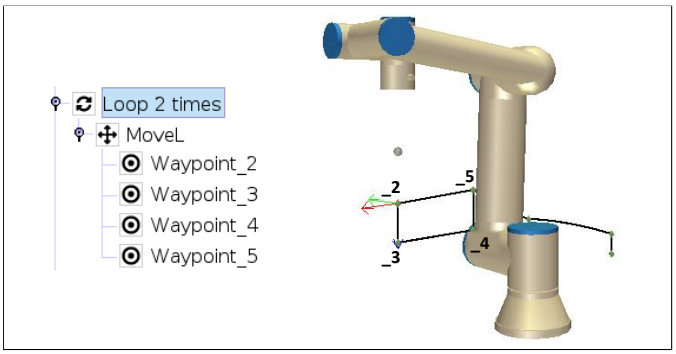


Figure 7.12: Linear movement pattern of Robot B when drawing on the box lid.

Assisted Packing

This use case represents the highest level of interaction and thus collaboration between the cobots. The actions encompass Robot A placing business cards into *Box-S* (the small yellow box) while Robot B is holding it in a easily accessible position for Robot A. The higher level of interaction is demonstrated by more exchanged signals. In particular, there are four signals in this scenario: (1) the user start signal, (2) the "GO" signal from Robot A to Robot B, (3) the "READY" signal from ROBOT B to Robot A once in position for packing, holding the box, and (4) the "FIN" signal from Robot A to Robot B indicating the completion of packing the business cards, which signals Robot B it can place *Box-S* at the final programmed position.

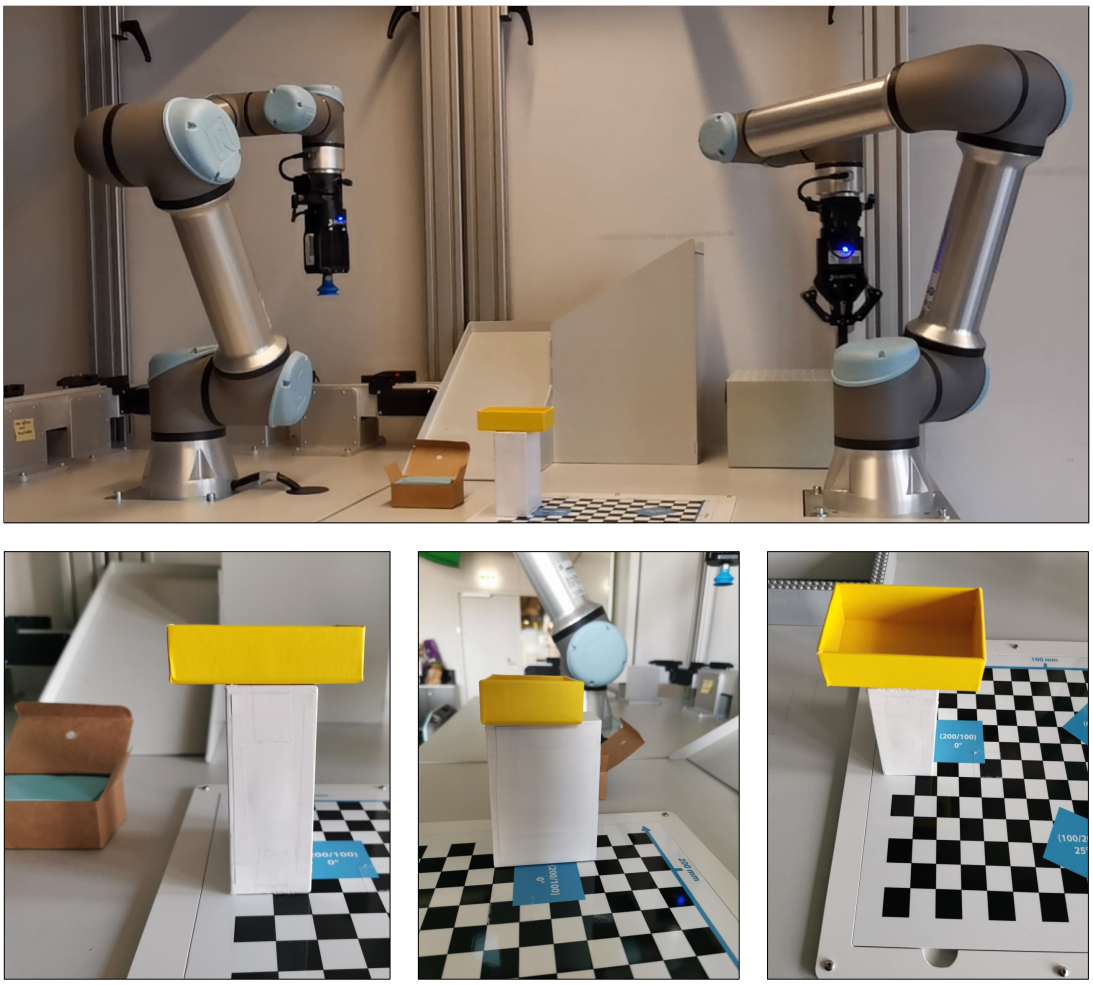


Figure 7.13: Physical cobot cell setup for Scenario 3: Assisted Packing

Independent Pick-and-Place

The use case with the least interaction, that is, the lowest level of collaboration where the cobots solely share their work environment, but do not exchange signals for the sake of common operation is Scenario 4: Independent Pick-and-Place. As the name indicates, the cobots in this scenario operate without dependencies or constraints with respect to one another, carrying out tasks that are part of different processes and that are performed upon different workpieces. Specifically, Robot A works on *Box-M*, placing it on the assembly lines, and Robot B works on *Box-S*, placing it on the slide.

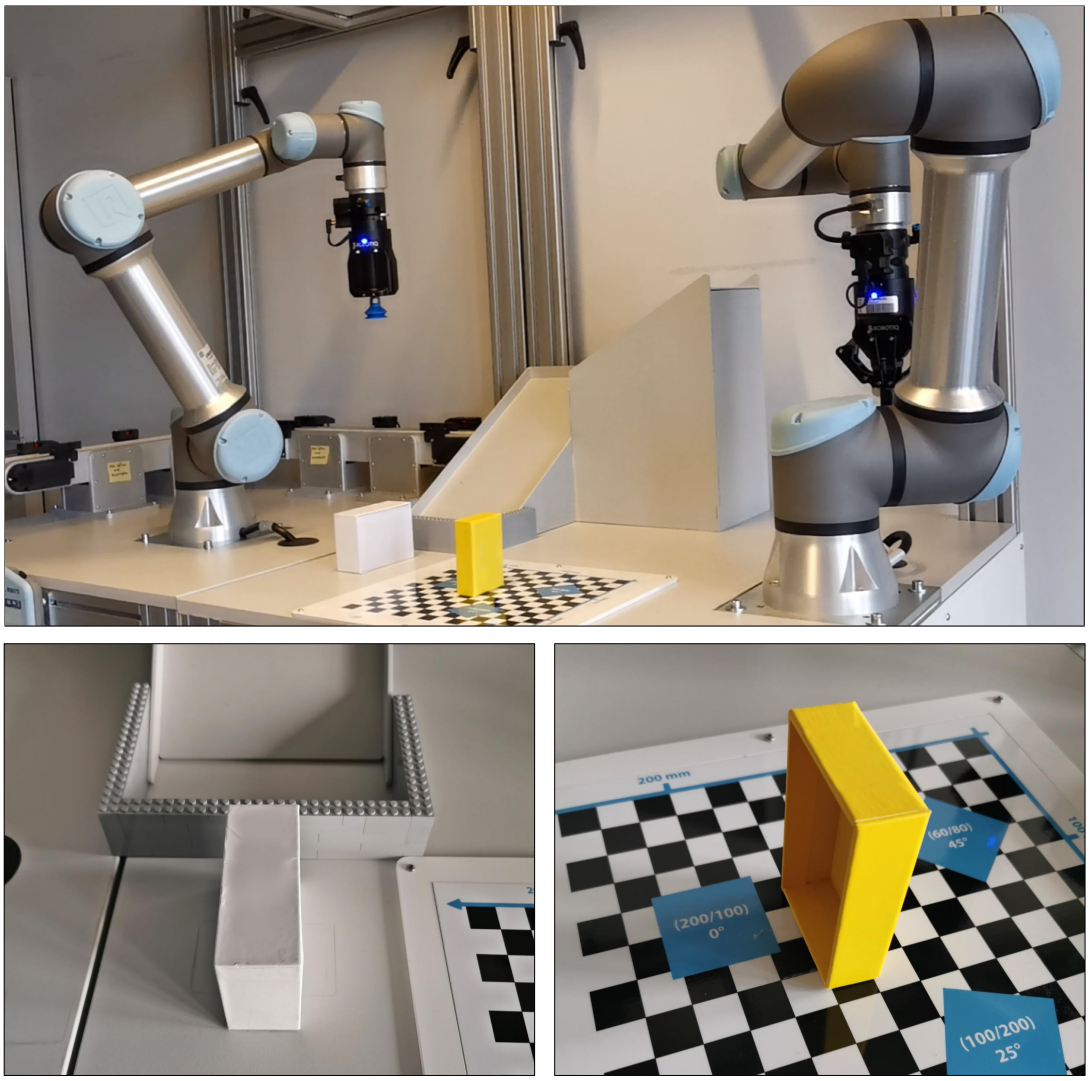


Figure 7.14: Physical cobot cell setup for Scenario 4: Independent Pick-and-Place

Note that in the implementation of this scenario, for the purpose of efficiency and convinience for performing multiple iterations without starting each cobot separately, Robot A forwards the "Start"

signal of the operator at the beginning of each iteration. However, this signaling is not considered collaboration in the same manner as in the other scenarios, as the tasks in the processes are completely independent.

7.2.3 Adaptations in URSim

As described in the evaluation steps at the beginning of this chapter, the programs created on the physical cobots in the research lab are transferred to a local MS Windows machine in order to use it in the installed URSim VMs. As mentioned before (cf. Section 6.3.2), using a shared folder, files can be exchanged between host and guest system. Thus, the transfer to URSim was done by copying the program files of the cobots to the corresponding virtual version's shared folder. Then, within URSim, the programs are moved to the *Programs UR5* folder on the respective VM. This makes the programs accessible from within the URSim UR5 application.

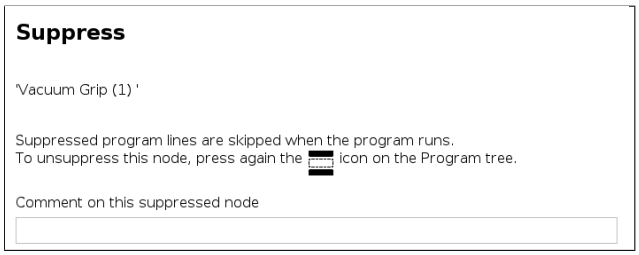
Inside the URSim environment, the programs needed to be adjusted for obvious reasons: absence of physical tools and different signal handling.

Additionally, what is still missing are the custom log output statements that will create a base for further analysis and mining of the cobot logs.

Particularly, the adaptation steps carried out in URSim include the following.

1. **"Suppress" *URCaps* program nodes (gripper-related actions).**

   This is done by selecting a program node and then the "Suppress" icon (right-most icon in the program tool bar). When a suppressed node is selected, more information on this functionality, as well as an input field for optional comments is shown in the *Command* tab:



2. **Add a loop in the outer scope of the program with the defined number of iterations (only on Robot A![1]).**

   In this project, the total number of iterations per scenario is 50. This is implemented using a *Loop* program node at the very beginning/top of the Robot Program, and moving all other nodes inside the loop.

3. **Define numeric variables for IDs of events and cases.**

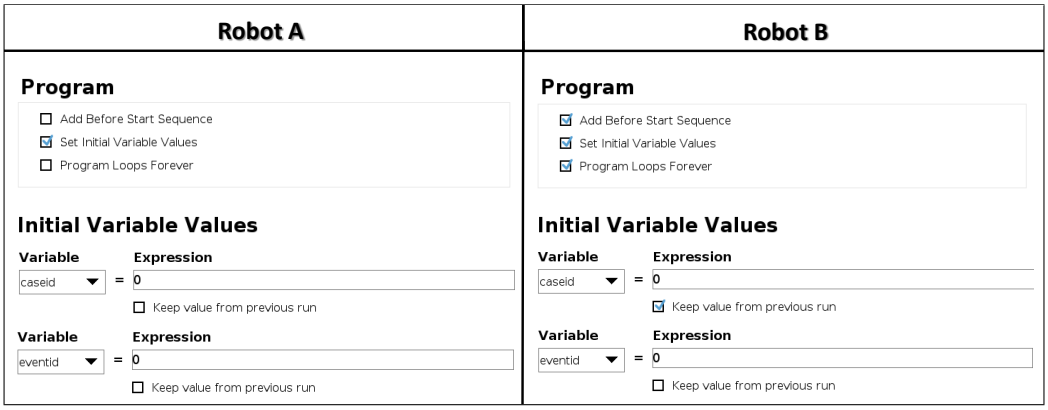
   The goal is to have two ID variables, one for each relevant activity (actionable event), and one for each iteration (case or trace). To create these, within the newly created loop, two *Assignment* program nodes (cf. Item 2.2.) need to be inserted. First, we create a variable

[1] As Robot A receives the start signal from the operator and initiates the execution of Robot B's activities, it controls the flow of the iterations.

called **eventid** and assign it the value "0" on Robot A, and the value "1" on Robot B. The reason for this distinction is that on Robot A, there is an additional event where the operator is the actor: the start signal event which indicates the beginning of an iteration. Second, a **caseid** variable is created, to which we assign the expression **caseid+1** (on both cobots). This ensures that each loop iteration (which is a new case/trace) is assigned a new ID: ID of the previous iteration incremented by 1.

After creating the variables, their initial values (before the execution of the Robot Program) need to be set. This is done via the Robot Program *Command* tab, where selecting "Set Initial Variable Values" creates a special node before the main Robot Program called "Init Variables" and opens the corresponding settings. Here, both the **eventid** and the **caseid** are set to 0.

However, note the differences in the setup depending on the cobot:



4. **Add a custom log output using the `textmsg` URScript function for every *start* and *end* of an actionable event [2].**

   In Section 5.1.3 we described in detail how custom outputs are written to cobot logs and what pieces of information those custom outputs consist of. In particular, recall that the format of the output string is `"LogID,ActorID,Activity,Target,EventID,Lifecycle,CaseID"`.

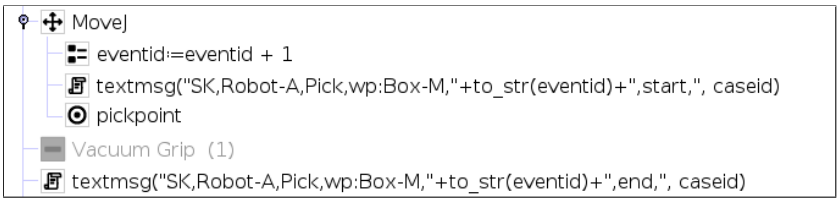
   The **textmsg** function accepts two parameters, where the second one is optional (recall Figure 5.4). The parameters can be of different types (i.e., they are not limited to strings). In the resulting log entry, these parameters are merged into a single value. In this case, all of the values in the custom output are strings except the EventID and the CaseID which are integers. Thus, we can pass one of the two as the second parameter, but we have to convert the other to a string when passing the values to the **textmsg** function. We use the **to_str** function for this purpose (see examples below).

   Additionally, note that the **eventid** is a variable, and we have to increment it before calling the **textmsg** function for the next event.

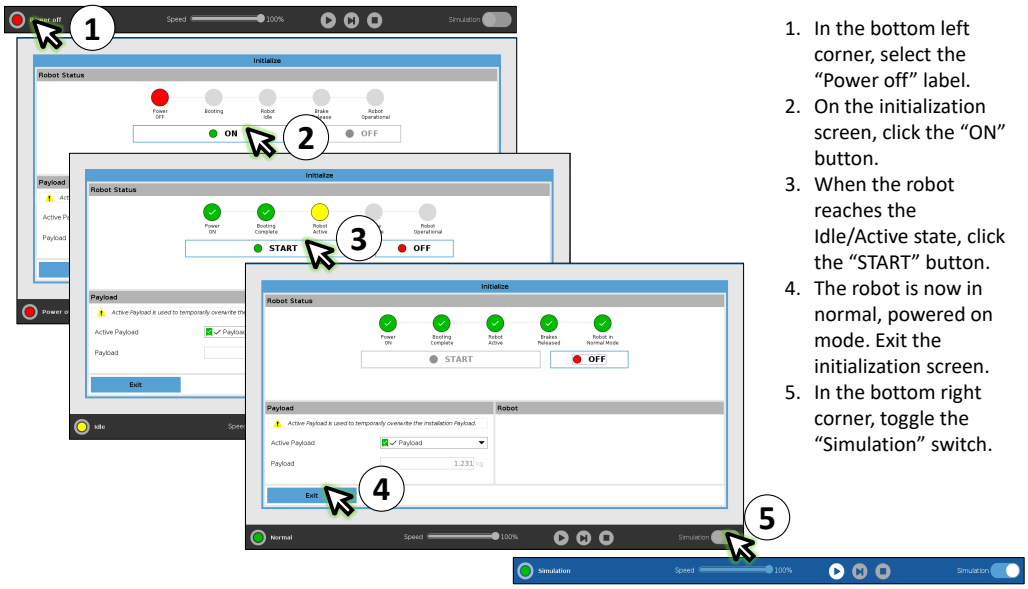
[2] Actionable events include sending a signal, moving to a target point/object, and picking and placing an object.

Since on Robot A we use the 0 event for the operator's signaling activity, we do not need to increment the ID in that case:

For all other events, we use the *Assignment* node to increment the `eventid` variable:



5. **When running a program, make sure to run it in *Simulation* mode.**

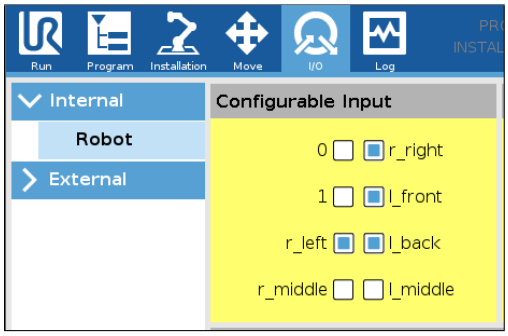


1. In the bottom left corner, select the "Power off" label.
2. On the initialization screen, click the "ON" button.
3. When the robot reaches the Idle/Active state, click the "START" button.
4. The robot is now in normal, powered on mode. Exit the initialization screen.
5. In the bottom right corner, toggle the "Simulation" switch.

6. **In *Simulation* mode, on the Robot A VM, for scenarios 1 and 4, make sure to activate[3] the following *Configurable Input* signals:**

[3] This will ensure that the programs where assembly line sensors provide feedback regarding workpiece location work without interruptions.

- r_left
- r_right
- l_back
- l_right



7.3 Log Export and Preparation

After all the iterations of all four scenarios have been executed to completion, the support files (containing the logs) can be exported, saved and then transferred to the host machine as shown in Figure 7.15.

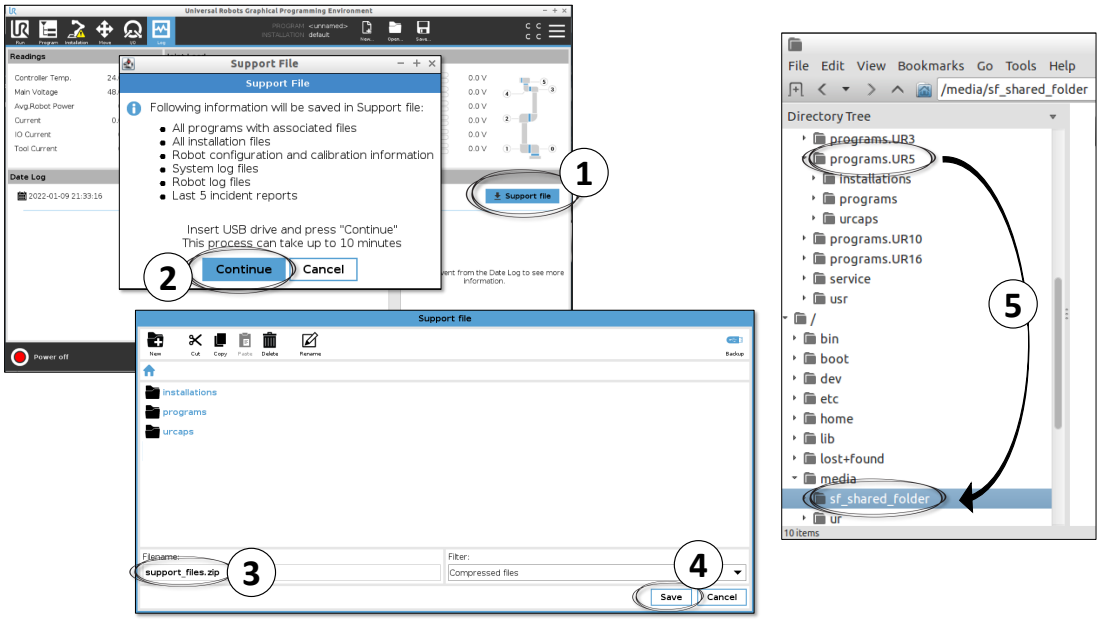


Figure 7.15: Steps for exporting and transferring the support file to local machine

Once the files are on the host machine, the steps described at the beginning of this chapter, namely the steps from Item 8 to Item 12, are carried out in the specified order.

After the separate scenario log files have been created and prepared as described (see Figure 7.16 for a sample of what a log file should look like), the data is ready to be analyzed and mined for insights using the CPM application.

```
3.5 :: 0006d19h55m08.486s :: 2023-12-19 15:19:52.516 :: -3 :: C0A0:7 :: null :: 1 :: progABC :: Program progABC started :: null
3.5 :: 0006d19h55m08.512s :: 2023-12-19 15:19:52.516 :: -3 :: C0A0:0 :: null :: 1 :: SK,Operator,Signal,START,0,start,1 ::  :: null
3.5 :: 0006d19h55m08.512s :: 2023-12-19 15:19:52.516 :: -3 :: C0A0:0 :: null :: 1 :: SK,Operator,Signal,START,0,end,1 ::  :: null
3.5 :: 0006d19h55m08.512s :: 2023-12-19 15:19:52.516 :: -3 :: C0A0:0 :: null :: 1 :: SK,Robot-A,Signal,GO,1,start,1 ::  :: null
3.5 :: 0006d19h55m08.512s :: 2023-12-19 15:19:52.516 :: -3 :: C0A0:0 :: null :: 1 :: SK,Robot-A,Signal,GO,1,end,1 ::  :: null
                                                                    •••
3.5 :: 0006d21h01m55.970s :: 2023-12-19 17:05:20.829 :: -3 :: C0A0:0 :: null :: 1 :: SK,Robot-A,Signal,FIN,22,start,50 ::  :: null
3.5 :: 0006d21h01m55.970s :: 2023-12-19 17:05:20.829 :: -3 :: C0A0:0 :: null :: 1 :: SK,Robot-A,Signal,FIN,22,end,50 ::  :: null
3.5 :: 0006d21h01m56.470s :: 2023-12-19 17:05:21.632 :: -3 :: C0A0:0 :: null :: 1 :: SK,Robot-A,Move,Home,23,start,50 ::  :: null
3.5 :: 0006d21h01m58.784s :: 2023-12-19 17:05:25.178 :: -3 :: C0A0:0 :: null :: 1 :: SK,Robot-A,Move,Home,23,end,50 ::  :: null
3.5 :: 0006d21h01m58.788s :: 2023-12-19 17:05:25.282 :: -3 :: C0A0:7 :: null :: 1 :: progABC :: Program progABC stopped :: null
```

Figure 7.16: A sample excerpt of a prepared log file from Robot A

7.4 Demo: Collaborative Pattern Miner

This section demonstrates how the CPM application works and how the intermediate processing steps are presented in the GUI.

Figure 7.17 shows the *main view* that appears when the application is first run. It contains five buttons, one for each collaboration scenario implemented as described in the previous sections, one additional button for uploading a custom (CSV) log file, and one for starting the log generator script (added for easy access directly from the GUI). In essence, the user can either select one of

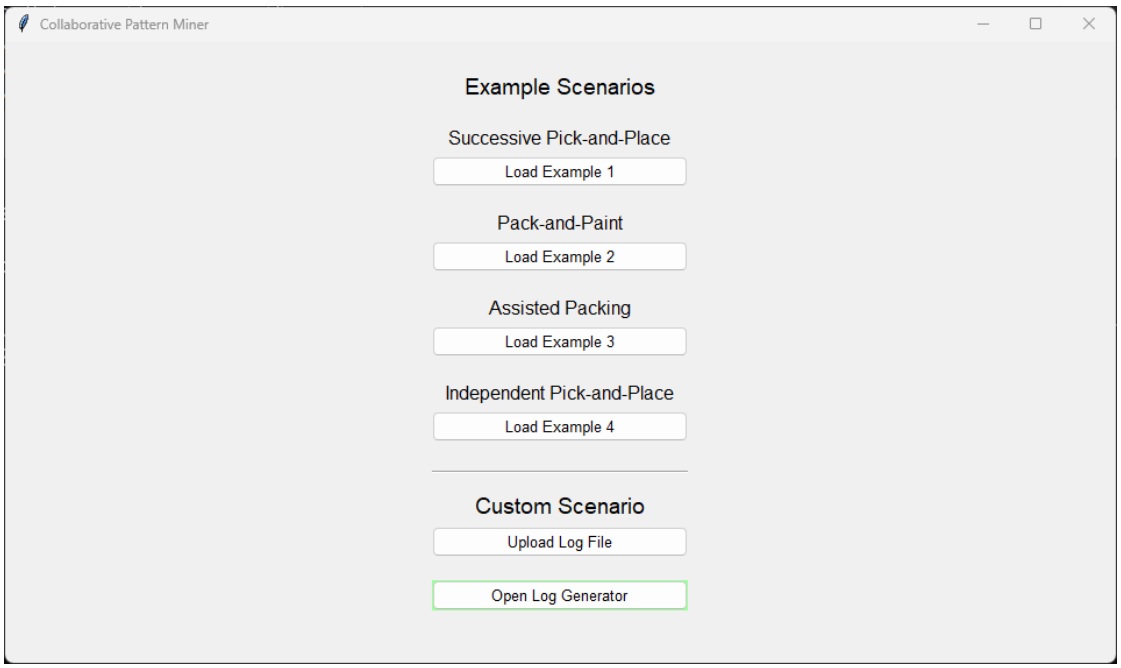
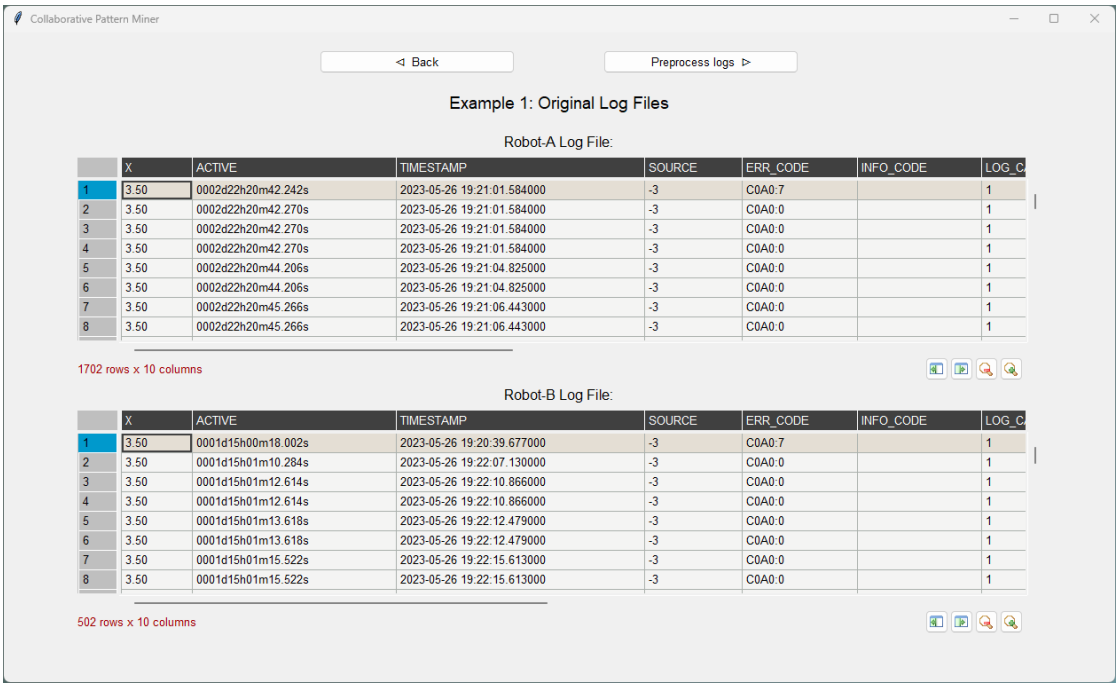


Figure 7.17: Main view in the CPM app with the four scenario selection choices and an "Upload" button.

the existing scenarios, or upload their own log file conforming to content format and structure as described in Section 6.3.3 under Additional Features: Log Generator and Log Upload. As the course of action slightly differs between these two choices, each is described separately in the following two sections.

7.4.1 Selecting a Scenario

After selecting a scenario, a new view appears, showing the log files that were read and loaded into the app. This is the *original logs view* and it shows the read and parsed log data from Robot A and Robot B of the chosen scenario. For example, this means that if Scenario 1 was selected, the app reads the *data/raw/robot-a/**logs_1.txt*** and *data/raw/robot-b/**logs_1.txt*** files and loads them into two table-like structures (dataframes). This view is shown in Figure 7.18 in the case of Scenario 1. Notice that in this step, no transformation of the content has been applied yet: the data was simply loaded into a dataframe and the columns were given custom names that (to the best of the author's ability) describe the corresponding column content.

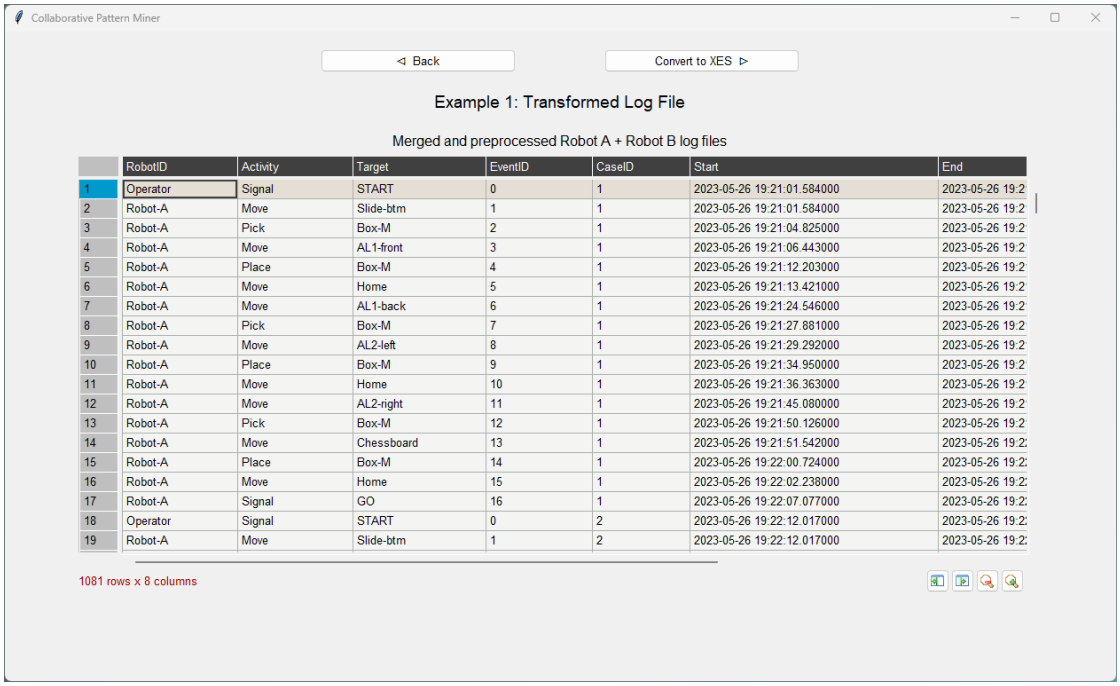


	X	ACTIVE	TIMESTAMP	SOURCE	ERR_CODE	INFO_CODE	LOG_C
1	3.50	0002d22h20m42.242s	2023-05-26 19:21:01.584000	-3	C0A0:7		1
2	3.50	0002d22h20m42.270s	2023-05-26 19:21:01.584000	-3	C0A0:0		1
3	3.50	0002d22h20m42.270s	2023-05-26 19:21:01.584000	-3	C0A0:0		1
4	3.50	0002d22h20m42.270s	2023-05-26 19:21:01.584000	-3	C0A0:0		1
5	3.50	0002d22h20m44.206s	2023-05-26 19:21:04.825000	-3	C0A0:0		1
6	3.50	0002d22h20m44.206s	2023-05-26 19:21:04.825000	-3	C0A0:0		1
7	3.50	0002d22h20m45.266s	2023-05-26 19:21:06.443000	-3	C0A0:0		1
8	3.50	0002d22h20m45.266s	2023-05-26 19:21:06.443000	-3	C0A0:0		1

	X	ACTIVE	TIMESTAMP	SOURCE	ERR_CODE	INFO_CODE	LOG_C
1	3.50	0001d15h00m18.002s	2023-05-26 19:20:39.677000	-3	C0A0:7		1
2	3.50	0001d15h01m10.284s	2023-05-26 19:22:07.130000	-3	C0A0:0		1
3	3.50	0001d15h01m12.614s	2023-05-26 19:22:10.866000	-3	C0A0:0		1
4	3.50	0001d15h01m12.614s	2023-05-26 19:22:10.866000	-3	C0A0:0		1
5	3.50	0001d15h01m13.618s	2023-05-26 19:22:12.479000	-3	C0A0:0		1
6	3.50	0001d15h01m13.618s	2023-05-26 19:22:12.479000	-3	C0A0:0		1
7	3.50	0001d15h01m15.522s	2023-05-26 19:22:15.613000	-3	C0A0:0		1
8	3.50	0001d15h01m15.522s	2023-05-26 19:22:15.613000	-3	C0A0:0		1

Figure 7.18: Original logs view in the CPM app for Scenario 1.

A simple navigation mechanism in the CPM is realized using buttons at the top of the GUI (in all views except the main view). As shown in the screenshots, the buttons can be used to move back and forth between the five views, and the right button is labeled with the next step in the processing in CPM. Thus, in the original logs view, the right button is annotated with "Preprocess logs". That is exactly what happens in the background, where the data displayed in the original logs view is cleaned and transformed into a format that can easily be converted into XES. As described in Section 5.2, this step includes merging the logs from the two cobots, column pruning, transforming the data, injecting an error case, creating lifecycle events from two-part events, and constructing events from information in the custom log outputs. This merged, transformed dataframe is shown in the next view: the *transformed logs view* (see Figure 7.19).



	RobotID	Activity	Target	EventID	CaseID	Start	End
1	Operator	Signal	START	0	1	2023-05-26 19:21:01.584000	2023-05-26 19:2
2	Robot-A	Move	Slide-btm	1	1	2023-05-26 19:21:01.584000	2023-05-26 19:2
3	Robot-A	Pick	Box-M	2	1	2023-05-26 19:21:04.825000	2023-05-26 19:2
4	Robot-A	Move	AL1-front	3	1	2023-05-26 19:21:06.443000	2023-05-26 19:2
5	Robot-A	Place	Box-M	4	1	2023-05-26 19:21:12.203000	2023-05-26 19:2
6	Robot-A	Move	Home	5	1	2023-05-26 19:21:13.421000	2023-05-26 19:2
7	Robot-A	Move	AL1-back	6	1	2023-05-26 19:21:24.546000	2023-05-26 19:2
8	Robot-A	Pick	Box-M	7	1	2023-05-26 19:21:27.881000	2023-05-26 19:2
9	Robot-A	Move	AL2-left	8	1	2023-05-26 19:21:29.292000	2023-05-26 19:2
10	Robot-A	Place	Box-M	9	1	2023-05-26 19:21:34.950000	2023-05-26 19:2
11	Robot-A	Move	Home	10	1	2023-05-26 19:21:36.363000	2023-05-26 19:2
12	Robot-A	Move	AL2-right	11	1	2023-05-26 19:21:45.080000	2023-05-26 19:2
13	Robot-A	Pick	Box-M	12	1	2023-05-26 19:21:50.126000	2023-05-26 19:2
14	Robot-A	Move	Chessboard	13	1	2023-05-26 19:21:51.542000	2023-05-26 19:2
15	Robot-A	Place	Box-M	14	1	2023-05-26 19:22:00.724000	2023-05-26 19:2
16	Robot-A	Move	Home	15	1	2023-05-26 19:22:02.238000	2023-05-26 19:2
17	Robot-A	Signal	GO	16	1	2023-05-26 19:22:07.077000	2023-05-26 19:2
18	Operator	Signal	START	0	2	2023-05-26 19:22:12.017000	2023-05-26 19:2
19	Robot-A	Move	Slide-btm	1	2	2023-05-26 19:22:12.017000	2023-05-26 19:2

Figure 7.19: Transformed logs view in the CPM app for Scenario 1.


```
<?xml version="1.0" encoding="utf-8" ?>
<log xes.version="1849-2016" xes.features="nested-attributes" xmlns="http://www.xes-standard.org/">
	<extension name="Time" prefix="time" uri="http://www.xes-standard.org/time.xesext" />
	<extension name="Concept" prefix="concept" uri="http://www.xes-standard.org/concept.xesext" />
	<string key="origin" value="csv" />
	<trace>
		<string key="concept:name" value="1" />
		<event>
			<string key="RobotID" value="Operator" />
			<string key="Activity" value="Signal" />
			<string key="Target" value="START" />
			<string key="EventID" value="0" />
			<string key="CaseID" value="1" />
			<date key="Start" value="2023-05-26T19:21:01.584000" />
			<date key="End" value="2023-05-26T19:21:01.584000" />
			<string key="Label" value="0 Operator: Signal START" />
			<string key="concept:name" value="0" />
			<date key="time:timestamp" value="2023-05-26T19:21:01.584000" />
			<int key="@@index" value="0" />
			<int key="@@case_index" value="0" />
		</event>
		<event>
			<string key="RobotID" value="Robot-A" />
			<string key="Activity" value="Move" />
			<string key="Target" value="Slide-btm" />
			<string key="EventID" value="1" />
			<string key="CaseID" value="1" />
			<date key="Start" value="2023-05-26T19:21:01.584000" />
			<date key="End" value="2023-05-26T19:21:04.825000" />
			<string key="Label" value="1 Robot-A: Move to Slide-btm" />
			<string key="concept:name" value="1" />
			<date key="time:timestamp" value="2023-05-26T19:21:04.825000" />
			<int key="@@index" value="1" />
			<int key="@@case_index" value="0" />
		</event>
		<event>
			<string key="RobotID" value="Robot-A" />
			<string key="Activity" value="Pick" />
			<string key="Target" value="Box-M" />
			<string key="EventID" value="2" />
```

Figure 7.20: XES view in the CPM app for Scenario 1.

The step that follows the preprocessing is a rather minor step of converting the prepared dataframe into an XES file which is displayed in the corresponding *XES view* (see Figure 7.20). The fifth and final view is the *process view*, as shown in Figure 7.21. Here, the result of process discovery is displayed (the BPMN process model), and the detected collaboration pattern with an according description, as well as an example case with relevant attributes that can be shown/hidden via button click. Additionally, for the "longer" process models, i.e., with multiple sequential activities, a button for simplifying the models appears. The simplifying is carried out by merging what can be considered substeps of an activity into one. An example of this is demonstrated in Figure 7.22. Notice that the error occurred in a case after one of the substeps, and thus, in the simplified model, the subprocess occurs with, as well as without an error.

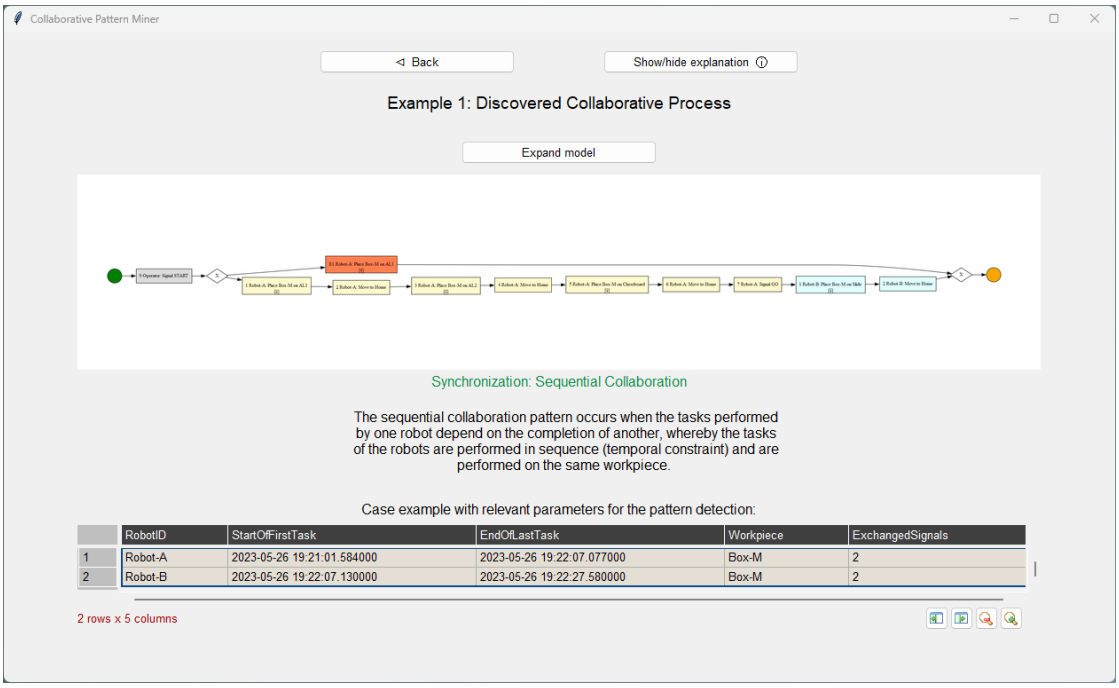


Figure 7.21: Process view in the CPM app for Scenario 1.

In summary, the CPM application is tool for presenting the intermediate processing steps performed on the cobot logs of the four implemented scenarios. It visualizes the contents of the logs in different stages of transformation in table format, XES format, and, finally, in form of a process model. The CPM showcases the mining process which leads to a conclusion regarding the nature of the collaboration between the cobots in the constructed use cases.

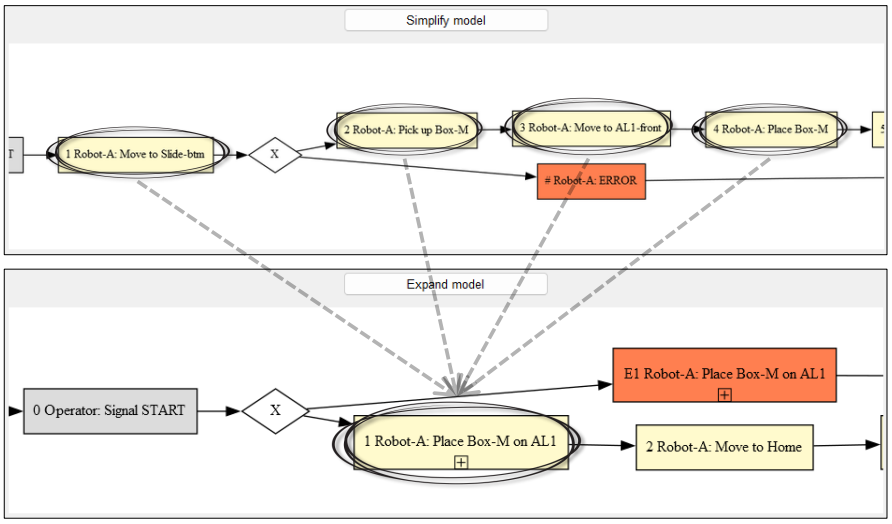


Figure 7.22: The BPMN process model in expanded mode (top) and simplified mode (bottom).

7.4.2 Uploading a Log File

After selecting the upload button in the main view, a new file-upload window appears where CSV files can be chosen from the file system (see Figure 7.23). Note again that the log file should conform to the content format and structure as described in Section 6.3.3 under Additional Features: Log Generator and Log Upload.

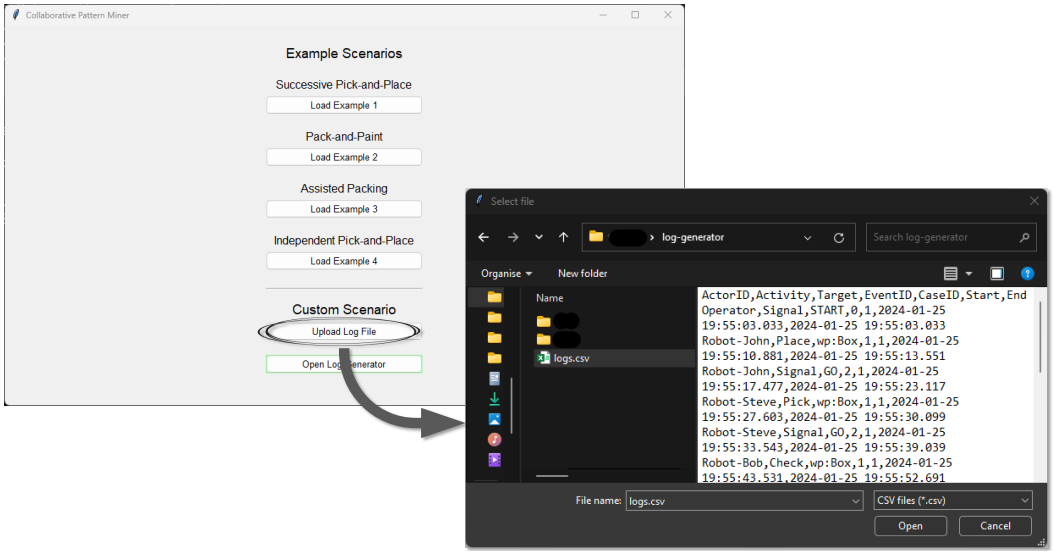


Figure 7.23: Window for choosing a log file in CSV format from the file system for upload in the CPM app.

Next, instead of displaying the *original logs view*, and subsequently the *transformed logs view* as

is the case when choosing a predefined scenario, the *XES view* is shown directly. This is because the app expects the log file to be in the required format for discovering the process and identifying the occurring pattern. The goal of the upload function is to demonstrate and evaluate the implemented algorithm for identifying a collaborative pattern based on a custom log dataset. Thus, after the XES-formatted log file was displayed in the *XES view*, the discovered process model and the identified pattern are displayed in the *process view*, as can be seen in Figure 7.24.

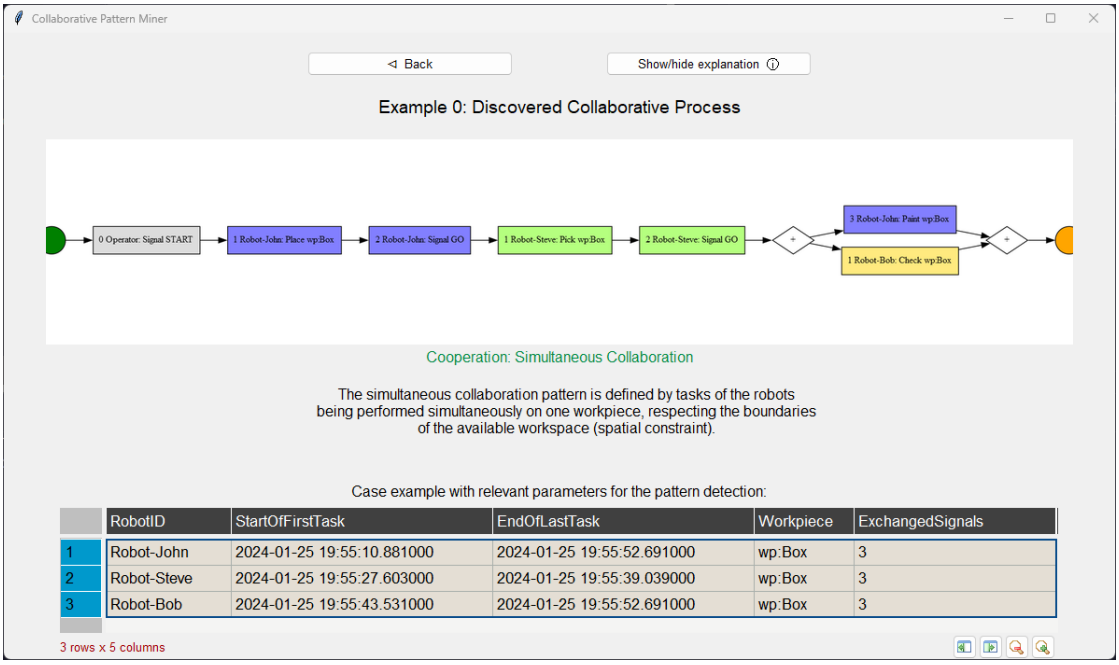


Figure 7.24: Resulting *process view* based on an uploaded log file.

7.5 Results: Identifying Collaborative Patterns

In this section, the results obtained using the CPM app are presented for each of the four predefined scenarios, as well as for examples with uploaded custom log files.

7.5.1 Predefined Scenarios

This section presents the results obtained in the CPM app when selecting each of the four predefined scenarios in the main view. The results are presented in terms of outputs in the CPM app, the generated BPMN models, and the discovered collaboration patterns. Each case result is briefly analyzed and elaborated on with references to the constraints defined in the algorithm.

Scenario 1: Successive Pick-and-Place

This scenario demonstrates the synchronized pattern in collaboration between two cobots. As the name indicates, the cobots perform their respective activities in succession: the first task of the second cobots starts only after the last task of the first cobot was completed. The resulting process model reflects this flow of events (see Figure 7.26 and Figure 7.27) and the synchronized collaborative pattern is successfully identified (see Figure 7.25).

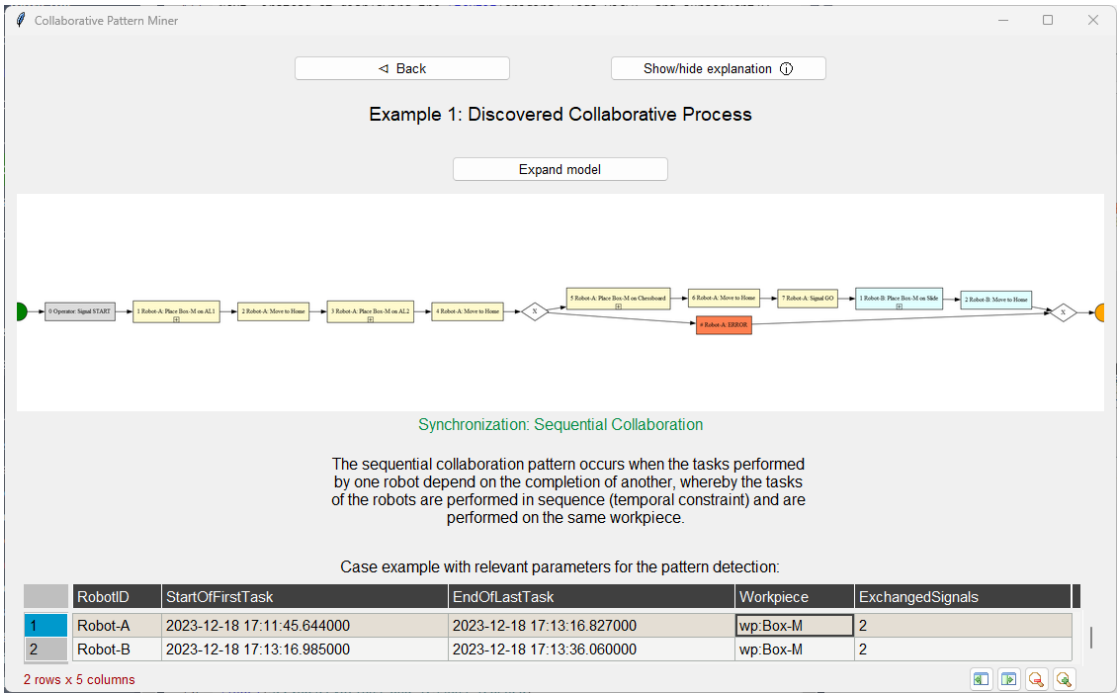
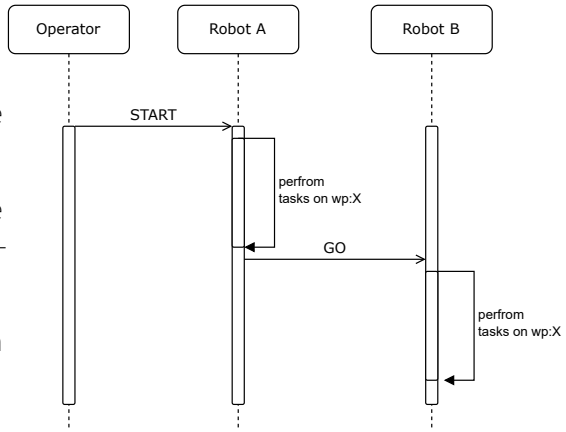


Figure 7.25: Resulting *process view* for Scenario 1 in the CPM app.

This pattern was identified because:

- the first task of Robot B starts only after the last task of Robot A was completed,
- the cobots operate on the same workpiece (as indicated by the "wp:" prefix in the target annotation), and
- the total number of exchanged signals in the process equals the number of cobots.



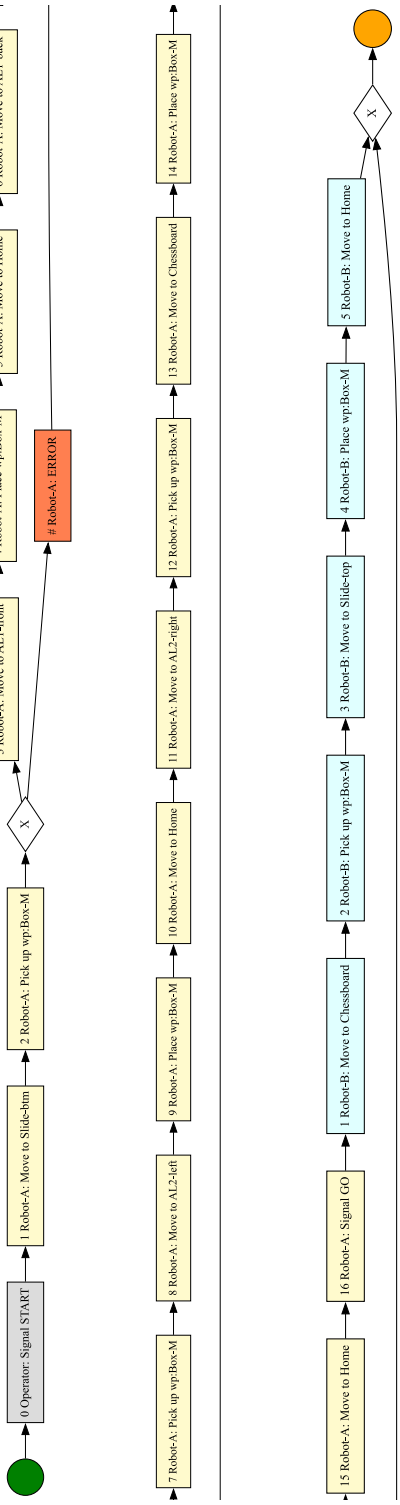


Figure 7.26: Resulting (full-length) process model for Scenario 1.

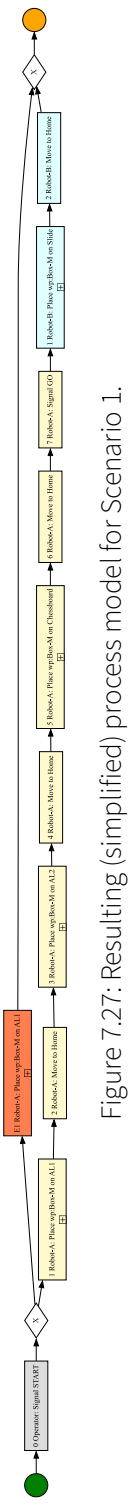


Figure 7.27: Resulting (simplified) process model for Scenario 1.

Scenario 2: Pack-and-Paint

This scenario demonstrates a collaborative behavior in which the two cobots operate on the same workpiece as is the case in Scenario 1, but the tasks are not constrained in a temporal manner. The activities of the cobots are instead spatially restricted, as the goal is for the cobots to perform their respective (mutually distinct) tasks on the same workpiece simultaneously. The described process is successfully discovered (depicted in Figure 7.29) and the corresponding collaborative pattern correctly identified (see Figure 7.28).

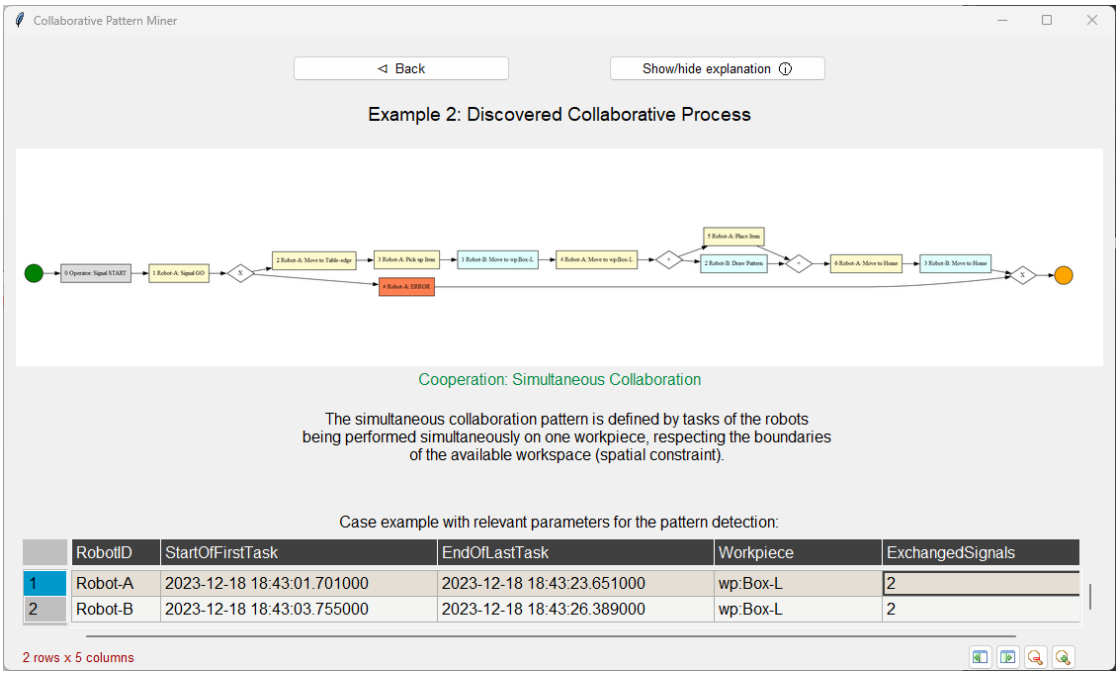
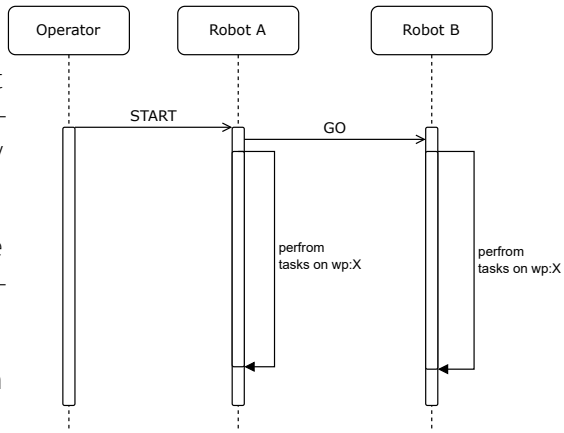


Figure 7.28: Resulting *process view* for Scenario 2 in the CPM app.

This pattern was identified because:

- the tasks of Robot A and Robot B are not performed in succession, there is no specific ordering and parallel activities may occur,
- the cobots operate on the same workpiece (as indicated by the "wp:" prefix in the target annotation), and
- the total number of exchanged signals in the process equals the number of cobots.



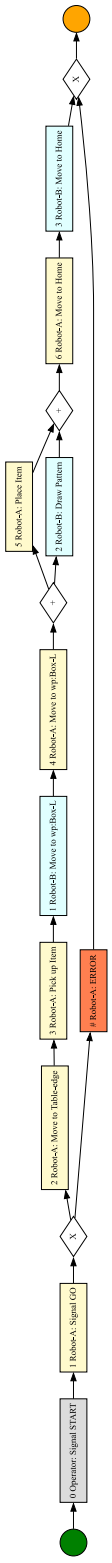


Figure 7.29: Resulting process model for Scenario 2.

Scenario 3: Assisted Packing

This scenario represents the highest level of collaboration, collaboration in a narrow sense or *supportive* collaboration, where the cobots work together to reach a common goal. In this scenario, one cobot "supports" the other by holding the workpiece in an easily accessible position while the other cobot performs its task (packing) on the same workpiece. This high degree of collaborative behavior is reflected in the higher count of exchanged signals (which can be translated into communication among the actors), as there is a need to coordinate their respective tasks. The discovered process model (see Figure 7.31 and Figure 7.32) illustrates this scenario well, and the supportive collaboration pattern is successfully identified (see Figure 7.30).

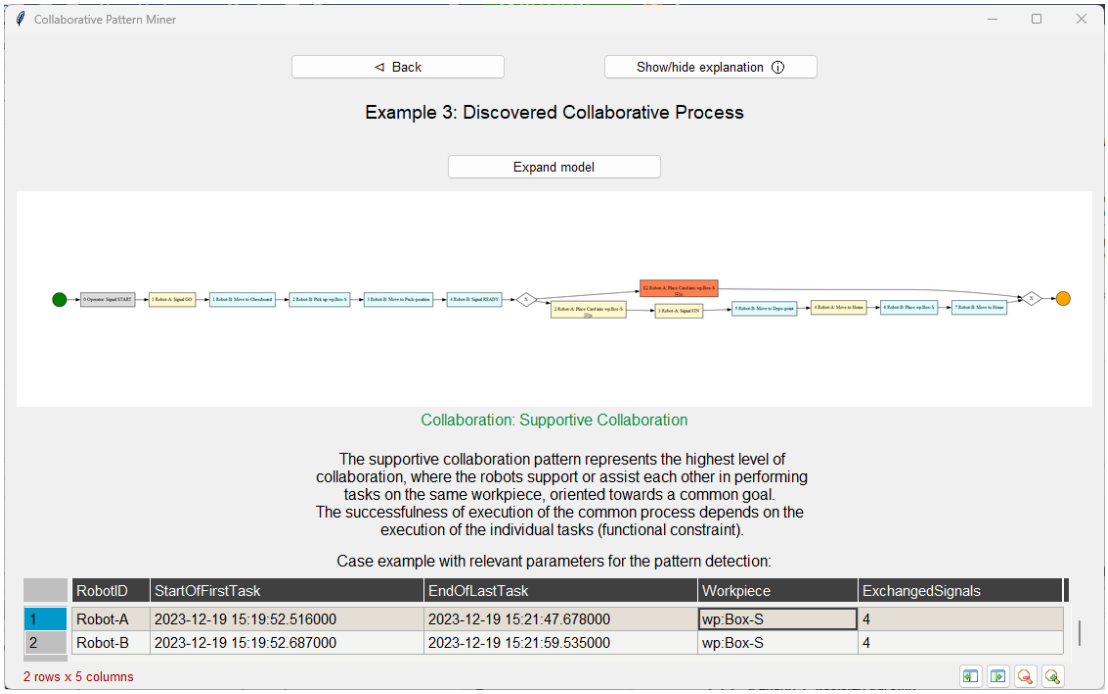
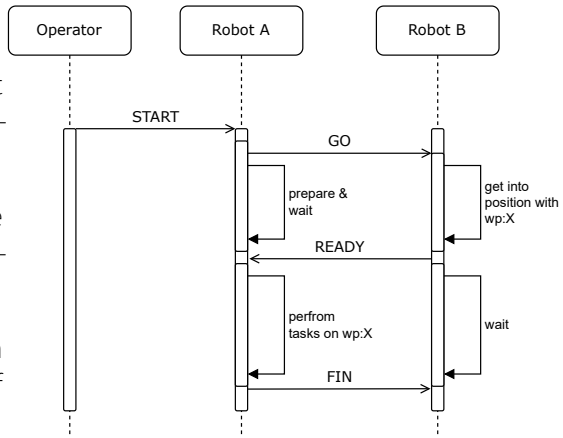


Figure 7.30: Resulting *process view* for Scenario 3 in the CPM app.

This pattern was identified because:

- the tasks of Robot A and Robot B are not performed in succession and multiple parallel activities may occur,
- the cobots operate on the same workpiece (as indicated by the "wp:" prefix in the target annotation), and
- the total number of exchanged signals in the process is greater than the number of cobots.



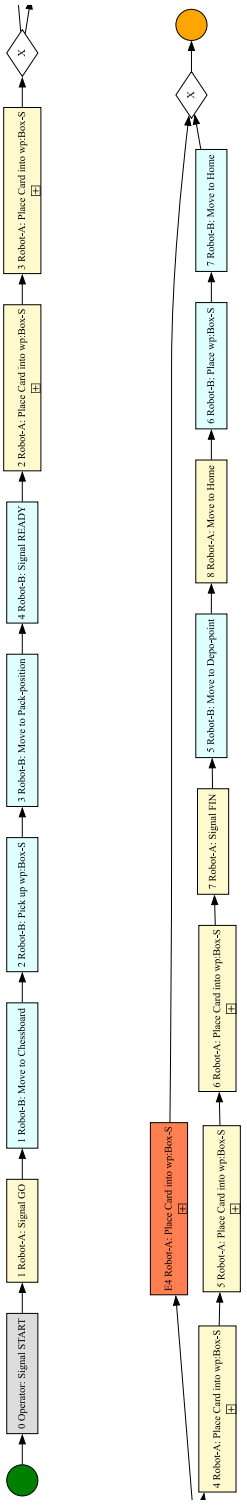


Figure 7.31: Resulting (full-length) process model for Scenario 3.

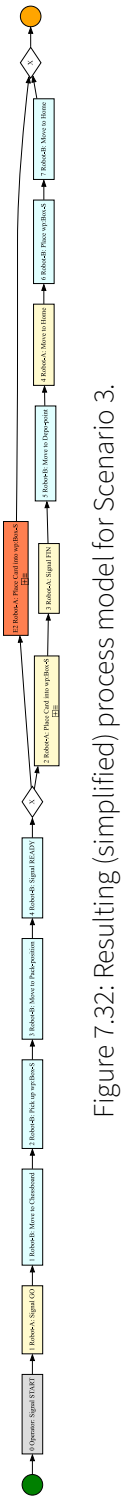


Figure 7.32: Resulting (simplified) process model for Scenario 3.

Scenario 4: Independent Pick-and-Place

This scenario demonstrates completely temporally, spatially and functionally independent tasks performed on different workpieces by the cobots. Nevertheless, it still represents a collaborative pattern, as the cobots operate in the same work environment in close proximity. The described process (see Figure 7.34), as well as the corresponding pattern (see Figure 7.33), are successfully discovered and displayed in the CPM app.

Notice how, unlike in the other examples, in this case the error does not affect both cobots but just the one on which the error occurred. This additionally shows the independence of the cobots in performing their respective activities.

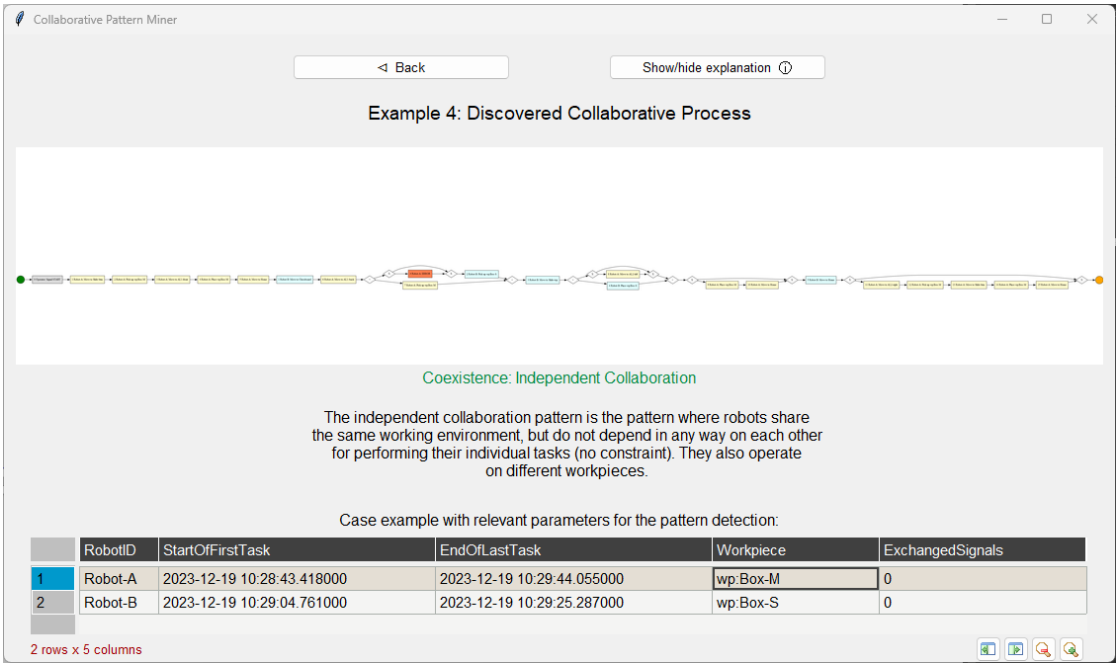
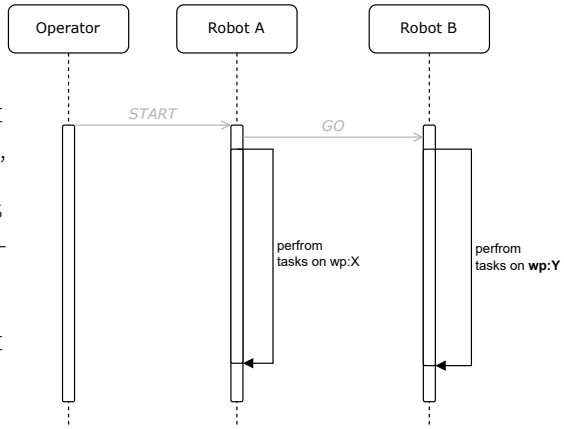


Figure 7.33: Resulting *process view* for Scenario 4 in the CPM app.

This pattern was identified because:

- the tasks of Robot A and the tasks of Robot B do not depend on each other in any way,
- the cobots operate on different workpieces (as indicated by the "wp:" prefix in the target annotation), and
- there are no exchanged signals (relevant to the process).



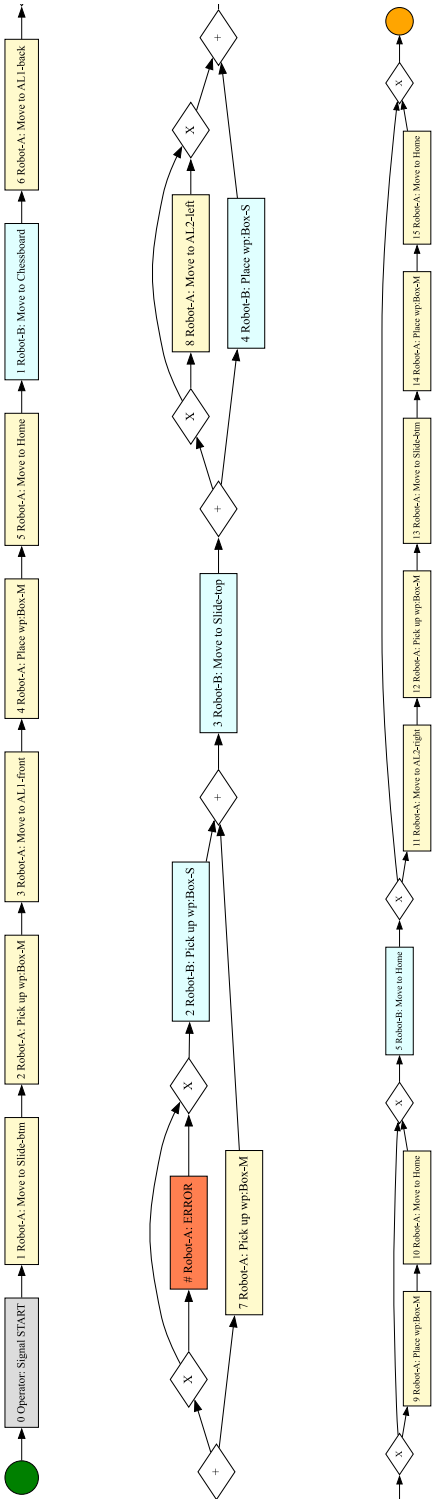


Figure 7.34: Resulting process model for Scenario 4.

7.5.2 Custom Scenarios (Upload Examples)

The additional "File Upload" feature enables loading custom process logs into the CPM app in form of a CSV file whose structure and content meet the necessary requirements defined in Section 6.3.3. To make the creation of such a file easier, a simple command-line log generator was developed. The log generator is easily accessible from the CPM app: in the main view, a click on the "Open Log Generator" button at the bottom runs the program in a separate command prompt window (see Figure 7.35).

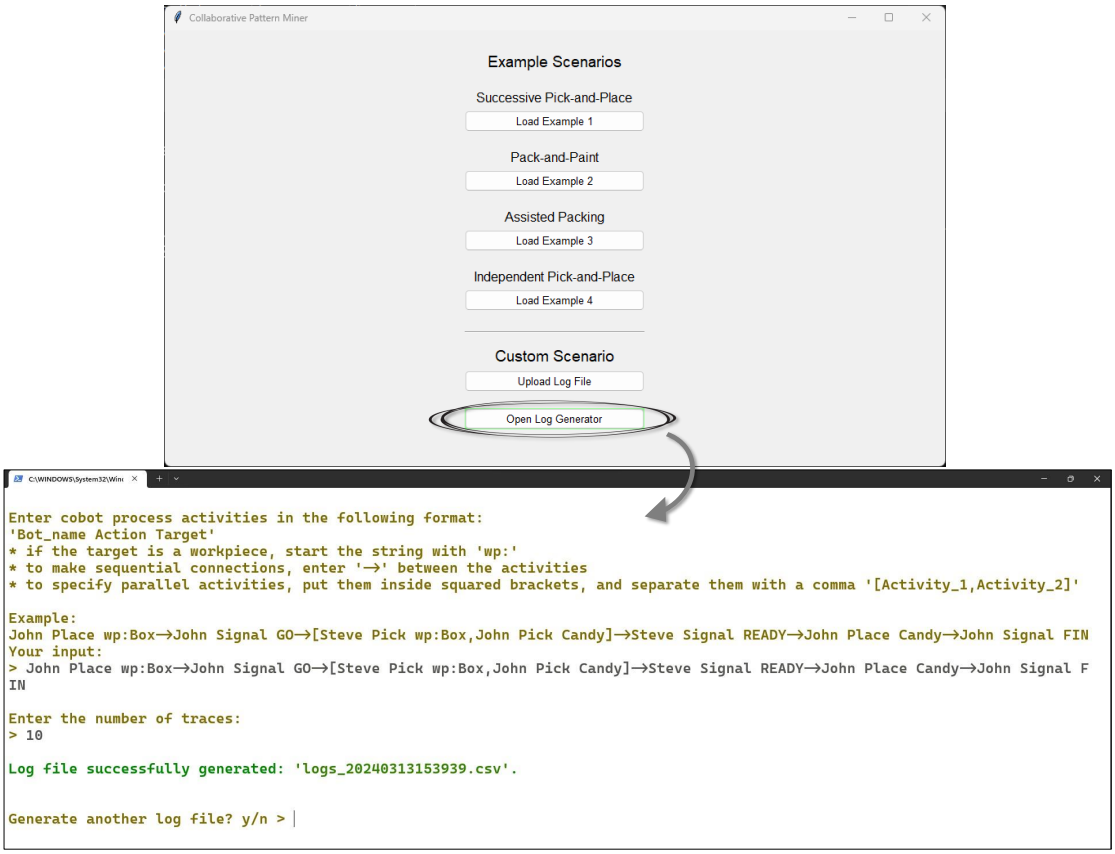


Figure 7.35: Opening the log generator from the CPM GUI.

The log generator firstly outputs the instructions for the user, provides a sample input process, and then prompts for the process input and the number of traces of the given process to be generated. The resulting CSV file is then stored into the `data/custom_csv` folder. A history of provided inputs, as well as the path where the generated file was stored, is logged into a `history.txt` file in the `log_generator` folder in the project.

The following sections describe the results obtained in the CPM app upon uploading a custom log file created using the log generator program.

Custom Synchronized Collaboration Example

The first example demonstrates the creation and discovery (Figure 7.36) of a synchronized collaboration process with three cobots and 5 traces.
The following input values are passed to the log generator:

- **Input process:**

  John Place wp:Box->John Signal GO->Steve Pick wp:Box->Steve Signal GO->Bob Paint wp:Box

- **Number of traces:**

  5

- **The generated CSV file:**

ActorID	Activity	Target	EventID	CaseID	Start	End
Operator	Signal	START	0	1	2024-03-13 15:58:45.027	2024-03-13 15:58:45.027
Robot-John	Place	wp:Box	1	1	2024-03-13 15:58:51.319	2024-03-13 15:58:59.421
Robot-John	Signal	GO	2	1	2024-03-13 15:59:04.859	2024-03-13 15:59:06.871
Robot-Steve	Pick	wp:Box	1	1	2024-03-13 15:59:14.483	2024-03-13 15:59:16.521
Robot-Steve	Signal	GO	2	1	2024-03-13 15:59:20.617	2024-03-13 15:59:26.251
Robot-Bob	Paint	wp:Box	1	1	2024-03-13 15:59:32.195	2024-03-13 15:59:35.047
Operator	Signal	START	0	2	2024-03-13 15:59:43.859	2024-03-13 15:59:43.859
Robot-John	Place	wp:Box	1	2	2024-03-13 15:59:46.287	2024-03-13 15:59:52.649
Robot-John	Signal	GO	2	2	2024-03-13 15:59:55.425	2024-03-13 16:00:01.183
Robot-Steve	Pick	wp:Box	1	2	2024-03-13 16:00:08.833	2024-03-13 16:00:12.719
Robot-Steve	Signal	GO	2	2	2024-03-13 16:00:15.825	2024-03-13 16:00:19.201
Robot-Bob	Paint	wp:Box	1	2	2024-03-13 16:00:21.409	2024-03-13 16:00:29.565
Operator	Signal	START	0	3	2024-03-13 16:00:35.275	2024-03-13 16:00:35.275
Robot-John	Place	wp:Box	1	3	2024-03-13 16:00:38.625	2024-03-13 16:00:47.283
Robot-John	Signal	GO	2	3	2024-03-13 16:00:50.431	2024-03-13 16:00:56.145
Robot-Steve	Pick	wp:Box	1	3	2024-03-13 16:00:58.163	2024-03-13 16:01:03.819
Robot-Steve	Signal	GO	2	3	2024-03-13 16:01:10.461	2024-03-13 16:01:19.147
Robot-Bob	Paint	wp:Box	1	3	2024-03-13 16:01:21.265	2024-03-13 16:01:25.613
Operator	Signal	START	0	4	2024-03-13 16:01:31.541	2024-03-13 16:01:31.541
Robot-John	Place	wp:Box	1	4	2024-03-13 16:01:34.705	2024-03-13 16:01:43.441
Robot-John	Signal	GO	2	4	2024-03-13 16:01:47.239	2024-03-13 16:01:57.151
Robot-Steve	Pick	wp:Box	1	4	2024-03-13 16:02:06.083	2024-03-13 16:02:15.247
Robot-Steve	Signal	GO	2	4	2024-03-13 16:02:19.329	2024-03-13 16:02:28.491
Robot-Bob	Paint	wp:Box	1	4	2024-03-13 16:02:36.103	2024-03-13 16:02:43.003
Operator	Signal	START	0	5	2024-03-13 16:02:51.187	2024-03-13 16:02:51.187
Robot-John	Place	wp:Box	1	5	2024-03-13 16:02:58.787	2024-03-13 16:03:03.135
Robot-John	Signal	GO	2	5	2024-03-13 16:03:06.361	2024-03-13 16:03:08.557
Robot-Steve	Pick	wp:Box	1	5	2024-03-13 16:03:17.875	2024-03-13 16:03:22.825
Robot-Steve	Signal	GO	2	5	2024-03-13 16:03:31.359	2024-03-13 16:03:39.367
Robot-Bob	Paint	wp:Box	1	5	2024-03-13 16:03:41.699	2024-03-13 16:03:49.489

As you can see in the resulting BPMN process model in Figure 7.37, the activities of different cobots are sequentially ordered, and they all work on the same workpiece. Additionally, the number of exchanged signals in the process (see the "Case example" in Figure 7.36) also indicates a sequential interactivity among the cobots: out of a total count of 3 exchanged signals, 1 is the Operator's, and 2 are the cobots' - after a cobot completes its tasks, it signals the "next" one to start its operation. As there are 3 cobots in this case, and the subprocesses of each cobot are successive, a count of 2 cobot signals indicates a sequential collaboration process.

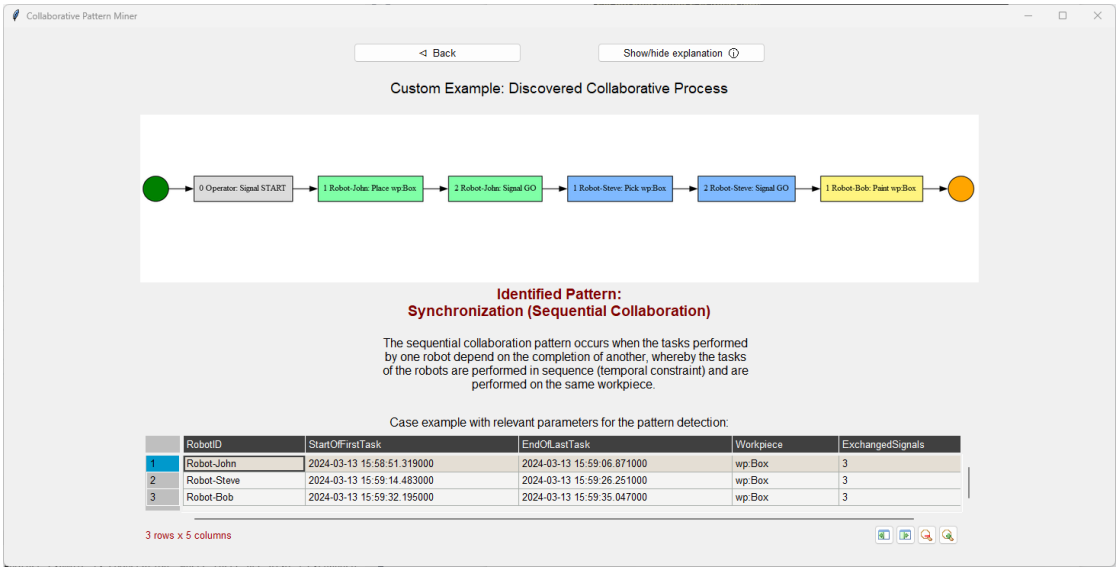


Figure 7.36: Resulting *process view* for a custom example of a synchronized collaboration process in the CPM app.

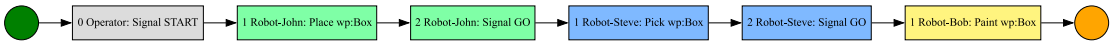


Figure 7.37: Resulting BPMN model of a custom example of a synchronized collaboration process.

Custom Cooperation Collaboration Example

Another example is cooperation, where there are also 3 exchanged signals in total, but there are parallel activities on the same workpiece, instead of sequential.
Using the log generator, we generate such an example with the following values:

- **Input process:**

  John Place wp:Box->John Signal GO->Steve Pick wp:Box->Steve Signal GO->[Bob Check wp:Box,John Paint wp:Box]

- **Number of traces:**

  5

- **The generated CSV file:**

ActorID	Activity	Target	EventID	CaseID	Start	End
Operator	Signal	START	0	1	2024-03-13 21:35:26.155	2024-03-13 21:35:26.155
Robot-John	Place	wp:Box	1	1	2024-03-13 21:35:31.349	2024-03-13 21:35:39.349
Robot-John	Signal	GO	2	1	2024-03-13 21:35:43.083	2024-03-13 21:35:49.621
Robot-Steve	Pick	wp:Box	1	1	2024-03-13 21:35:58.881	2024-03-13 21:36:08.627
Robot-Steve	Signal	GO	2	1	2024-03-13 21:36:14.937	2024-03-13 21:36:22.135
Robot-Bob	Check	wp:Box	1	1	2024-03-13 21:36:26.821	2024-03-13 21:36:31.953
Robot-John	Paint	wp:Box	3	1	2024-03-13 21:36:26.821	2024-03-13 21:36:31.953
Operator	Signal	START	0	2	2024-03-13 21:36:50.469	2024-03-13 21:36:50.469
Robot-John	Place	wp:Box	1	2	2024-03-13 21:36:54.525	2024-03-13 21:37:00.995
Robot-John	Signal	GO	2	2	2024-03-13 21:37:09.673	2024-03-13 21:37:14.681
Robot-Steve	Pick	wp:Box	1	2	2024-03-13 21:37:19.851	2024-03-13 21:37:26.309
Robot-Steve	Signal	GO	2	2	2024-03-13 21:37:35.883	2024-03-13 21:37:43.249
Robot-John	Paint	wp:Box	3	2	2024-03-13 21:37:51.087	2024-03-13 21:37:56.973
Robot-Bob	Check	wp:Box	1	2	2024-03-13 21:37:51.087	2024-03-13 21:37:56.973
Operator	Signal	START	0	3	2024-03-13 21:38:15.869	2024-03-13 21:38:15.869
Robot-John	Place	wp:Box	1	3	2024-03-13 21:38:23.299	2024-03-13 21:38:32.153
Robot-John	Signal	GO	2	3	2024-03-13 21:38:39.593	2024-03-13 21:38:45.475
Robot-Steve	Pick	wp:Box	1	3	2024-03-13 21:38:55.353	2024-03-13 21:39:05.213
Robot-Steve	Signal	GO	2	3	2024-03-13 21:39:14.661	2024-03-13 21:39:23.899
Robot-Bob	Check	wp:Box	1	3	2024-03-13 21:39:31.001	2024-03-13 21:39:39.565
Robot-John	Paint	wp:Box	3	3	2024-03-13 21:39:31.001	2024-03-13 21:39:39.565
Operator	Signal	START	0	4	2024-03-13 21:40:03.191	2024-03-13 21:40:03.191
Robot-John	Place	wp:Box	1	4	2024-03-13 21:40:05.951	2024-03-13 21:40:14.275
Robot-John	Signal	GO	2	4	2024-03-13 21:40:17.743	2024-03-13 21:40:24.955
Robot-Steve	Pick	wp:Box	1	4	2024-03-13 21:40:31.731	2024-03-13 21:40:40.445
Robot-Steve	Signal	GO	2	4	2024-03-13 21:40:47.479	2024-03-13 21:40:55.367
Robot-John	Paint	wp:Box	3	4	2024-03-13 21:40:58.527	2024-03-13 21:41:03.221
Robot-Bob	Check	wp:Box	1	4	2024-03-13 21:40:58.527	2024-03-13 21:41:03.221
Operator	Signal	START	0	5	2024-03-13 21:41:18.719	2024-03-13 21:41:18.719
Robot-John	Place	wp:Box	1	5	2024-03-13 21:41:21.161	2024-03-13 21:41:25.125
Robot-John	Signal	GO	2	5	2024-03-13 21:41:28.285	2024-03-13 21:41:30.833
Robot-Steve	Pick	wp:Box	1	5	2024-03-13 21:41:34.257	2024-03-13 21:41:43.079
Robot-Steve	Signal	GO	2	5	2024-03-13 21:41:45.617	2024-03-13 21:41:50.341
Robot-Bob	Check	wp:Box	1	5	2024-03-13 21:41:54.185	2024-03-13 21:41:57.215
Robot-John	Paint	wp:Box	3	5	2024-03-13 21:41:54.185	2024-03-13 21:41:57.215

Figure 7.38 shows the corresponding result (process view) in the CPM app, where the cooperation pattern was correctly identified. This pattern was identified because there were 3 exchanged signals in total, the same workpiece was used by all 3 cobots, but - unlike the previous example - the subprocesses of the cobots were not executed in succession (sequentially). The process model itself can be seen in 7.39.

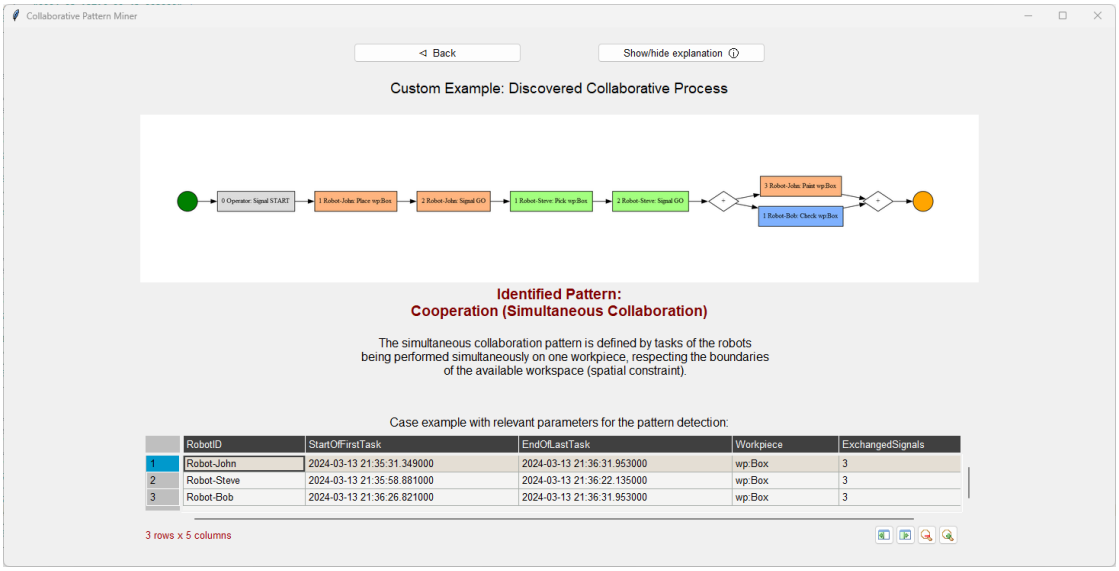


Figure 7.38: Resulting *process view* for a custom example of a cooperation collaboration process in the CPM app.

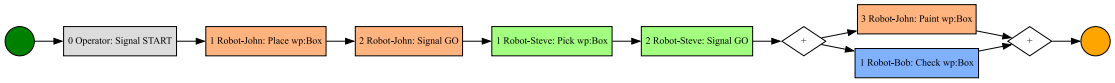


Figure 7.39: Resulting BPMN model of a custom example of a cooperation collaboration process.

Simulating an Error Case Es mentioned in Section 6.3.3Additional Features: Log Generator and Log Upload, an error case can be created directly in the CSV file by modifying a log entry. Using the current cooperation pattern example and the generated CSV file, the modifications can be performed as shown in Figure 7.40. In this example, the second event in the second trace is altered to contain "Error" in the Activity column, a single space " " in the Target column, and "E2" in the EventID column. Note that the other events in the affected trace are removed: this emphasizes the codependence of the events in the process and thus illustrates the collaborative nature of the process. However, this step is optional as it does not affect the algorithm's outcome.

Before

	ActorID,Activity,Target,EventID,CaseID,Start,End
Trace 1	Operator,Signal,START,0,1,2024-03-13 21:35:26.155,2024-03-13 21:35:26.155
	Robot-John,Place,wp:Box,1,1,2024-03-13 21:35:31.349,2024-03-13 21:35:39.349
	Robot-John,Signal,GO,2,1,2024-03-13 21:35:43.083,2024-03-13 21:35:49.621
	Robot-Steve,Pick,wp:Box,1,1,2024-03-13 21:35:58.881,2024-03-13 21:36:08.627
	Robot-Steve,Signal,GO,2,1,2024-03-13 21:36:14.937,2024-03-13 21:36:22.135
	Robot-Bob,Check,wp:Box,1,1,2024-03-13 21:36:26.821,2024-03-13 21:36:31.953
	Robot-John,Paint,wp:Box,3,1,2024-03-13 21:36:26.821,2024-03-13 21:36:31.953
Trace 2	Operator,Signal,START,0,2,2024-03-13 21:36:50.469,2024-03-13 21:36:50.469
	Robot-John,Place,wp:Box,1,2,2024-03-13 21:36:54.525,2024-03-13 21:37:00.995
	Robot-John,Signal,GO,2,2,2024-03-13 21:37:09.673,2024-03-13 21:37:14.681
	Robot-Steve,Pick,wp:Box,1,2,2024-03-13 21:37:19.851,2024-03-13 21:37:26.309
	Robot-Steve,Signal,GO,2,2,2024-03-13 21:37:35.883,2024-03-13 21:37:43.249
	Robot-John,Paint,wp:Box,3,2,2024-03-13 21:37:51.087,2024-03-13 21:37:56.973
	Robot-Bob,Check,wp:Box,1,2,2024-03-13 21:37:51.087,2024-03-13 21:37:56.973
Trace 3	Operator,Signal,START,0,3,2024-03-13 21:38:15.869,2024-03-13 21:38:15.869
	Robot-John,Place,wp:Box,1,3,2024-03-13 21:38:23.299,2024-03-13 21:38:32.153
	Robot-John,Signal,GO,2,3,2024-03-13 21:38:39.593,2024-03-13 21:38:45.475
	Robot-Steve,Pick,wp:Box,1,3,2024-03-13 21:38:55.353,2024-03-13 21:39:05.213
	Robot-Steve,Signal,GO,2,3,2024-03-13 21:39:14.661,2024-03-13 21:39:23.899
	Robot-Bob,Check,wp:Box,1,3,2024-03-13 21:39:31.001,2024-03-13 21:39:39.565
	Robot-John,Paint,wp:Box,3,3,2024-03-13 21:39:31.001,2024-03-13 21:39:39.565

After

ActorID,Activity,Target,EventID,CaseID,Start,End
Operator,Signal,START,0,1,2024-03-13 21:35:26.155,2024-03-13 21:35:26.155
Robot-John,Place,wp:Box,1,1,2024-03-13 21:35:31.349,2024-03-13 21:35:39.349
Robot-John,Signal,GO,2,1,2024-03-13 21:35:43.083,2024-03-13 21:35:49.621
Robot-Steve,Pick,wp:Box,1,1,2024-03-13 21:35:58.881,2024-03-13 21:36:08.627
Robot-Steve,Signal,GO,2,1,2024-03-13 21:36:14.937,2024-03-13 21:36:22.135
Robot-Bob,Check,wp:Box,1,1,2024-03-13 21:36:26.821,2024-03-13 21:36:31.953
Robot-John,Paint,wp:Box,3,1,2024-03-13 21:36:26.821,2024-03-13 21:36:31.953
Operator,Signal,START,0,2,2024-03-13 21:36:50.469,2024-03-13 21:36:50.469
Robot-John,Place,wp:Box,1,2,2024-03-13 21:36:54.525,2024-03-13 21:37:00.995
Robot-John,Error, ,E2,2,2024-03-13 21:37:09.673,2024-03-13 21:37:14.681
~~Robot-Steve,Pick,wp:Box,1,2,2024-03-13 21:37:19.851,2024-03-13 21:37:26.309~~
~~Robot-Steve,Signal,GO,2,2,2024-03-13 21:37:35.883,2024-03-13 21:37:43.249~~
~~Robot-John,Paint,wp:Box,3,2,2024-03-13 21:37:51.087,2024-03-13 21:37:56.973~~
~~Robot-Bob,Check,wp:Box,1,2,2024-03-13 21:37:51.087,2024-03-13 21:37:56.973~~
Operator,Signal,START,0,3,2024-03-13 21:38:15.869,2024-03-13 21:38:15.869
Robot-John,Place,wp:Box,1,3,2024-03-13 21:38:23.299,2024-03-13 21:38:32.153
Robot-John,Signal,GO,2,3,2024-03-13 21:38:39.593,2024-03-13 21:38:45.475
Robot-Steve,Pick,wp:Box,1,3,2024-03-13 21:38:55.353,2024-03-13 21:39:05.213
Robot-Steve,Signal,GO,2,3,2024-03-13 21:39:14.661,2024-03-13 21:39:23.899
Robot-Bob,Check,wp:Box,1,3,2024-03-13 21:39:31.001,2024-03-13 21:39:39.565
Robot-John,Paint,wp:Box,3,3,2024-03-13 21:39:31.001,2024-03-13 21:39:39.565

Figure 7.40: Adapting a generated CSV file to include an error case.

Uploading the altered CSV file in the CPM app yields the process model shown in Figure 7.41.

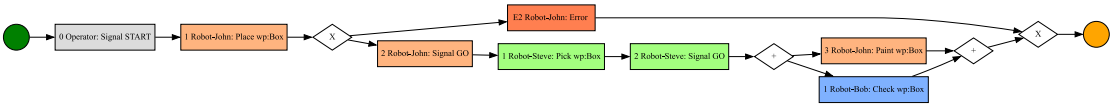


Figure 7.41: Resulting BPMN model of a custom example of a cooperation collaboration process with an error case where the other events in the affected trace were removed.

If the remaining events in the error case are not removed, the resulting process model looks as shown in Figure 7.42. This would indicate that there is little or no codependency among the individual tasks. The behavior and task flow in cases with error occurrences might be an interesting additional aspect to consider when determining the collaboration pattern. However, that is beyond the scope of this thesis and may be considered for future work.

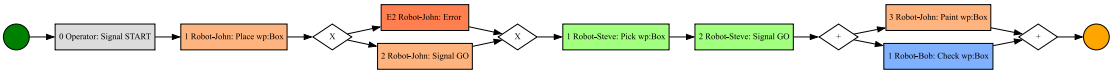


Figure 7.42: Resulting BPMN model of a custom example of a cooperation collaboration process with an error case where the other events in the affected trace were not removed.

Custom Supportive Collaboration Example

The third and final example is a custom process describing the highest level of collaboration: supportive collaboration.

- **Input process:**

  John Prep wp:Box->John Signal GO->Sam Place wp:Box->Sam Signal READY->[Steve Pick wp:Box,John Pick Item]->Steve Signal READY->John Place Item->John Signal FIN

- **Number of traces:**

  5

- **The generated CSV file:**

ActorID	Activity	Target	EventID	CaseID	Start	End
Operator	Signal	START	0	1	2024-03-14 22:51:27.843	2024-03-14 22:51:27.843
Robot-John	Prep	wp:Box	1	1	2024-03-14 22:51:32.939	2024-03-14 22:51:42.893
Robot-John	Signal	GO	2	1	2024-03-14 22:51:49.199	2024-03-14 22:51:52.417
Robot-Sam	Place	wp:Box	1	1	2024-03-14 22:51:56.599	2024-03-14 22:52:03.433
Robot-Sam	Signal	READY	2	1	2024-03-14 22:52:11.249	2024-03-14 22:52:20.985
Robot-Steve	Pick	wp:Box	1	1	2024-03-14 22:52:28.431	2024-03-14 22:52:31.885
Robot-John	Pick	Item	3	1	2024-03-14 22:52:28.431	2024-03-14 22:52:31.885
Robot-Steve	Signal	READY	2	1	2024-03-14 22:52:44.337	2024-03-14 22:52:48.487
Robot-John	Place	Item	4	1	2024-03-14 22:52:54.931	2024-03-14 22:53:02.625
Robot-John	Signal	FIN	5	1	2024-03-14 22:53:09.191	2024-03-14 22:53:13.891
Operator	Signal	START	0	2	2024-03-14 22:53:20.713	2024-03-14 22:53:20.713
Robot-John	Prep	wp:Box	1	2	2024-03-14 22:53:27.289	2024-03-14 22:53:29.525
Robot-John	Signal	GO	2	2	2024-03-14 22:53:32.115	2024-03-14 22:53:37.581
Robot-Sam	Place	wp:Box	1	2	2024-03-14 22:53:43.813	2024-03-14 22:53:52.787
Robot-Sam	Signal	READY	2	2	2024-03-14 22:53:57.545	2024-03-14 22:54:02.969
Robot-John	Pick	Item	3	2	2024-03-14 22:54:05.617	2024-03-14 22:54:09.233
Robot-Steve	Pick	wp:Box	1	2	2024-03-14 22:54:05.617	2024-03-14 22:54:09.233
Robot-Steve	Signal	READY	2	2	2024-03-14 22:54:28.861	2024-03-14 22:54:32.513
Robot-John	Place	Item	4	2	2024-03-14 22:54:38.979	2024-03-14 22:54:48.021
Robot-John	Signal	FIN	5	2	2024-03-14 22:54:52.515	2024-03-14 22:54:59.103
Operator	Signal	START	0	3	2024-03-14 22:55:06.779	2024-03-14 22:55:06.779
Robot-John	Prep	wp:Box	1	3	2024-03-14 22:55:13.207	2024-03-14 22:55:21.145
Robot-John	Signal	GO	2	3	2024-03-14 22:55:26.795	2024-03-14 22:55:33.531
Robot-Sam	Place	wp:Box	1	3	2024-03-14 22:55:39.869	2024-03-14 22:55:46.711
Robot-Sam	Signal	READY	2	3	2024-03-14 22:55:52.785	2024-03-14 22:55:58.449
Robot-Steve	Pick	wp:Box	1	3	2024-03-14 22:56:04.993	2024-03-14 22:56:07.837
Robot-John	Pick	Item	3	3	2024-03-14 22:56:04.993	2024-03-14 22:56:07.837
Robot-Steve	Signal	READY	2	3	2024-03-14 22:56:24.787	2024-03-14 22:56:30.235
Robot-John	Place	Item	4	3	2024-03-14 22:56:38.157	2024-03-14 22:56:42.765
Robot-John	Signal	FIN	5	3	2024-03-14 22:56:45.685	2024-03-14 22:56:54.359
Operator	Signal	START	0	4	2024-03-14 22:57:01.781	2024-03-14 22:57:01.781
Robot-John	Prep	wp:Box	1	4	2024-03-14 22:57:10.099	2024-03-14 22:57:12.635
Robot-John	Signal	GO	2	4	2024-03-14 22:57:15.613	2024-03-14 22:57:20.189
Robot-Sam	Place	wp:Box	1	4	2024-03-14 22:57:27.363	2024-03-14 22:57:34.659
Robot-Sam	Signal	READY	2	4	2024-03-14 22:57:40.499	2024-03-14 22:57:46.667
Robot-John	Pick	Item	3	4	2024-03-14 22:57:53.939	2024-03-14 22:57:57.521
Robot-Steve	Pick	wp:Box	1	4	2024-03-14 22:57:53.939	2024-03-14 22:57:57.521
Robot-Steve	Signal	READY	2	4	2024-03-14 22:58:11.889	2024-03-14 22:58:14.413
Robot-John	Place	Item	4	4	2024-03-14 22:58:18.269	2024-03-14 22:58:20.383
Robot-John	Signal	FIN	5	4	2024-03-14 22:58:24.897	2024-03-14 22:58:27.343
Operator	Signal	START	0	5	2024-03-14 22:58:35.637	2024-03-14 22:58:35.637
Robot-John	Prep	wp:Box	1	5	2024-03-14 22:58:42.159	2024-03-14 22:58:47.253
Robot-John	Signal	GO	2	5	2024-03-14 22:58:53.595	2024-03-14 22:59:02.979
Robot-Sam	Place	wp:Box	1	5	2024-03-14 22:59:10.049	2024-03-14 22:59:19.555
Robot-Sam	Signal	READY	2	5	2024-03-14 22:59:22.871	2024-03-14 22:59:27.203
Robot-Steve	Pick	wp:Box	1	5	2024-03-14 22:59:32.425	2024-03-14 22:59:36.215
Robot-John	Pick	Item	3	5	2024-03-14 22:59:32.425	2024-03-14 22:59:36.215
Robot-Steve	Signal	READY	2	5	2024-03-14 22:59:54.207	2024-03-14 22:59:58.065
Robot-John	Place	Item	4	5	2024-03-14 23:00:06.779	2024-03-14 23:00:14.173
Robot-John	Signal	FIN	5	5	2024-03-14 23:00:23.485	2024-03-14 23:00:31.013

Figure 7.43 shows the correctly identified process and pattern in the CPM app. Notice how the high number of interactions (5) with respect to the number of participating cobots (3) results in this pattern being detected. Of course, the other crucial requirement (same workpiece in each of the cobots' operations) is also satisfied.

Note that, a scenario with the same number of exchanged signals in the process (5) but with strictly sequential operation of the cobots (as in the synchronized example) would still be detected as supportive collaboration: the reason is that this indicates that more information was exchanged among the cobots.

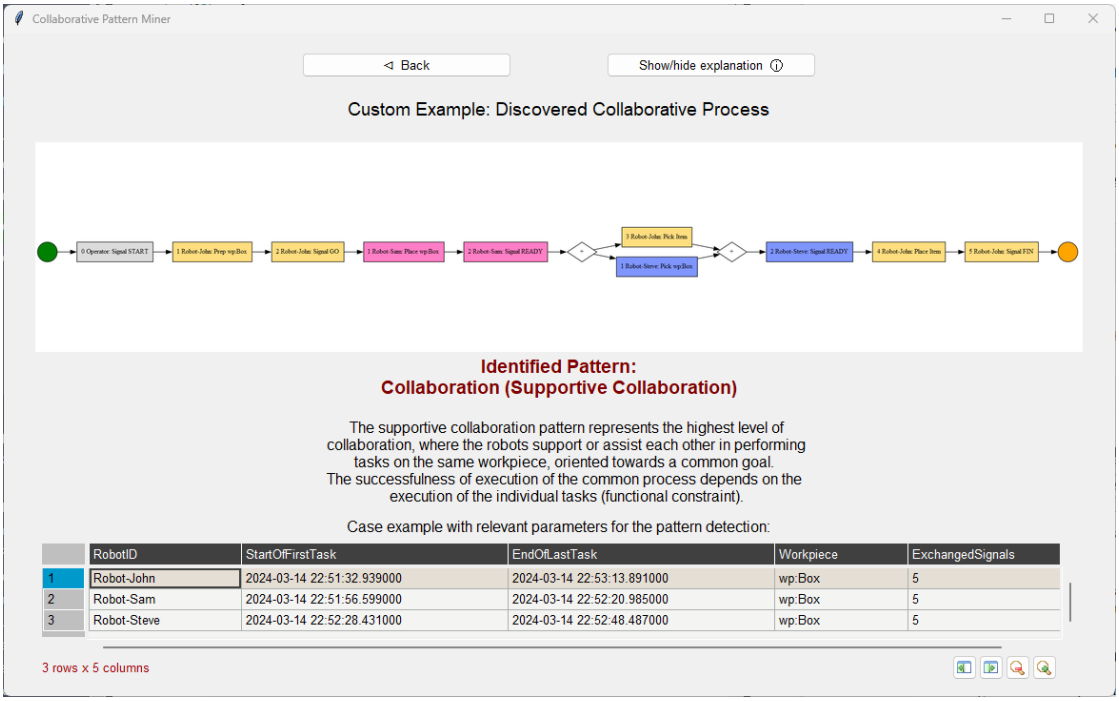


Figure 7.43: Resulting *process view* for a custom example of a supportive collaboration process in the CPM app.

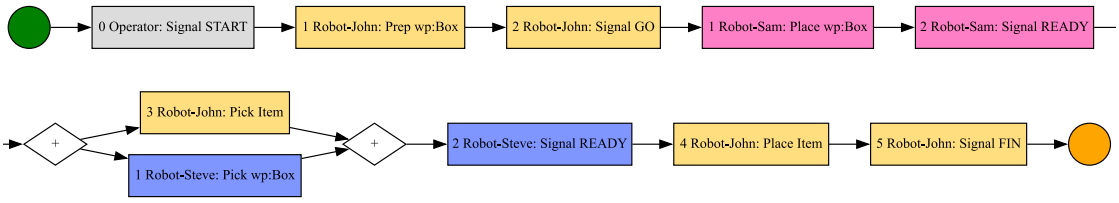


Figure 7.44: Resulting BPMN model of a custom example of a supportive collaboration process.

In the section that follows, a few observations regarding the scenario implementation and the outcomes of the CPM application are outlined.

7.6 Challenges and Lessons Learned

During the scenario implementation, some challenges arose. In terms of physical environment, the designed use cases were limited to the available cobot tools and the surrounding equipment. In particular, the capabilities of the available tools, the EPick vacuum gripper, and the 2F85 gripper are constrained by the tool properties and dimensions, as well as the kind of their operation. The EPick gripper's suction cup requires the workpiece to have an area of at least 1600mm^2 (40mm x 40mm) on the gripping surface, and it should be a flat, smooth surface, as we found that it does not grip well uneven or textured objects. Another restriction related to this is the width of the assembly lines (which are used by Robot A with the EPick gripper tool installed): the assembly lines allow for a maximum of 45mm object width. These are the reasons for using *Box-M* and the business cards (cf. Table 7.1) when implementing the tasks of Robot A in the scenarios. The 2F85 poses similar constraints. Specifically, its maximum grip width is 85mm, and all the objects used for its operation had to adhere to that. What is more, the slide, which is a rigid metal object with a smooth surface, introduced the challenge of objects slipping or rolling off of the work surface. For this reason, the Lego wall (cf. Table 7.1) was constructed to keep objects from sliding away.

The available equipment in the physical cobot cell used in this project is restricted to two cobots, and a single point of access (in terms of equipment that is not a teach pendant), namely the I/O signal panel mounted on the table of Robot A and connected to its control box. This setup is efficient for the simple case with a small number of cobots involved as presented in this project. However, with a large number cobots in, for example, a big factory setting, the propagation of the start signal from a signal access point may introduce a delay in terms of execution start time and thus a potential decrease in efficiency. Future work in this regard may be focused on researching possible architectures and signal handling for tackling this potential issue with cobots in a larger-scale setting.

Next, a challenge in this project also appeared in the logging mechanism of the cobots. In particular, the activities *during* the execution of a robot program do not appear in any of the files within the support file bundle[4]. This limitation required the use of custom log outputs using the `textmsg` function within the programs. Additionally, these custom logs had to be carefully designed to contain all necessary information for creating an XES-compliant dataframe (cf. Section 5.1.3).

The analysis and mining of the cobot logs could only be efficiently performed after thorough preparation and preprocessing of the logs. The preparation also includes the extensive research and manual analysis on the log data, as well as finding available information on the Internet with regards to logging mechanisms. As no information was found on the specific meanings of all the columns (which are not labeled) in the logs, manual analysis was conducted and column names inferred from the content. Despite the challenges, a combination of the already available data from the logs (such as timestamps and program start- and end indicators) and the custom log entries yielded sufficient information for building XES events, which in turn enabled successful process discovery and pattern recognition.

The results obtained with the CPM application based on the implementation of the algorithm

[4]Note that this claim refers to the time the log analysis was performed in this project (January, 2023) and may be subject to change in the future.

presented in Section 6.2 meet the expectations and seem to successfully identify the collaboration patterns in the scenarios with the given data. Further research could focus on trialing the application and the algorithm in cases with more than two cobots and/or operators.

8 Conclusion

In this thesis, a concept and a proof-of-concept implementation for mining cobot logs with the goal of identifying collaborative patterns are presented. More specifically, the entire research process including the initial research, the data gathering and analysis, the design and development of the concept, as well as the evaluation are described throughout this thesis.

The aim of this work was to find answers to the following research questions (RQs):

- **RQ1**: How is logging supported by a collaborative robot and how can it be transformed to reflect the cobot's activity during program execution?
- **RQ2**: How can a (collaborative) production process be discovered based on the cobot's logs?
- **RQ3**: What are the defining characteristics, constraints and attributes of collaborative patterns that can be recognized in the logs?
- **RQ4**: What conclusions can be drawn from evaluating the implemented approach on UR5e cobot logs?

The respective results can be summarized as follows.

RQ1. We enabled process-oriented logging mechanisms in cobots by implementing custom log outputs. The first big working block was aimed at the analysis of the logs produced by cobots in order to determine what is being logged and how, and especially, whether a cobot's activity during program execution is visible in the logs. The main finding, but simultaneously a challenge, was that only the start, end, and pause of a robot program are logged, while the only events logged during execution are errors. In other words, regular, programmed activities of a cobot executed when a program is running are not contained in the logs. This challenge was solved by utilizing the URScript `textmsg` function within the robot programs to produce custom log entries.

RQ2. We successfully discovered processes based on cobot logs using the inductive miner algorithm. The function for writing custom entries to cobot logs enabled the output of all relevant attributes for constructing valid process events as defined by the XES standard. Thus, process discovery was made possible by firstly defining the requirements for valid event attributes, and then designing the custom log entries according to those requirements. After the logs containing the custom entries are exported, they are read and parsed by the developed application which transforms the data and processes it further. Upon adequate transformation and formatting of the log data, the process discovery itself is performed using the inductive miner process discovery algorithm of the *pm4py* library.

RQ3. We identified collaborative patterns in the logs with a custom algorithm. In order to identify a collaborative pattern based on cobot logs, certain distinguishable attributes of the patterns had to be reflected in the logged events. Concretely, the idea was to specify attributes that can reflect the constraints posed by each of the patterns so that they can be recognized in a given use case or scenario. The attributes used for this purpose were actor IDs, workpiece annotations, number of exchanged signals, and timestamps. Based on the collaborative pattern definitions and the listed attributes, an algorithm for identifying collaborative patterns was developed and integrated into the proof-of-concept application. The algorithm is applicable for a work environment setup with N cobots and one operator.

RQ4. We evaluated the developed approach with a proof-of-concept implementation based on UR5e cobots. The designed and developed cobot log mining application was evaluated based on logs exported from two UR5e cobots, where four collaboration scenarios had been implemented. Each scenario was designed to demonstrate a specific collaborative pattern in the cobots' activities. The scenario programs were then executed on both cobots, where 50 iterations or runs of the programmed activities were performed per scenario. Afterwards, the support files containing the logs were exported from both cobots, prepared and subsequently stored in the *data* folder accessible to the mining application. Running the application with the logs from all four scenarios yielded the correct results with collaborative patterns that indeed occurred in the given cases.

8.1 Outlook and Future Work

As mentioned throughout this thesis, the analysis and the results obtained here are highly dependent on (1) the available equipment and consequently (2) the setup of the designed scenarios. In this project, two UR5e cobots by Universal Robots were available, and two Robotiq tools (end-effectors); namely the EPick gripper, and the 2F85 gripper. The rest of the available equipment such as the assembly lines and the slide, affected the design of the scenarios as well.

Thus, in the context of cobot log mining, it might be of interest to explore other cobot models with possibly different logging mechanisms, and different setups resembling other and/or more complex manufacturing use cases than those presented here.

In particular, the difference in setup, for instance, with two or more human operators instead of one, and multiple cobots collaborating with them, would provide interesting grounds for further exploration. Such more complex setups would require a certain amount of refactoring and adaptation in the pattern mining algorithm, as the signal flow (i.e., communication) among the participating actors would differ. This setup would resemble a real-life scenario with a longer manufacturing line, with multiple cobot and operator actors working alongside each other in a shared production space.

Another possible extension of this work would be to consider use cases where multiple collaboration patterns occur in a single process. Such an extension would encompass adapting the current version of the developed pattern identification algorithm, as well as possibly adding more attributes to the custom log outputs to facilitate distinguishing between patterns in a single process.

Furthermore, different cobot models may be used in such scenarios, depending on their tasks and respective capabilities. This, in turn, affects the produced logs: their structure and content, which needs to be taken into account when developing an analysis tool. A great challenge of the setup used and presented in this thesis was that the cobot model used does not provide information on the behavior (movement and signaling) during program execution in the log file, and thus the "artificial" solution of custom log outputs had to be implemented. Other cobot models may provide this kind of information out of the box, eliminating the need to manipulate and adapt the original logs.

Moreover, it might be of interest to explore the impact of different tools (end-effectors) on the quality of scenarios similar to the ones presented here. For instance, a camera or a motion sensor might be useful for implementing the constraint-checks for the cooperation (simultaneous) collaboration pattern, where spatial boundaries should be considered by the participating actors.

As the concept of analyzing cobot logs in a manner as described in this thesis - process discovery and pattern identification - is, at the time this thesis was composed, a new approach, there is a lot more left to be researched and explored.

Bibliography

[1] M. Atzmueller and B. Kloepper. Mining Attributed Interaction Networks on Industrial Event Logs. In H. Yin, D. Camacho, P. Novais, and A. J. Tallón-Ballesteros, editors, *Intelligent Data Engineering and Automated Learning – IDEAL 2018*, Lecture Notes in Computer Science, pages 94–102, Cham, 2018. Springer International Publishing. doi:10.1007/978-3-030-03496-2_11.

[2] L. T. Duong, L. Travé-Massuyès, A. Subias, and N. B. Roa. Assessing product quality from the production process logs. *The International Journal of Advanced Manufacturing Technology*, 117(5):1615–1631, Nov. 2021. doi:10.1007/s00170-021-07764-2.

[3] Fraunhofer Institute for Applied Information Technology. Example xes, 2023. Last accessed: 11.09.2023. URL: https://pm4py.fit.fraunhofer.de/static/assets/data/getting_started/running-example.xes.

[4] Fraunhofer Institute for Applied Information Technology. Getting Started: Importing Your First Log, 2023. Last accessed: 11.09.2023. URL: https://pm4py.fit.fraunhofer.de/getting-started-page.

[5] R. Galin and R. Meshcheryakov. Review on Human–Robot Interaction During Collaboration in a Shared Workspace. In A. Ronzhin, G. Rigoll, and R. Meshcheryakov, editors, *Interactive Collaborative Robotics*, Lecture Notes in Computer Science, pages 63–74, Cham, 2019. Springer International Publishing. doi:10.1007/978-3-030-26118-4_7.

[6] IEEE. IEEE Standard for eXtensible Event Stream (XES) for Achieving Interoperability in Event Logs and Event Streams. *IEEE Std 1849-2016*, pages 1–50, Nov. 2016. Conference Name: IEEE Std 1849-2016. doi:10.1109/IEEESTD.2016.7740858.

[7] Modbus Organization. MODBUS Messaging on TCP/IP Implementation Guide v1.0b, 2023. Last accessed: 13.09.2023. URL: https://www.modbus.org/docs/Modbus_Messaging_Implementation_Guide_V1_0b.pdf.

[8] NumFOCUS, Inc. pandas - Python Data Analysis Library, 2023. Last accessed: 13.09.2023. URL: https://pandas.pydata.org/.

[9] H. Oliff and Y. Liu. Towards Industry 4.0 Utilizing Data-Mining Techniques: A Case Study on Quality Improvement. *Procedia CIRP*, 63:167–172, Jan. 2017. URL: https://www.sciencedirect.com/science/article/pii/S2212827117304997, doi:10.1016/j.procir.2017.03.311.

[10] OMG. About the Business Process Model and Notation Specification Version 2.0, 2023. Last accessed: 11.09.2023. URL: **https://www.omg.org/spec/BPMN/2.0/**.

[11] Python Software Foundation. tkinter — Python interface to Tcl/Tk. **https://docs.python.org/3/library/tkinter.html**, 2023. Last accessed: 13.09.2023.

[12] Robotiq. TCP and Center of Mass of Robotiq Products. Technical report, Robotiq, 2023. **https://s3.amazonaws.com/com-robotiq-website-prod-assets/website-assets/support_documents/document/TCP_20and_20Center_20of_20Mass_20of_20Robotiq_20Products_20V1.1_20210406.pdf**. Last accessed: 10.09.2023.

[13] J. J. Roldán, M. A. Olivares-Méndez, J. del Cerro, and A. Barrientos. Analyzing and improving multi-robot missions by using process mining. *Autonomous Robots*, 42(6):1187–1205, Aug. 2018. **doi:10.1007/s10514-017-9686-1**.

[14] S. Samhaber and M. Leitner. Collaborative Patterns for Workflows with Collaborative Robots. In M. Sellami, P. Ceravolo, H. A. Reijers, W. Gaaloul, and H. Panetto, editors, *Cooperative Information Systems*, Lecture Notes in Computer Science, pages 131–148, Cham, 2022. Springer International Publishing. **doi:10.1007/978-3-031-17834-4_8**.

[15] F. Sherwani, M. M. Asad, and B. Ibrahim. Collaborative Robots and Industrial Revolution 4.0 (IR 4.0). In *2020 International Conference on Emerging Trends in Smart Technologies (ICETST)*, pages 1–5, Mar. 2020. **doi:10.1109/ICETST49965.2020.9080724**.

[16] Smartshift Robotics AS. *Technical Instructions: Smartshift Tool Change System*, 2019. **https://smartshift-robotics.com/wp-content/uploads/Technical-Instructions-Smartshift-Tool-Change-system.pdf**. Last accessed: 10.09.2023.

[17] A. Sulhi. Data Mining Technology Used in an Internet of Things-Based Decision Support System for Information Processing Intelligent Manufacturing. *International Journal of Informatics and Information Systems*, 4(3):168–179, Dec. 2021. Number: 3. URL: **http://ijiis.org/index.php/IJIIS/article/view/114**, **doi:10.47738/ijiis.v4i3.114**.

[18] Universal Robots. *The URScript Programming Language Manual*, 2021. **https://s3-eu-west-1.amazonaws.com/ur-support-site/18383/scriptmanual_en_1.3.pdf**. Last accessed: 10.09.2023.

[19] Universal Robots. *Universal Robots e-Series User Manual*, 2021. **https://s3-eu-west-1.amazonaws.com/ur-support-site/115737/99404_UR5e_User_Manual_en_Global.pdf**. Last accessed: 10.09.2023.

[20] Universal Robots. UR5 collaborative robot arm, 2022. Last accessed: 6.09.2022. URL: **https://www.universal-robots.com/products/ur5-robot/**.

[21] Universal Robots. Offline Simulator – E-series – UR Sim for Non Linux 5.9.4, 2023. Last accessed: 11.09.2023. URL: **https://www.universal-robots.com/download/software-e-series/simulator-non-linux/offline-simulator-e-series-ur-sim-for-non-linux-594/**.

[22] Universal Robots. Polyscope, 2023. Last accessed: 13.09.2023. URL: **https://www.universal-robots.com/products/polyscope/**.

[23] Universal Robots. Redefining automation: A collaborative ecosystem, 2023. Last accessed: 11.09.2023. URL: **https://www.universal-robots.com/redefining-automation-a-collaborative-ecosystem/**.

[24] Universal Robots. *UR Log Viewer - Manual*, 2023. Last accessed: 11.09.2023. URL: **https://www.universal-robots.com/articles/ur/robot-care-maintenance/ur-log-viewer-manual/**.

[25] W. van der Aalst. Process Mining: Overview and Opportunities. *ACM Transactions on Management Information Systems*, 3(2):1–17, July 2012. URL: **https://dl.acm.org/doi/10.1145/2229156.2229157**, **doi:10.1145/2229156.2229157**.

Acronyms

AA Apriori Algorithm. 7

ARM Association Rule Mining. 7

BPMN Business Process Model and Notation. 12, 14, 15

CPS Cyber-Physical System. 1

DSS Decision Support System. 7

GPL GNU General Public License. 42

GUI Graphical User Interface. 32, 33, 57

IoT Internet of Things. 1

PNG Portable Network Graphics. 48

SSH Secure Shell Protocol. 21

SVG Scalable Vector Graphics. 48

TCP Tool Center Point. 39

UR Universal Robots. 2, 3, 9, 57

VM Virtual Machine. 43, 75, 77

XES eXtensible Event Stream. 1, 9, 10, 24, 25, 105